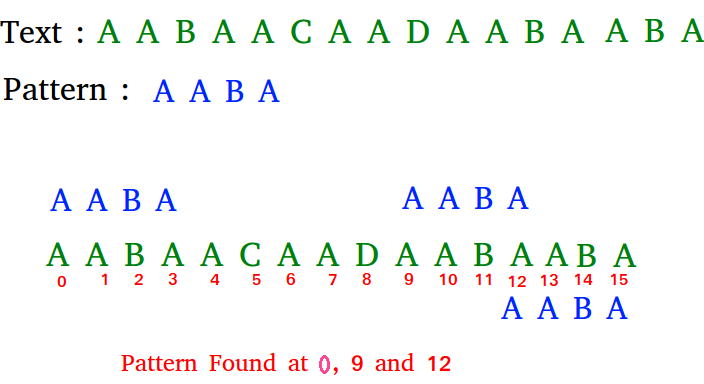
Pattern Searching

[**Learn more about Pattern Searching in DSA Self Paced course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=articles+pattern_searching_lp+header_link_click&utm_campaign=dsa+course+tracker)

[**Practice Problems on Pattern Searching**](https://practice.geeksforgeeks.org/explore/?page=1&category%5B%5D=Pattern%20Searching&utm_source=geeksforgeeks&utm_medium=articles+pattern_searching_lp+header_link_click&utm_campaign=practice+tracker)

[**Recent Articles on Pattern Searching**](https://www.geeksforgeeks.org/category/algorithm/pattern-searching/)

The Pattern Searching algorithms are sometimes also referred to as String Searching Algorithms and are considered as a part of the String algorithms. These algorithms are useful in the case of searching a string within another string.



**Topics :**

* [Introduction](https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/#introduction)
* [Some Standard algorithms](https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/#algo)
* [Some practice problems](https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/#practice)
* [Quick links](https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/#quick)

**Introduction:**

1. [Introduction to Pattern Searching – Data Structure and Algorithm Tutorial](https://www.geeksforgeeks.org/introduction-to-pattern-searching-data-structure-and-algorithm-tutorial/)
2. [Naive Pattern Searching](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/)

**Some Standard Algorithms:**

1. [Rabin-Karp Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/)
2. [KMP Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/)
3. [Z algorithm (Linear time pattern searching Algorithm)](https://www.geeksforgeeks.org/z-algorithm-linear-time-pattern-searching-algorithm/)
4. [Finite Automata](https://www.geeksforgeeks.org/searching-for-patterns-set-5-finite-automata/)
5. [Boyer Moore Algorithm – Bad Character Heuristic](https://www.geeksforgeeks.org/pattern-searching-set-7-boyer-moore-algorithm-bad-character-heuristic/)
6. [Aho-Corasick Algorithm for Pattern Searching](https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/)
7. [Suffix Array](https://www.geeksforgeeks.org/suffix-array-set-1-introduction/)
8. [kasai’s Algorithm for Construction of LCP array from Suffix Array](https://www.geeksforgeeks.org/%c2%ad%c2%adkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/)
9. [Online algorithm for checking palindrome in a stream](https://www.geeksforgeeks.org/online-algorithm-for-checking-palindrome-in-a-stream/)
10. [Manacher’s Algorithm – Linear Time Longest Palindromic Substring – Part 4](https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-4/)
11. [Ukkonen’s Suffix Tree Construction – Part 1](https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/)
12. [Generalized Suffix Tree](https://www.geeksforgeeks.org/generalized-suffix-tree)

**Some Practice problems:**

1. [Pattern Searching using C++ library](https://www.geeksforgeeks.org/pattern-searching-using-c-library/)
2. [Anagram Substring Search (Or Search for all permutations)](https://www.geeksforgeeks.org/anagram-substring-search-search-permutations/)
3. [Pattern Searching using a Trie of all Suffixes](https://www.geeksforgeeks.org/pattern-searching-using-trie-suffixes/)
4. [Dynamic Programming | Wildcard Pattern Matching | Linear Time and Constant Space](https://www.geeksforgeeks.org/dynamic-programming-wildcard-pattern-matching-linear-time-constant-space/)
5. [Longest prefix which is also suffix](https://www.geeksforgeeks.org/longest-prefix-also-suffix/)
6. [Count of number of given string in 2D character array](https://www.geeksforgeeks.org/find-count-number-given-string-present-2d-character-array/)
7. [Find all the patterns of “1(0+)1” in a given string (General Approach)](https://www.geeksforgeeks.org/find-all-the-pat%E2%80%A6general-approach/)
8. [Maximum length prefix of one string that occurs as subsequence in another](https://www.geeksforgeeks.org/maximum-length-prefix-one-string-occurs-subsequence-another/)
9. [Wildcard Pattern Matching](https://www.geeksforgeeks.org/wildcard-pattern-matching/)
10. [Search a Word in a 2D Grid of characters](https://www.geeksforgeeks.org/search-a-word-in-a-2d-grid-of-characters/)
11. [String matching where one string contains wildcard characters](https://www.geeksforgeeks.org/wildcard-character-matching/)
12. [Suffix Tree Application 1 – Substring Check](https://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/)

**Easy Questions:**

**Anagram Substring Search (Or Search for all permutations)**

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] and its permutations (or anagrams) in txt[]. You may assume that n > m.

Expected time complexity is O(n)

**Examples:**

1) Input: txt[] = "BACDGABCDA" pat[] = "ABCD"  
 Output: Found at Index 0  
 Found at Index 5  
 Found at Index 6  
2) Input: txt[] = "AAABABAA" pat[] = "AABA"  
 Output: Found at Index 0  
 Found at Index 1  
 Found at Index 4

[We strongly recommend that you click here and practice it, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/count-occurences-of-anagrams5839/1)

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](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/), [Rabin Karp](https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/), [Boyer Moore](https://www.geeksforgeeks.org/pattern-searching-set-7-boyer-moore-algorithm-bad-character-heuristic/), etc.

**Approach 1 :**

**Brute Force :**   
Consider the Input txt[] = "BACDGABCDA" pat[] = "ABCD".  
Occurrences of the pat[] and its permutations are found at indexes 0,5,6.   
The permutations are BACD,ABCD,BCDA.   
Let's sort the pat[] and the permutations of pat[] in txt[].  
pat[] after sorting becomes : ABCD  
permutations of pat[] in txt[] after sorting becomes : ABCD, ABCD,ABCD.  
So we can say that the sorted version of pat[] and sorted version of its  
permutations yield the same result.

**INTUITION:**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.

# Python code for the approach

**def** search(pat, txt):

  # finding lengths of strings pat and txt

  n **=** len(txt)

  m **=** len(pat);

  # string sortedpat stores the sorted version of pat

  sortedpat **=** pat;

  sortedpat **=** list(sortedpat);

  sortedpat.sort()

  sortedpat **=** ' '.join([str(elem) **for** elem **in** sortedpat])

  # temp for storing the substring of length equal to pat

**for** i **in** range(0,n**-**m**+**1):

    temp **=** txt[i:i**+**m]

    temp **=** list(temp);

    temp.sort()

    temp **=** ' '.join([str(elem) **for** elem **in** temp])

    # checking whether sorted versions are equal or not

**if** (sortedpat **==** temp):

**print**("Found at Index ",i);

# driver code

txt **=** "BACDGABCDA";

pat **=** "ABCD";

search(pat, txt);

# This code is contributed by kothavvsaakash

**Output**

Found at Index 0  
Found at Index 5  
Found at Index 6

**Time Complexity : O(mlogm) + O( (n-m+1)(m + mlogm + m) )**

mlogm for sorting pat. So **O(mlogm)**

The for loop runs for**n-m+1**times in each iteration we build string temp, which takes **O(m)**time, and sorting temp, which takes **O(mlogm)** time, and comparing sorted pat and sorted substring, which takes **O(m)**. So time complexity is **O( (n-m+1)\*(m+mlogm+m) )**

Total Time complexity :  **O(mlogm) + O( (n-m+1)(m + mlogm + m) )**

**Space Complexity: O(m)** As we are using Extra space for strings temp and sortedpat

**Approach 2 :**

The idea is to modify [Rabin Karp Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/). For example, we can keep the hash value as sum of ASCII values of all characters under modulo of a big prime number. For every character of text, we can add the current character to hash value and subtract the first character of previous window. This solution looks good, but like standard Rabin Karp, the worst case time complexity of this solution is O(mn). The worst case occurs when all hash values match and we one by one match all characters.

We can achieve O(n) time complexity under the assumption that alphabet size is fixed which is typically true as we have maximum 256 possible characters in ASCII. The idea is to use two count arrays:

1. The first count array store frequencies of characters in pattern.
2. The second count array stores frequencies of characters in current window of text.

The important thing to note is, time complexity to compare two count arrays is O(1) as the number of elements in them are fixed (independent of pattern and text sizes). Following are steps of this algorithm.

1. Store counts of frequencies of pattern in first count array *countP[]*. Also store counts of frequencies of characters in first window of text in array *countTW[]*.
2. Now run a loop from i = M to N-1. Do following in loop.

* If the two count arrays are identical, we found an occurrence.
* Increment count of current character of text in countTW[]
* Decrement count of first character in previous window in countWT[]

1. The last window is not checked by above loop, so explicitly check it.

Following is the implementation of above algorithm.

**Implementation:**

# Python program to search all

# anagrams of a pattern in a text

MAX**=**256

# This function returns true

# if contents of arr1[] and arr2[]

# are same, otherwise false.

**def** compare(arr1, arr2):

**for** i **in** range(MAX):

**if** arr1[i] !**=** arr2[i]:

**return** False

**return** True

# This function search for all

# permutations of pat[] in txt[]

**def** search(pat, txt):

    M **=** len(pat)

    N **=** len(txt)

    # countP[]:  Store count of

    # all characters of pattern

    # countTW[]: Store count of

    # current window of text

    countP **=** [0]**\***MAX

    countTW **=** [0]**\***MAX

**for** i **in** range(M):

        (countP[ord(pat[i]) ]) **+=** 1

        (countTW[ord(txt[i]) ]) **+=** 1

    # Traverse through remaining

    # characters of pattern

**for** i **in** range(M,N):

        # Compare counts of current

        # window of text with

        # counts of pattern[]

**if** compare(countP, countTW):

**print**("Found at Index", (i**-**M))

        # Add current character to current window

        (countTW[ ord(txt[i]) ]) **+=** 1

        # Remove the first character of previous window

        (countTW[ ord(txt[i**-**M]) ]) **-=** 1

    # Check for the last window in text

**if** compare(countP, countTW):

**print**("Found at Index", N**-**M)

# Driver program to test above function

txt **=** "BACDGABCDA"

pat **=** "ABCD"

search(pat, txt)

# This code is contributed

# by Upendra Singh Bartwal

**Output**

Found at Index 0  
Found at Index 5  
Found at Index 6

**Time Complexity:** O(256 \* (n – m) + m)

**Auxiliary space:** O(m), where m is 256

**Pattern Searching using a Trie of all Suffixes**

Problem Statement: Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

As discussed in the [previous post](https://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/), we discussed that there are two ways efficiently solve the above problem.

**1)** Preprocess Pattern: [KMP Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/), [Rabin Karp Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/), [Finite Automata](https://www.geeksforgeeks.org/searching-for-patterns-set-5-finite-automata/), [Boyer Moore Algorithm](https://www.geeksforgeeks.org/pattern-searching-set-7-boyer-moore-algorithm-bad-character-heuristic/).

**2)** Preprocess Text: [Suffix Tree](https://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/)

The best possible time complexity achieved by first (preprocessing pattern) is O(n) and by second (preprocessing text) is O(m) where m and n are lengths of pattern and text respectively.

Note that the second way does the searching only in O(m) time and it is preferred when text doesn’t change very frequently and there are many search queries. We have discussed [Suffix Tree (A compressed Trie of all suffixes of Text)](https://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/).

Implementation of Suffix Tree may be time consuming for problems to be coded in a technical interview or programming contexts. In this post simple implementation of a [Standard Trie](https://www.geeksforgeeks.org/trie-insert-and-search/) of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it’s a [simple Trie](https://www.geeksforgeeks.org/trie-insert-and-search/) instead of compressed Trie.

As discussed in [Suffix Tree](https://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/) post, the idea is, every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. So if we build a Trie of all suffixes, we can find the pattern in O(m) time where m is pattern length.

**Building a Trie of Suffixes**

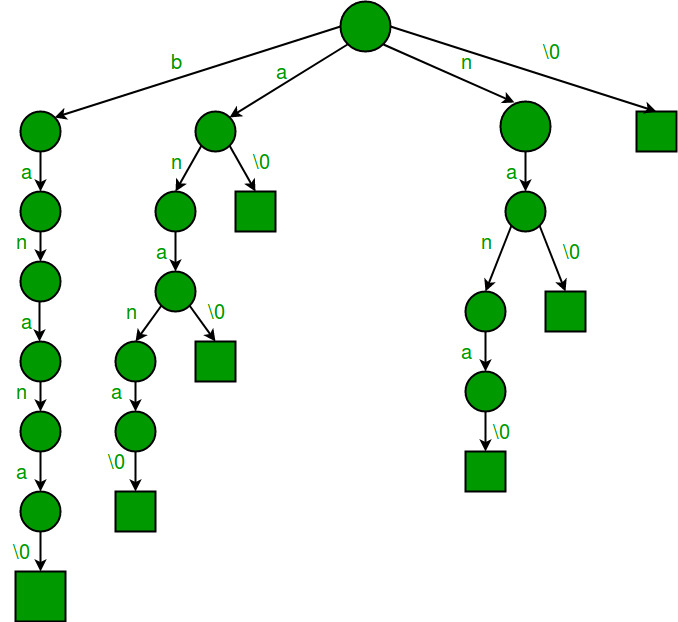
1) Generate all suffixes of given text.

2) Consider all suffixes as individual words and build a trie.

Let us consider an example text “banana\0” where ‘\0’ is string termination character. Following are all suffixes of “banana\0”

banana\0  
anana\0  
nana\0  
ana\0  
na\0  
a\0  
\0

If we consider all of the above suffixes as individual words and build a Trie, we get following.



**How to search a pattern in the built Trie?**

Following are steps to search a pattern in the built Trie.

**1)** Starting from the first character of the pattern and root of the Trie, do following for every character.

…..**a)** For the current character of pattern, if there is an edge from the current node, follow the edge.

…..**b)** If there is no edge, print “pattern doesn’t exist in text” and return.

**2)** If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print all indexes where pattern is present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

Following is the implementation of the above idea.

**class** SuffixTrieNode:

**def** \_\_init\_\_(self):

        self.children **=** [None] **\*** 256

        self.indexes **=** []

**def** insert\_suffix(self, suffix, index):

        self.indexes.append(index)

**if** suffix:

            c\_index **=** ord(suffix[0])

**if not** self.children[c\_index]:

                self.children[c\_index] **=** SuffixTrieNode()

            self.children[c\_index].insert\_suffix(suffix[1:], index **+** 1)

**def** search(self, pat):

**if not** pat:

**return** self.indexes

        c\_index **=** ord(pat[0])

**if** self.children[c\_index]:

**return** self.children[c\_index].search(pat[1:])

**return** None

**class** SuffixTrie:

**def** \_\_init\_\_(self, txt):

        self.root **=** SuffixTrieNode()

**for** i **in** range(len(txt)):

            self.root.insert\_suffix(txt[i:], i)

**def** search(self, pat):

        result **=** self.root.search(pat)

**if not** result:

            print("Pattern not found")

**else**:

            pat\_len **=** len(pat)

**for** i **in** result:

**print**(f"Pattern found at position {i - pat\_len}")

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Let us build the suffix trie for text "geeksforgeeks.org"

    txt **=** "geeksforgeeks.org"

    st **=** SuffixTrie(txt)

    # Let us search for different patterns

    pat **=** "ee"

**print**(f"Search for '{pat}'")

    st.search(pat)

    print()

    pat **=** "geek"

**print**(f"Search for '{pat}'")

    st.search(pat)

**print**()

    pat **=** "quiz"

**print**(f"Search for '{pat}'")

    st.search(pat)

**print**()

    pat **=** "forgeeks"

**print**(f"Search for '{pat}'")

    st.search(pat)

**print**()

**Output:**

Search for 'ee'  
Pattern found at position 1  
Pattern found at position 9

Search for 'geek'  
Pattern found at position 0  
Pattern found at position 8

Search for 'quiz'  
Pattern not found

Search for 'forgeeks'  
Pattern found at position 5

Time Complexity of the above search function is O(m+k) where m is length of the pattern and k is the number of occurrences of pattern in text.

Space Complexity: O(n \* MAX\_CHAR) where n is the length of the input text.

**Dynamic Programming | Wildcard Pattern Matching | Linear Time and Constant Space**

Given a text and a wildcard pattern, find if wildcard pattern is matched with text. The matching should cover the entire text (not partial text).

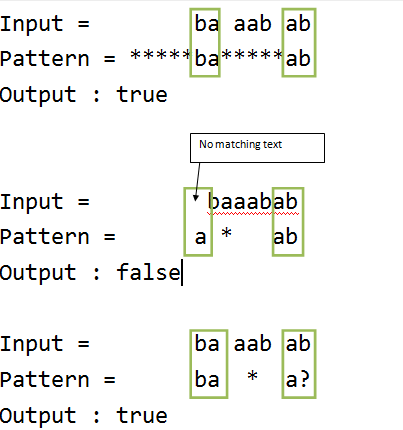
The wildcard pattern can include the characters ‘?’ and ‘\*’:

1. ‘?’ – matches any single character
2. ‘\*’ – Matches any sequence of characters (including the empty sequence)

**Pre-requisite:**[Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/) | [Wildcard Pattern Matching](https://www.geeksforgeeks.org/wildcard-pattern-matching/)

**Examples:**

Text = "baaabab",  
Pattern = “\*\*\*\*\*ba\*\*\*\*\*ab", output : true  
Pattern = "baaa?ab", output : true  
Pattern = "ba\*a?", output : true  
Pattern = "a\*ab", output : false



Each occurrence of ‘?’ character in wildcard pattern can be replaced with any other character and each occurrence of ‘\*’ with a sequence of characters such that the wildcard pattern becomes identical to the input string after replacement.

[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/wildcard-pattern-matching/1)

We have discussed a solution [here](https://www.geeksforgeeks.org/wildcard-pattern-matching/) which has O(m x n) time and O(m x n) space complexity.

For applying the optimization, we will at the first note the **BASE CASE** which says, if the length of the pattern is zero then answer will be true only if the length of the text with which we have to match the pattern is also zero.

***Algorithm:***

1. *Let i be the marker to point at the current character of the text.   
   Let j be the marker to point at the current character of the pattern.   
   Let index\_txt be the marker to point at the character of text on which we encounter ‘\*’ in the pattern.   
   Let index\_pat be the marker to point at the position of ‘\*’ in the pattern.*
2. *At any instant, if we observe that txt[i] == pat[j], then we increment both i and j as no operation needs to be performed in this case.*
3. *If we encounter pat[j] == ‘?’, then it resembles the case mentioned in step – (2) as ‘?’ has the property to match with any single character.*
4. *If we encounter pat[j] == ‘\*’, then we update the value of index\_txt and index\_pat as ‘\*’ has the property to match any sequence of characters (including the empty sequence) and we will increment the value of j to compare next character of pattern with the current character of the text. (As character represented by i has not been answered yet).*
5. *Now if txt[i] == pat[j], and we have encountered a ‘\*’ before, then it means that ‘\*’ included the empty sequence, else if txt[i] != pat[j], a character needs to be provided by ‘\*’ so that current character matching takes place, then i needs to be incremented as it is answered now but the character represented by j still needs to be answered, therefore, j = index\_pat + 1, i = index\_txt + 1 (as ‘\*’ can capture other characters as well), index\_txt++ (as current character in text is matched).*
6. *If step – (5) is not valid, that means txt[i] != pat[j], also we have not encountered a ‘\*’ that means it is not possible for the pattern to match the string. (return false).*
7. *Check whether j reached its final value or not, then return the final answer.*

**Let us see the above algorithm in action, then we will move to the coding section:**  
text = "baaabab"   
pattern = "\*\*\*\*\*ba\*\*\*\*\*ab"  
**NOW APPLYING THE ALGORITHM**  
Step - (1) : i = 0 (i --> 'b')   
j = 0 (j --> '\*')   
index\_txt = -1   
index\_pat = -1  
**NOTE: LOOP WILL RUN TILL i REACHES ITS FINAL**  
**VALUE OR THE ANSWER BECOMES FALSE MIDWAY.**  
**FIRST COMPARISON :-**  
As we see here that pat[j] == '\*', therefore directly jumping on to step - (4).   
Step - (4) : index\_txt = i (index\_txt --> 'b')   
index\_pat = j (index\_pat --> '\*')   
j++ (j --> '\*')  
After four more comparisons : i = 0 (i --> 'b')   
j = 5 (j --> 'b')   
index\_txt = 0 (index\_txt --> 'b')   
index\_pat = 4 (index\_pat --> '\*')  
**SIXTH COMPARISON :-**  
As we see here that txt[i] == pat[j], but we already encountered '\*' therefore using step - (5).   
Step - (5) : i = 1 (i --> 'a')   
j = 6 (j --> 'a')   
index\_txt = 0 (index\_txt --> 'b')   
index\_pat = 4 (index\_pat --> '\*')  
**SEVENTH COMPARISON :-**  
Step - (5) : i = 2 (i --> 'a')   
j = 7 (j --> '\*')   
index\_txt = 0 (index\_txt --> 'b')   
index\_pat = 4 (index\_pat --> '\*')  
**EIGHT COMPARISON :-**  
Step - (4) : i = 2 (i --> 'a')   
j = 8 (j --> '\*')   
index\_txt = 2 (index\_txt --> 'a')   
index\_pat = 7 (index\_pat --> '\*')  
After four more comparisons : i = 2 (i --> 'a')   
j = 12 (j --> 'a')   
index\_txt = 2 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**THIRTEENTH COMPARISON :-**  
Step - (5) : i = 3 (i --> 'a')   
j = 13 (j --> 'b')   
index\_txt = 2 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**FOURTEENTH COMPARISON :-**  
Step - (5) : i = 3 (i --> 'a')   
j = 12 (j --> 'a')   
index\_txt = 3 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**FIFTEENTH COMPARISON :-**  
Step - (5) : i = 4 (i --> 'b')   
j = 13 (j --> 'b')   
index\_txt = 3 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**SIXTEENTH COMPARISON :-**  
Step - (5) : i = 5 (i --> 'a')   
j = 14 (j --> end)   
index\_txt = 3 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**SEVENTEENTH COMPARISON :-**  
Step - (5) : i = 4 (i --> 'b')   
j = 12 (j --> 'a')   
index\_txt = 4 (index\_txt --> 'b')   
index\_pat = 11 (index\_pat --> '\*')  
**EIGHTEENTH COMPARISON :-**  
Step - (5) : i = 5 (i --> 'a')   
j = 12 (j --> 'a')   
index\_txt = 5 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**NINETEENTH COMPARISON :-**  
Step - (5) : i = 6 (i --> 'b')   
j = 13 (j --> 'b')   
index\_txt = 5 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**TWENTIETH COMPARISON :-**  
Step - (5) : i = 7 (i --> end)   
j = 14 (j --> end)   
index\_txt = 5 (index\_txt --> 'a')   
index\_pat = 11 (index\_pat --> '\*')  
**NOTE : NOW WE WILL COME OUT OF LOOP TO RUN STEP - 7.**  
Step - (7) : j is already present at its end position, therefore answer is true.

Below is the implementation of the above approach:

# Python3 program to implement

# wildcard pattern matching

# algorithm

# Function that matches input

# txt with given wildcard pattern

**def** stringmatch(txt, pat, n, m):

    # empty pattern can only

    # match with empty string

    # Base case

**if** (m **==** 0):

**return** (n **==** 0)

    # step 1

    # initialize markers :

    i **=** 0

    j **=** 0

    index\_txt **= -**1

    index\_pat **= -**1

**while**(i < n **-** 2):

        # For step - (2, 5)

**if** (j < m **and** txt[i] **==** pat[j]):

            i **+=** 1

            j **+=** 1

        # For step - (3)

**elif**(j < m **and** pat[j] **==** '?'):

            i **+=** 1

            j **+=** 1

        # For step - (4)

**elif**(j < m **and** pat[j] **==** '\*'):

            index\_txt **=** i

            index\_pat **=** j

            j **+=** 1

        # For step - (5)

**elif**(index\_pat !**= -**1):

            j **=** index\_pat **+** 1

            i **=** index\_txt **+** 1

            index\_txt **+=** 1

        # For step - (6)

**else**:

**return** False

    # For step - (7)

**while** (j < m **and** pat[j] **==** '\*'):

        j **+=** 1

    # Final Check

**if**(j **==** m):

**return** True

**return** False

# Driver code

strr **=** "baaabab"

pattern **=** "\*\*\*\*\*ba\*\*\*\*\*ab"

# char pattern[] = "ba\*\*\*\*\*ab"

# char pattern[] = "ba \* ab"

# char pattern[] = "a \* ab"

**if** (stringmatch(strr, pattern, len(strr),

                               len(pattern))):

    print("Yes")

**else**:

    print( "No")

pattern2 **=** "a\*\*\*\*\*ab";

**if** (stringmatch(strr, pattern2, len(strr),

                                len(pattern2))):

**print**("Yes")

**else**:

    print( "No")

# This code is contributed

# by sahilhelangia

# Python3 program to implement

# wildcard pattern matching

# algorithm

# Function that matches input

# txt with given wildcard pattern

**def** stringmatch(txt, pat, n, m):

    # empty pattern can only

    # match with empty string

    # Base case

**if** (m **==** 0):

**return** (n **==** 0)

    # step 1

    # initialize markers :

    i **=** 0

    j **=** 0

    index\_txt **= -**1

    index\_pat **= -**1

**while**(i < n **-** 2):

        # For step - (2, 5)

**if** (j < m **and** txt[i] **==** pat[j]):

            i **+=** 1

            j **+=** 1

        # For step - (3)

**elif**(j < m **and** pat[j] **==** '?'):

            i **+=** 1

            j **+=** 1

        # For step - (4)

**elif**(j < m **and** pat[j] **==** '\*'):

            index\_txt **=** i

            index\_pat **=** j

            j **+=** 1

        # For step - (5)

**elif**(index\_pat !**= -**1):

            j **=** index\_pat **+** 1

            i **=** index\_txt **+** 1

            index\_txt **+=** 1

        # For step - (6)

**else**:

**return** False

    # For step - (7)

**while** (j < m **and** pat[j] **==** '\*'):

        j **+=** 1

    # Final Check

**if**(j **==** m):

**return** True

**return** False

# Driver code

strr **=** "baaabab"

pattern **=** "\*\*\*\*\*ba\*\*\*\*\*ab"

# char pattern[] = "ba\*\*\*\*\*ab"

# char pattern[] = "ba \* ab"

# char pattern[] = "a \* ab"

**if** (stringmatch(strr, pattern, len(strr),

                               len(pattern))):

    print("Yes")

**else**:

    print( "No")

pattern2 **=** "a\*\*\*\*\*ab";

**if** (stringmatch(strr, pattern2, len(strr),

                                len(pattern2))):

**print**("Yes")

**else**:

    print( "No")

# This code is contributed

# by sahilhelangia

**Complexity Analysis:**

1. **Time Complexity:** O(m + n), where ‘m’ and ‘n’ are the lengths of text and pattern respectively.
2. **Auxiliary Space:**O(1).No use of any data structure for storing values, since no extra space has been taken.

**Longest prefix which is also suffix**

Given a string s, find the length of the longest prefix, which is also a suffix. The prefix and suffix should not overlap.

**Examples:**

Input : aabcdaabc  
Output : 4  
The string "aabc" is the longest  
prefix which is also suffix.

Input : abcab  
Output : 2

Input : aaaa  
Output : 2

Recommended Problem

Longest Prefix Suffix

**Simple Solution:**Since overlapping prefixes and suffixes is not allowed, we break the string from the middle and start matching left and right strings. If they are equal return size of one string, else they try for shorter lengths on both sides.

Below is a solution to the above approach!

# Python3 program to find length

# of the longest prefix which

# is also suffix

**def** longestPrefixSuffix(s) :

    n **=** len(s)

**for** res **in** range(n **//** 2, 0, **-**1) :

        # Check for shorter lengths

        # of first half.

        prefix **=** s[0: res]

        suffix **=** s[n **-** res: n]

**if** (prefix **==** suffix) :

**return** res

    # if no prefix and suffix match

    # occurs

**return** 0

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    s **=** "blablabla"

    print(longestPrefixSuffix(s))

# This code is contributed by Nikita Tiwari.

**Output:**

3

**Time Complexity:**O(n)

**Auxiliary Space:**O(1)

**Efficient Solution:**The idea is to use the preprocessing algorithm [KMP search](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/). In the preprocessing algorithm, we build lps array which stores the following values.

*lps[i] = the longest proper prefix of pat[0..i]*

*which is also a suffix of pat[0..i].*

# Efficient Python 3 program

# to find length of

# the longest prefix

# which is also suffix

# Returns length of the longest prefix

# which is also suffix and the two do

# not overlap. This function mainly is

# copy computeLPSArray() of in below post

# <https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/>

**def** longestPrefixSuffix(s) :

    n **=** len(s)

    lps **=** [0] **\*** n   # lps[0] is always 0

    # length of the previous

    # longest prefix suffix

    l **=** 0

    # the loop calculates lps[i]

    # for i = 1 to n-1

    i **=** (n**+**1)**//**2;

**while** (i < n) :

**if** (s[i] **==** s[l]) :

            l **=** l **+** 1

            lps[i] **=** l

            i **=** i **+** 1

**else** :

            # (pat[i] != pat[len])

            # This is tricky. Consider

            # the example. AAACAAAA

            # and i = 7. The idea is

            # similar to search step.

**if** (l !**=** 0) :

                l **=** lps[l**-**1]

                # Also, note that we do

                # not increment i here

**else** :

                # if (len == 0)

                lps[i] **=** 0

                i **=** i **+** 1

    res **=** lps[n**-**1]

    # Since we are looking for

    # non overlapping parts.

**return** res;

# Driver program to test above function

s **=** "bbabbabb"

print(longestPrefixSuffix(s))

# This code is contributed

# by Nikita Tiwari.

#Corrected by Nilanshu Yadav

**Output:**

2

**Time Complexity:**O(n)

**Auxiliary Space:**O(n)

Please refer computeLPSArray() of [KMP search](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/) for an explanation.

**Solution using RegEx:**

1. Python3

# Python code to find length of the longest

# prefix which is also suffix

**import** re

s **=** "ABCABCABCABCABC" # Example input

**print**(len(re.findall(r'^(\w\*).\*\1$',s)[0]))

**Output:**

6

**Efficient Solution:**The idea is to traverse the suffix in reverse order and try to find a match in first half of the string (possible prefix). Here we take advantage of the property of a prefix substring – when traversed in reverse order, the prefix substring’s last character will always terminate at the string’s beginning.

Please note that we search for the prefix in the first half of the string alone because of the constraint given in the problem that the prefix and suffix are non-overlapping.

**Algorithm :**

        1. Maintain two pointers –  one which starts at the end of string(for suffix) and one which starts at the middle of string(for prefix)

        2. Keep on decrementing both the pointers provided they match and the prefix pointer is not exhausted( >0) .

        3. When a mismatch occurs, reset the suffix pointer back to end of string and repeat step 2.

        4. When prefix pointer reaches ‘-1’ (i.e. string is exhausted) the longest common suffix/prefix will be the substring from suffix pointer

            to end of the string.  Return the length of this substring.

**Implementation:**

1. Python3

# Python3 program to find length

# of the longest prefix which

# is also suffix

# Returns length of the longest prefix

# which is also suffix and the two do

# not overlap.

**def** longestPrefixSuffix(s):

    n **=** len(s)

**if** n **==** 0:

**return** 0

    # end\_suffix and end\_prefix are used to keep track of the common suffix and prefix respectively.

    # For the prefix we search only in first half of string (0-->n//2-1) since

    # suffix and prefix do not overlap.

    end\_suffix **=** n**-**1

    end\_prefix **=** n **//** 2 **-** 1

    # Traverse each character of suffix from end to start and check for a match of prefix

    # in first half of array.

**while** end\_prefix >**=** 0:

**if** s[end\_prefix] !**=** s[end\_suffix]:

**if** end\_suffix !**=** n**-**1:

                end\_suffix **=** n**-**1  # reset end\_suffix

**else**:

                end\_prefix **-=** 1

**else**:

            end\_suffix **-=** 1

            end\_prefix **-=** 1

    # The longest common suffix and prefix is s[end+1:]

**return** n**-**end\_suffix**-**1

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    s **=** "ABCABCABCABCABC"

    print(longestPrefixSuffix(s))

# This code is contributed by Reshma Koshy.

**Output**

6

**Time Complexity:**O(n)

**Auxiliary Space:**O(1)

**Count of number of given string in 2D character array**

Given a 2-Dimensional character array and a string, we need to find the given string in 2-dimensional character array, such that individual characters can be present left to right, right to left, top to down or down to top.

**Examples:**

**Input :** a ={  
 {D,D,D,G,D,D},  
 {B,B,D,E,B,S},  
 {B,S,K,E,B,K},  
 {D,D,D,D,D,E},  
 {D,D,D,D,D,E},  
 {D,D,D,D,D,G}  
 }  
 str= "GEEKS"  
**Output :**2

**Input :** a = {  
 {B,B,M,B,B,B},  
 {C,B,A,B,B,B},  
 {I,B,G,B,B,B},  
 {G,B,I,B,B,B},  
 {A,B,C,B,B,B},  
 {M,C,I,G,A,M}  
 }  
 str= "MAGIC"

**Output :**3

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We have discussed simpler problem to [find if a word exists or not in a matrix](https://www.geeksforgeeks.org/search-a-word-in-a-2d-grid-of-characters/).

**Approach:**

1. To count all occurrences, we follow simple brute force approach.
2. Traverse through each character of the matrix and taking each character as a start of the string to be found.
3. Try to search in all the possible directions.
4. Whenever, a word is found, increase the count.
5. After traversing the matrix what ever will be the value of count will be number of times string exists in character matrix.

***Algorithm :***

* ***Step 1****– Traverse matrix character by character and take one character as string start*
* ***Step 2****– For each character find the string in all the four directions recursively*
* ***Step 3****– If a string found, we increase the count*
* ***Step 4****– When we are done with one character as start, we repeat the same process for the next character*
* ***Step 5****– Calculate the sum of count for each character*
* ***Step 6****– Final count will be the answer*

**Implementation:**

# Python code for finding count

# of string in a given 2D

# character array.

# utility function to search

# complete string from any

# given index of 2d array

**def** internalSearch(ii, needle, row, col, hay,

                    row\_max, col\_max):

    found **=** 0

**if** (row >**=** 0 **and** row <**=** row\_max **and**

        col >**=** 0 **and** col <**=** col\_max **and**

        needle[ii] **==** hay[row][col]):

        match **=** needle[ii]

        ii **+=** 1

        hay[row][col] **=** 0

**if** (ii **==** len(needle)):

            found **=** 1

**else**:

            # through Backtrack searching

            # in every directions

            found **+=** internalSearch(ii, needle, row,

                               col **+** 1, hay, row\_max, col\_max)

            found **+=** internalSearch(ii, needle, row,

                               col **-** 1, hay, row\_max, col\_max)

            found **+=** internalSearch(ii, needle, row **+** 1,

                               col, hay, row\_max, col\_max)

            found **+=** internalSearch(ii, needle, row **-** 1,

                               col, hay, row\_max, col\_max)

        hay[row][col] **=** match

**return** found

# Function to search the string in 2d array

**def** searchString(needle, row, col,strr,

                row\_count, col\_count):

    found **=** 0

**for** r **in** range(row\_count):

**for** c **in** range(col\_count):

            found **+=** internalSearch(0, needle, r, c,

                        strr, row\_count **-** 1, col\_count **-** 1)

**return** found

# Driver code

needle **=** "MAGIC"

inputt **=** ["BBABBM","CBMBBA","IBABBG",

            "GOZBBI","ABBBBC","MCIGAM"]

strr **=** [0] **\*** len(inputt)

**for** i **in** range(len(inputt)):

    strr[i] **=** list(inputt[i])

print("count: ", searchString(needle, 0, 0, strr,

                        len(strr), len(strr[0])))

# This code is contributed by SHUBHAMSINGH10

**Output**

count: 3

**Time Complexity:** **O(n\*m)^2,** where n is the row size and m is the column size.

**Auxiliary Space:** **O(n\*m)**

**Maximum length prefix of one string that occurs as subsequence in another**

Given two strings **s** and **t**. The task is to find maximum length of some prefix of the string S which occur in string t as subsequence.

**Examples :**

Input : s = "digger"  
 t = "biggerdiagram"  
Output : 3  
**dig**ger  
bigger**di**a**g**ram  
Prefix "dig" of s is longest subsequence in t.

Input : s = "geeksforgeeks"  
 t = "agbcedfeitk"  
Output : 4

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

A **simple solutions** is to consider all prefixes one by one and check if current prefix of s[] is a subsequence of t[] or not. Finally return length of the largest prefix.

An **efficient solution** is based on the fact that to find a prefix of length n, we must first find the prefix of length n – 1 and then look for s[n-1] in t. Similarly, to find a prefix of length n – 1, we must first find the prefix of length n – 2 and then look for s[n – 2] and so on.

Thus, we keep a counter which stores the current length of prefix found. We initialize it with 0 and begin with the first letter of s and keep iterating over t to find the occurrence of the first letter. As soon as we encounter the first letter of s we update the counter and look for second letter. We keep updating the counter and looking for next letter, until either the string s is found or there are no more letters in t.

Below is the implementation of this approach:

# Python 3 program to find maximum

# length prefix of one string occur

# as subsequence in another string.

# Return the maximum length

# prefix which is subsequence.

**def** maxPrefix(s, t) :

    count **=** 0

    # Iterating string T.

**for** i **in** range(0,len(t)) :

        # If end of string S.

**if** (count **==** len(s)) :

**break**

        # If character match,

        # increment counter.

**if** (t[i] **==** s[count]) :

            count **=** count **+** 1

**return** count

# Driver Code

S **=** "digger"

T **=** "biggerdiagram"

print(maxPrefix(S, T))

# This code is contributed

# by Nikita Tiwari.

**Output**

3

**Time complexity:** O(n)

**Space complexity:** O(1)

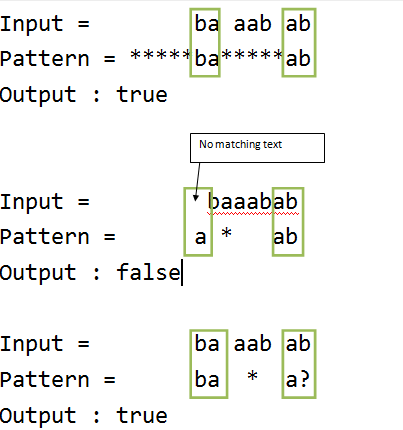
**Wildcard Pattern Matching**

Given a text and a wildcard pattern, implement wildcard pattern matching algorithm that finds if wildcard pattern is matched with text. The matching should cover the entire text (not partial text). The wildcard pattern can include the characters ‘?’ and ‘\*’

* ‘?’ – matches any single character
* ‘\*’ – Matches any sequence of characters (including the empty sequence)

For example:

Text = "baaabab",  
Pattern = “\*\*\*\*\*ba\*\*\*\*\*ab", output : true  
Pattern = "baaa?ab", output : true  
Pattern = "ba\*a?", output : true  
Pattern = "a\*ab", output : false



Each occurrence of ‘?’ character in wildcard pattern can be replaced with any other character and each occurrence of ‘\*’ with a sequence of characters such that the wildcard pattern becomes identical to the input string after replacement.

Let’s consider any character in the pattern.

**Case 1: The character is ‘\*’** . Here two cases arises as follows:

1. We can ignore ‘\*’ character and move to next character in the Pattern.
2. ‘\*’ character matches with one or more characters in Text. Here we will move to next character in the string.

**Case 2: The character is ‘?’**

We can ignore current character in Text and move to next character in the Pattern and Text.

**Case 3: The character is not a wildcard character**

If current character in Text matches with current character in Pattern, we move to next character in the Pattern and Text. If they do not match, wildcard pattern and Text do not match.

We can use Dynamic Programming to solve this problem:

Let **T[i][j]** is true if first i characters in given string matches the first j characters of pattern.

**DP Initialization:**

// both text and pattern are null  
T[0][0] = true;

// pattern is null  
T[i][0] = false;

// text is null  
T[0][j] = T[0][j - 1] if pattern[j – 1] is '\*'

**DP relation:**

// If current characters match, result is same as   
// result for lengths minus one. Characters match  
// in two cases:  
// a) If pattern character is '?' then it matches   
// with any character of text.   
// b) If current characters in both match  
if ( pattern[j – 1] == ‘?’) ||   
 (pattern[j – 1] == text[i - 1])  
 T[i][j] = T[i-1][j-1]   
   
// If we encounter ‘\*’, two choices are possible-  
// a) We ignore ‘\*’ character and move to next   
// character in the pattern, i.e., ‘\*’   
// indicates an empty sequence.  
// b) '\*' character matches with ith character in  
// input   
else if (pattern[j – 1] == ‘\*’)  
 T[i][j] = T[i][j-1] || T[i-1][j]

else // if (pattern[j – 1] != text[i - 1])  
 T[i][j] = false

**Implementation:**

Below is the implementation of the above dynamic programming approach.

# Python program to implement wildcard

# pattern matching algorithm

# Function that matches input strr with

# given wildcard pattern

**def** strrmatch(strr, pattern, n, m):

    # empty pattern can only match with

    # empty string

**if** (m **==** 0):

**return** (n **==** 0)

    # lookup table for storing results of

    # subproblems

    lookup **=** [[False **for** i **in** range(m **+** 1)] **for** j **in** range(n **+** 1)]

    # empty pattern can match with empty string

    lookup[0][0] **=** True

    # Only '\*' can match with empty string

**for** j **in** range(1, m **+** 1):

**if** (pattern[j **-** 1] **==** '\*'):

            lookup[0][j] **=** lookup[0][j **-** 1]

    # fill the table in bottom-up fashion

**for** i **in** range(1, n **+** 1):

**for** j **in** range(1, m **+** 1):

            # Two cases if we see a '\*'

            # a) We ignore ‘\*’ character and move

            # to next character in the pattern,

            # i.e., ‘\*’ indicates an empty sequence.

            # b) '\*' character matches with ith

            # character in input

**if** (pattern[j **-** 1] **==** '\*'):

                lookup[i][j] **=** lookup[i][j **-** 1] **or** lookup[i **-** 1][j]

            # Current characters are considered as

            # matching in two cases

            # (a) current character of pattern is '?'

            # (b) characters actually match

**else if** (pattern[j **-** 1] **==** '?' **or** strr[i **-** 1] **==** pattern[j **-** 1]):

                lookup[i][j] **=** lookup[i **-** 1][j **-** 1]

            # If characters don't match

**else**:

                lookup[i][j] **=** False

**return** lookup[n][m]

# Driver code

strr **=** "baaabab"

pattern **=** "\*\*\*\*\*ba\*\*\*\*\*ab"

# char pattern[] = "ba\*\*\*\*\*ab"

# char pattern[] = "ba\*ab"

# char pattern[] = "a\*ab"

# char pattern[] = "a\*\*\*\*\*ab"

# char pattern[] = "\*a\*\*\*\*\*ab"

# char pattern[] = "ba\*ab\*\*\*\*"

# char pattern[] = "\*\*\*\*"

# char pattern[] = "\*"

# char pattern[] = "aa?ab"

# char pattern[] = "b\*b"

# char pattern[] = "a\*a"

# char pattern[] = "baaabab"

# char pattern[] = "?baaabab"

# char pattern[] = "\*baaaba\*"

**if** (strrmatch(strr, pattern, len(strr), len(pattern))):

**print**("Yes")

**else**:

**print**("No")

# This code is contributed by shubhamsingh10

**Output**

Yes

***Time complexity:****O(m x n)*

***Auxiliary space:****O(m x n)*

**Approach: DP Memoization solution**

# Python program to implement wildcard

# pattern matching algorithm

**def** finding(s, p, n, m):

    # return 1 if n and m are negative

**if** n < 0 **and** m < 0:

**return** 1

    # return 0 if m is negative

**if** m < 0:

**return** 0

    # return n if n is negative

**if** n < 0:

        # while m is positive

**while** m >**=** 0:

**if** p[m] !**=** '\*':

**return** 0

            m **-=** 1

**return** 1

    # if dp state is not visited

**if** dp[n][m] **== -**1:

**if** p[m] **==** '\*':

            dp[n][m] **=** finding(s, p, n **-** 1, m) **or** finding(s, p, n, m **-** 1)

**return** dp[n][m]

**else**:

**if** p[m] !**=** s[n] **and** p[m] !**=** '?':

                dp[n][m] **=** 0

**return** dp[n][m]

**else**:

                dp[n][m] **=** finding(s, p, n **-** 1, m **-** 1)

**return** dp[n][m]

    # return dp[n][m] if dp state is previsited

**return** dp[n][m]

**def** isMatch(s, p):

**global** dp

    dp **=** []

    # resize the dp array

**for** i **in** range(len(s) **+** 1):

        dp.append([**-**1] **\*** (len(p) **+** 1))

    dp[len(s)][len(p)] **=** finding(s, p, len(s) **-** 1, len(p) **-** 1)

**return** dp[len(s)][len(p)]

# Driver code

**def** main():

    s **=** "baaabab"

    p **=** "\*\*\*\*\*ba\*\*\*\*\*ab"

    # p = "ba\*\*\*\*\*ab"

    # p = "ba\*ab"

    # p = "a\*ab"

    # p = "a\*\*\*\*\*ab"

    # p = "\*a\*\*\*\*\*ab"

    # p = "ba\*ab\*\*\*\*"

    # p = "\*\*\*\*"

    # p = "\*"

    # p = "aa?ab"

    # p = "b\*b"

    # p = "a\*a"

    # p = "baaabab"

    # p = "?baaabab"

    # p = "\*baaaba\*"

**if** isMatch(s, p):

**print**("Yes")

**else**:

**print**("No")

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    main()

# This code is contributed by divyansh2212

**Output**

Yes

***Time complexity****: O(m x n).*

***Auxiliary space:****O(m x n).*

**Further Scope:**We can improve space complexity by making use of the fact that we only uses the result from last row.

// C++ program to implement wildcard

// pattern matching algorithm

#include <bits/stdc++.h>

**using namespace** std;

// Function that matches input str with

// given wildcard pattern

**bool** strmatch(**char** str[], **char** pattern[], **int** m, **int** n)

{

    // lookup table for storing results of

    // subproblems

    vector<**bool**> prev(m + 1, **false**), curr(m + 1, **false**);

    // empty pattern can match with empty string

    prev[0] = **true**;

    // fill the table in bottom-up fashion

**for** (**int** i = 1; i <= n; i++) {

**bool** flag = **true**;

**for** (**int** ii = 1; ii < i; ii++) {

**if** (pattern[ii - 1] != '\*') {

                flag = **false**;

**break**;

            }

        }

        curr[0] = flag; // for every row we are assigning

                        // 0th column value.

**for** (**int** j = 1; j <= m; j++) {

            // Two cases if we see a '\*'

            // a) We ignore ‘\*’ character and move

            //    to next  character in the pattern,

            //     i.e., ‘\*’ indicates an empty sequence.

            // b) '\*' character matches with ith

            //     character in input

**if** (pattern[i - 1] == '\*')

                curr[j] = curr[j - 1] || prev[j];

            // Current characters are considered as

            // matching in two cases

            // (a) current character of pattern is '?'

            // (b) characters actually match

**else if** (pattern[i - 1] == '?'

                     || str[j - 1] == pattern[i - 1])

                curr[j] = prev[j - 1];

            // If characters don't match

**else**

                curr[j] = **false**;

        }

        prev = curr;

    }

**return** prev[m];

}

**int** main()

{

**char** str[] = "baaabab";

**char** pattern[] = "\*\*\*\*\*ba\*\*\*\*\*ab";

    // char pattern[] = "ba\*\*\*\*\*ab";

    // char pattern[] = "ba\*ab";

    // char pattern[] = "a\*ab";

    // char pattern[] = "a\*\*\*\*\*ab";

    // char pattern[] = "\*a\*\*\*\*\*ab";

    // char pattern[] = "ba\*ab\*\*\*\*";

    // char pattern[] = "\*\*\*\*";

    // char pattern[] = "\*";

    // char pattern[] = "aa?ab";

    // char pattern[] = "b\*b";

    // char pattern[] = "a\*a";

    // char pattern[] = "baaabab";

    // char pattern[] = "?baaabab";

    // char pattern[] = "\*baaaba\*";

**if** (strmatch(str, pattern, **strlen**(str),

**strlen**(pattern)))

        cout << "Yes" << endl;

**else**

        cout << "No" << endl;

**return** 0;

}

**Output**

Yes

**Time complexity:** O(m x n).

**Auxiliary space:**  O(m).

One more improvement is you can merge consecutive ‘\*’ in the pattern to single ‘\*’ as they mean the same thing. For example for pattern “\*\*\*\*\*ba\*\*\*\*\*ab”, if we merge consecutive stars, the resultant string will be “\*ba\*ab”. So, value of m is reduced from 14 to 6.

**Search a Word in a 2D Grid of characters**

Given a 2D grid of characters and a single word/an array of words, find all occurrences of the given word/words in the grid. A word can be matched in all 8 directions at any point. Word is said to be found in a direction if all characters match in this direction (not in zig-zag form).

The 8 directions are, Horizontally Left, Horizontally Right, Vertically Up, Vertically Down and 4 Diagonal directions.

**Example:**

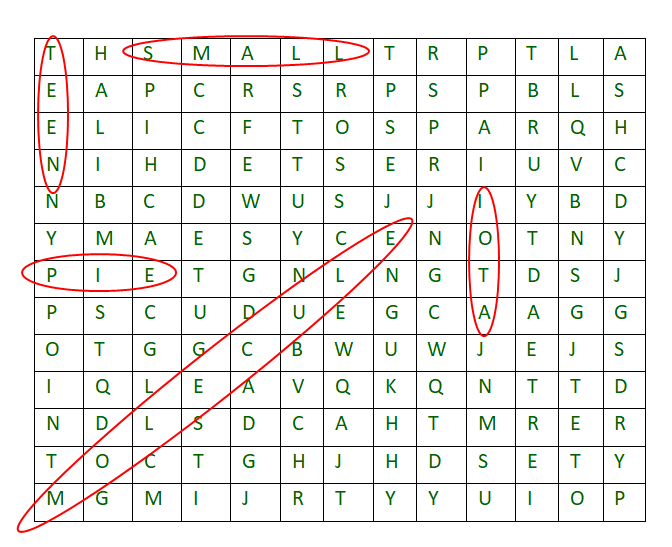
**Input:** grid[][] = {"GEEKSFORGEEKS",  
 "GEEKSQUIZGEEK",  
 "IDEQAPRACTICE"};  
 word = "GEEKS"

**Output:** pattern found at 0, 0  
 pattern found at 0, 8  
 pattern found at 1, 0  
**Explanation:** 'GEEKS' can be found as prefix of  
1st 2 rows and suffix of first row

**Input:** grid[][] = {"GEEKSFORGEEKS",  
 "GEEKSQUIZGEEK",  
 "IDEQAPRACTICE"};  
 word = "EEE"

**Output:** pattern found at 0, 2  
 pattern found at 0, 10  
 pattern found at 2, 2  
 pattern found at 2, 12  
**Explanation:** EEE can be found in first row   
twice at index 2 and index 10  
and in second row at 2 and 12

Below diagram shows a bigger grid and presence of different words in it.



**Source:** Microsoft Interview Question.

Recommended Problem

Find the string in grid

**Approach when a single word is given:** The idea used here is simple, we check every cell. If cell has first character, then we one by one try all 8 directions from that cell for a match. Implementation is interesting though. We use two arrays x[] and y[] to find next move in all 8 directions.

Below are implementation of the same:

1. C++
2. Java
3. Python3
4. C#
5. Javascript

# Python3 program to search a word in a 2D grid

**class** GFG:

**def** \_\_init\_\_(self):

        self.R **=** None

        self.C **=** None

        self.dir **=** [[**-**1, 0], [1, 0], [1, 1],

                    [1, **-**1], [**-**1, **-**1], [**-**1, 1],

                    [0, 1], [0, **-**1]]

    # This function searches in all 8-direction

    # from point(row, col) in grid[][]

**def** search2D(self, grid, row, col, word):

        # If first character of word doesn't match

        # with the given starting point in grid.

**if** grid[row][col] !**=** word[0]:

**return** False

        # Search word in all 8 directions

        # starting from (row, col)

**for** x, y **in** self.dir:

            # Initialize starting point

            # for current direction

            rd, cd **=** row **+** x, col **+** y

            flag **=** True

            # First character is already checked,

            # match remaining characters

**for** k **in** range(1, len(word)):

                # If out of bound or not matched, break

**if** (0 <**=** rd <self.R **and**

                    0 <**=** cd < self.C **and**

                    word[k] **==** grid[rd][cd]):

                    # Moving in particular direction

                    rd **+=** x

                    cd **+=** y

**else**:

                    flag **=** False

**break**

            # If all character matched, then

            # value of flag must be false

**if** flag:

**return** True

**return** False

    # Searches given word in a given matrix

    # in all 8 directions

**def** patternSearch(self, grid, word):

        # Rows and columns in given grid

        self.R **=** len(grid)

        self.C **=** len(grid[0])

        # Consider every point as starting point

        # and search given word

**for** row **in** range(self.R):

**for** col **in** range(self.C):

**if** self.search2D(grid, row, col, word):

                    print("pattern found at " **+**

                           str(row) **+** ', ' **+** str(col))

# Driver Code

**if** \_\_name\_\_**==**'\_\_main\_\_':

    grid **=** ["GEEKSFORGEEKS",

            "GEEKSQUIZGEEK",

            "IDEQAPRACTICE"]

    gfg **=** GFG()

    gfg.patternSearch(grid, 'GEEKS')

    print('')

    gfg.patternSearch(grid, 'EEE')

# This code is contributed by Yezheng Li

**Output**

pattern found at 0, 0  
pattern found at 0, 8  
pattern found at 1, 0

pattern found at 0, 2  
pattern found at 0, 10  
pattern found at 2, 2  
pattern found at 2, 12

**Complexity Analysis:**

1. **Time complexity:** O(R\*C\*8\*len(str)).   
   All the cells will be visited and traversed in all 8 directions, where R and C is side of matrix so time complexity is O(R\*C).
2. **Auxiliary Space:** O(1).   
   As no extra space is needed.

**String matching where one string contains wildcard characters**

Given two strings where first string may contain wild card characters and second string is a normal string. Write a function that returns true if the two strings match. The following are allowed wild card characters in first string.

\* --> Matches with 0 or more instances of any character or set of characters.  
? --> Matches with any one character.

For example, “g\*ks” matches with “geeks” match. And string “ge?ks\*” matches with “geeksforgeeks” (note ‘\*’ at the end of first string). But “g\*k” doesn’t match with “gee” as character ‘k’ is not present in second string.

Recommended Problem

Wildcard string matching

# Python program to match wild card characters

# The main function that checks if two given strings match.

# The first string may contain wildcard characters

**def** match(first, second):

    # If we reach at the end of both strings, we are done

**if** len(first) **==** 0 **and** len(second) **==** 0:

**return** True

    # Make sure to eliminate consecutive '\*'

**if** len(first) > 1 **and** first[0] **==** '\*':

        i **=** 0

**while** i**+**1 < len(first) **and** first[i**+**1] **==** '\*':

            i **=** i**+**1

        first **=** first[i:]

    # Make sure that the characters after '\*' are present

    # in second string. This function assumes that the first

    # string will not contain two consecutive '\*'

**if** len(first) > 1 **and** first[0] **==** '\*' **and** len(second) **==** 0:

**return** False

    # If the first string contains '?', or current characters

    # of both strings match

**if** (len(first) > 1 **and** first[0] **==** '?') **or** (len(first) !**=** 0

**and** len(second) !**=** 0 **and** first[0] **==** second[0]):

**return** match(first[1:], second[1:])

    # If there is \*, then there are two possibilities

    # a) We consider current character of second string

    # b) We ignore current character of second string.

**if** len(first) !**=** 0 **and** first[0] **==** '\*':

**return** match(first[1:], second) **or** match(first, second[1:])

**return** False

# A function to run test cases

**def** test(first, second):

**if** match(first, second):

        print("Yes")

**else**:

**print**("No")

# Driver program

test("g\*ks", "geeks")  # Yes

test("ge?ks\*", "geeksforgeeks")  # Yes

test("g\*k", "gee")  # No because 'k' is not in second

test("\*pqrs", "pqrst")  # No because 't' is not in first

test("abc\*bcd", "abcdhghgbcd")  # Yes

test("abc\*c?d", "abcd")  # No because second must have 2 instances of 'c'

test("\*c\*d", "abcd")  # Yes

test("\*?c\*d", "abcd")  # Yes

test("geeks\*\*", "geeks")  # Yes

# This code is contributed by BHAVYA JAIN and ROHIT SIKKA

**Output:**

Yes  
Yes  
No  
No  
Yes  
No  
Yes  
Yes  
Yes

**Time Complexity:** O(n)

**Auxiliary Space:** O(1)

**Exercise**

**1)** In the above solution, all non-wild characters of first string must be there is second string and all characters of second string must match with either a normal character or wildcard character of first string. Extend the above solution to work like other [pattern searching solutions](https://www.geeksforgeeks.org/tag/pattern-searching/) where the first string is pattern and second string is text and we should print all occurrences of first string in second.

**2)**Write a pattern searching function where the meaning of ‘?’ is same, but ‘\*’ means 0 or more occurrences of the character just before ‘\*’. For example, if first string is ‘a\*b’, then it matches with ‘aaab’, but doesn’t match with ‘abb’.

**Suffix Tree Application 1 – Substring Check**

Given a text string and a pattern string, check if a pattern exists in text or not.

Few pattern searching algorithms ([KMP](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/), [Rabin-Karp](https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/), [Naive Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/), [Finite Automata](https://www.geeksforgeeks.org/pattern-searching-set-5-efficient-constructtion-of-finite-automata/)) are already discussed, which can be used for this check.

Here we will discuss suffix tree based algorithm.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.

Suffix Tree Application

Here we will build suffix tree using Ukkonen’s Algorithm, discussed already as below:

[Ukkonen’s Suffix Tree Construction – Part 1](https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/)

[Ukkonen’s Suffix Tree Construction – Part 2](https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-2/)

[Ukkonen’s Suffix Tree Construction – Part 3](https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-3/)

[Ukkonen’s Suffix Tree Construction – Part 4](https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-4/)

[Ukkonen’s Suffix Tree Construction – Part 5](https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-5/)

[Ukkonen’s Suffix Tree Construction – Part 6](https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-6/)

The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms.

1. C

// A C program for substring check using Ukkonen's Suffix Tree Construction

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_CHAR 256

**struct** SuffixTreeNode {

**struct** SuffixTreeNode \*children[MAX\_CHAR];

    //pointer to other node via suffix link

**struct** SuffixTreeNode \*suffixLink;

    /\*(start, end) interval specifies the edge, by which the

     node is connected to its parent node. Each edge will

     connect two nodes,  one parent and one child, and

     (start, end) interval of a given edge  will be stored

     in the child node. Let's say there are two nods A and B

     connected by an edge with indices (5, 8) then this

     indices (5, 8) will be stored in node B. \*/

**int** start;

**int** \*end;

    /\*for leaf nodes, it stores the index of suffix for

      the path  from root to leaf\*/

**int** suffixIndex;

};

**typedef struct** SuffixTreeNode Node;

**char** text[100]; //Input string

Node \*root = NULL; //Pointer to root node

/\*lastNewNode will point to newly created internal node,

  waiting for it's suffix link to be set, which might get

  a new suffix link (other than root) in next extension of

  same phase. lastNewNode will be set to NULL when last

  newly created internal node (if there is any) got it's

  suffix link reset to new internal node created in next

  extension of same phase. \*/

Node \*lastNewNode = NULL;

Node \*activeNode = NULL;

/\*activeEdge is represented as an input string character

  index (not the character itself)\*/

**int** activeEdge = -1;

**int** activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to

// be added in tree

**int** remainingSuffixCount = 0;

**int** leafEnd = -1;

**int** \*rootEnd = NULL;

**int** \*splitEnd = NULL;

**int** size = -1; //Length of input string

Node \*newNode(**int** start, **int** \*end)

{

    Node \*node =(Node\*) **malloc**(**sizeof**(Node));

**int** i;

**for** (i = 0; i < MAX\_CHAR; i++)

          node->children[i] = NULL;

    /\*For root node, suffixLink will be set to NULL

    For internal nodes, suffixLink will be set to root

    by default in  current extension and may change in

    next extension\*/

    node->suffixLink = root;

    node->start = start;

    node->end = end;

    /\*suffixIndex will be set to -1 by default and

      actual suffix index will be set later for leaves

      at the end of all phases\*/

    node->suffixIndex = -1;

**return** node;

}

**int** edgeLength(Node \*n) {

**if**(n == root)

**return** 0;

**return** \*(n->end) - (n->start) + 1;

}

**int** walkDown(Node \*currNode)

{

    /\*activePoint change for walk down (APCFWD) using

     Skip/Count Trick  (Trick 1). If activeLength is greater

     than current edge length, set next  internal node as

     activeNode and adjust activeEdge and activeLength

     accordingly to represent same activePoint\*/

**if** (activeLength >= edgeLength(currNode))

    {

        activeEdge += edgeLength(currNode);

        activeLength -= edgeLength(currNode);

        activeNode = currNode;

**return** 1;

    }

**return** 0;

}

**void** extendSuffixTree(**int** pos)

{

    /\*Extension Rule 1, this takes care of extending all

    leaves created so far in tree\*/

    leafEnd = pos;

    /\*Increment remainingSuffixCount indicating that a

    new suffix added to the list of suffixes yet to be

    added in tree\*/

    remainingSuffixCount++;

    /\*set lastNewNode to NULL while starting a new phase,

     indicating there is no internal node waiting for

     it's suffix link reset in current phase\*/

    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree

**while**(remainingSuffixCount > 0) {

**if** (activeLength == 0)

            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with

        // activeEdge from activeNode

**if** (activeNode->children] == NULL)

        {

            //Extension Rule 2 (A new leaf edge gets created)

            activeNode->children] =

                                          newNode(pos, &leafEnd);

            /\*A new leaf edge is created in above line starting

             from  an existing node (the current activeNode), and

             if there is any internal node waiting for it's suffix

             link get reset, point the suffix link from that last

             internal node to current activeNode. Then set lastNewNode

             to NULL indicating no more node waiting for suffix link

             reset.\*/

**if** (lastNewNode != NULL)

            {

                lastNewNode->suffixLink = activeNode;

                lastNewNode = NULL;

            }

        }

        // There is an outgoing edge starting with activeEdge

        // from activeNode

**else**

        {

            // Get the next node at the end of edge starting

            // with activeEdge

            Node \*next = activeNode->children];

**if** (walkDown(next))//Do walkdown

            {

                //Start from next node (the new activeNode)

**continue**;

            }

            /\*Extension Rule 3 (current character being processed

              is already on the edge)\*/

**if** (text[next->start + activeLength] == text[pos])

            {

                //If a newly created node waiting for it's

                //suffix link to be set, then set suffix link

                //of that waiting node to current active node

**if**(lastNewNode != NULL && activeNode != root)

                {

                    lastNewNode->suffixLink = activeNode;

                    lastNewNode = NULL;

                }

                //APCFER3

                activeLength++;

                /\*STOP all further processing in this phase

                and move on to next phase\*/

**break**;

            }

            /\*We will be here when activePoint is in the middle of

              the edge being traversed and current character

              being processed is not  on the edge (we fall off

              the tree). In this case, we add a new internal node

              and a new leaf edge going out of that new node. This

              is Extension Rule 2, where a new leaf edge and a new

            internal node get created\*/

            splitEnd = (**int**\*) **malloc**(**sizeof**(**int**));

            \*splitEnd = next->start + activeLength - 1;

            //New internal node

            Node \*split = newNode(next->start, splitEnd);

            activeNode->children] = split;

            //New leaf coming out of new internal node

            split->children] = newNode(pos, &leafEnd);

            next->start += activeLength;

            split->children] = next;

            /\*We got a new internal node here. If there is any

              internal node created in last extensions of same

              phase which is still waiting for it's suffix link

              reset, do it now.\*/

**if** (lastNewNode != NULL)

            {

            /\*suffixLink of lastNewNode points to current newly

              created internal node\*/

                lastNewNode->suffixLink = split;

            }

            /\*Make the current newly created internal node waiting

              for it's suffix link reset (which is pointing to root

              at present). If we come across any other internal node

              (existing or newly created) in next extension of same

              phase, when a new leaf edge gets added (i.e. when

              Extension Rule 2 applies is any of the next extension

              of same phase) at that point, suffixLink of this node

              will point to that internal node.\*/

            lastNewNode = split;

        }

        /\* One suffix got added in tree, decrement the count of

          suffixes yet to be added.\*/

        remainingSuffixCount--;

**if** (activeNode == root && activeLength > 0) //APCFER2C1

        {

            activeLength--;

            activeEdge = pos - remainingSuffixCount + 1;

        }

**else if** (activeNode != root) //APCFER2C2

        {

            activeNode = activeNode->suffixLink;

        }

    }

}

**void** print(**int** i, **int** j)

{

**int** k;

**for** (k=i; k<=j; k++)

**printf**("%c", text[k]);

}

//Print the suffix tree as well along with setting suffix index

//So tree will be printed in DFS manner

//Each edge along with it's suffix index will be printed

**void** setSuffixIndexByDFS(Node \*n, **int** labelHeight)

{

**if** (n == NULL)  **return**;

**if** (n->start != -1) //A non-root node

    {

        //Print the label on edge from parent to current node

        //Uncomment below line to print suffix tree

       // print(n->start, \*(n->end));

    }

**int** leaf = 1;

**int** i;

**for** (i = 0; i < MAX\_CHAR; i++)

    {

**if** (n->children[i] != NULL)

        {

            //Uncomment below two lines to print suffix index

           // if (leaf == 1 && n->start != -1)

             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing

            //edges from it.

            leaf = 0;

            setSuffixIndexByDFS(n->children[i], labelHeight +

                                  edgeLength(n->children[i]));

        }

    }

**if** (leaf == 1)

    {

        n->suffixIndex = size - labelHeight;

        //Uncomment below line to print suffix index

        //printf(" [%d]\n", n->suffixIndex);

    }

}

**void** freeSuffixTreeByPostOrder(Node \*n)

{

**if** (n == NULL)

**return**;

**int** i;

**for** (i = 0; i < MAX\_CHAR; i++)

    {

**if** (n->children[i] != NULL)

        {

            freeSuffixTreeByPostOrder(n->children[i]);

        }

    }

**if** (n->suffixIndex == -1)

**free**(n->end);

**free**(n);

}

/\*Build the suffix tree and print the edge labels along with

suffixIndex. suffixIndex for leaf edges will be >= 0 and

for non-leaf edges will be -1\*/

**void** buildSuffixTree()

{

    size = **strlen**(text);

**int** i;

    rootEnd = (**int**\*) **malloc**(**sizeof**(**int**));

    \*rootEnd = - 1;

    /\*Root is a special node with start and end indices as -1,

    as it has no parent from where an edge comes to root\*/

    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root

**for** (i=0; i<size; i++)

        extendSuffixTree(i);

**int** labelHeight = 0;

    setSuffixIndexByDFS(root, labelHeight);

}

**int** traverseEdge(**char** \*str, **int** idx, **int** start, **int** end)

{

**int** k = 0;

    //Traverse the edge with character by character matching

**for**(k=start; k<=end && str[idx] != '\0'; k++, idx++)

    {

**if**(text[k] != str[idx])

**return** -1;  // mo match

    }

**if**(str[idx] == '\0')

**return** 1;  // match

**return** 0;  // more characters yet to match

}

**int** doTraversal(Node \*n, **char**\* str, **int** idx)

{

**if**(n == NULL)

    {

**return** -1; // no match

    }

**int** res = -1;

    //If node n is not root node, then traverse edge

    //from node n's parent to node n.

**if**(n->start != -1)

    {

        res = traverseEdge(str, idx, n->start, \*(n->end));

**if**(res != 0)

**return** res;  // match (res = 1) or no match (res = -1)

    }

    //Get the character index to search

    idx = idx + edgeLength(n);

    //If there is an edge from node n going out

    //with current character str[idx], traverse that edge

**if**(n->children[str[idx]] != NULL)

**return** doTraversal(n->children[str[idx]], str, idx);

**else**

**return** -1;  // no match

}

**void** checkForSubString(**char**\* str)

{

**int** res = doTraversal(root, str, 0);

**if**(res == 1)

**printf**("Pattern <%s> is a Substring\n", str);

**else**

**printf**("Pattern <%s> is NOT a Substring\n", str);

}

// driver program to test above functions

**int** main(**int** argc, **char** \*argv[])

{

**strcpy**(text, "THIS IS A TEST TEXT$");

    buildSuffixTree();

    checkForSubString("TEST");

    checkForSubString("A");

    checkForSubString(" ");

    checkForSubString("IS A");

    checkForSubString(" IS A ");

    checkForSubString("TEST1");

    checkForSubString("THIS IS GOOD");

    checkForSubString("TES");

    checkForSubString("TESA");

    checkForSubString("ISB");

    //Free the dynamically allocated memory

    freeSuffixTreeByPostOrder(root);

**return** 0;

}

Output:

Pattern <TEST> is a Substring  
Pattern <A> is a Substring  
Pattern < > is a Substring  
Pattern <IS A> is a Substring  
Pattern < IS A > is a Substring  
Pattern <TEST1> is NOT a Substring  
Pattern <THIS IS GOOD> is NOT a Substring  
Pattern <TES> is a Substring  
Pattern <TESA> is NOT a Substring  
Pattern <ISB> is NOT a Substring

Ukkonen’s Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M.

With a slight modification in the traversal algorithm discussed here, we can answer the following:

1. Find all occurrences of a given pattern P present in text T.
2. How to check if a pattern is prefix of a text?
3. How to check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

* [Suffix Tree Application 2 – Searching All Patterns](https://www.geeksforgeeks.org/suffix-tree-application-2-searching-all-patterns/)
* [Suffix Tree Application 3 – Longest Repeated Substring](https://www.geeksforgeeks.org/suffix-tree-application-3-longest-repeated-substring/)
* [Suffix Tree Application 4 – Build Linear Time Suffix Array](https://www.geeksforgeeks.org/suffix-tree-application-4-build-linear-time-suffix-array/)
* [Generalized Suffix Tree 1](https://www.geeksforgeeks.org/generalized-suffix-tree-1/)
* [Suffix Tree Application 5 – Longest Common Substring](https://www.geeksforgeeks.org/suffix-tree-application-5-longest-common-substring-2/)
* [Suffix Tree Application 6 – Longest Palindromic Substring](https://www.geeksforgeeks.org/suffix-tree-application-6-longest-palindromic-substring/)

**Medium Questions:**

**Hard Questions:**