

# Contents

<b>1 Find Nth term of the series 1, 8, 54, 384...</b>	<b>6</b>
Source . . . . .	9
<b>2 Find Nth term of the series 0, 2, 4, 8, 12, 18...</b>	<b>10</b>
Source . . . . .	12
<b>3 Number of substrings of one string present in other</b>	<b>13</b>
Source . . . . .	17
<b>4 Find Nth term of series 1, 4, 15, 72, 420...</b>	<b>18</b>
Source . . . . .	22
<b>5 Count Occurences of Anagrams</b>	<b>23</b>
Source . . . . .	26
<b>6 Minimum characters to be replaced to remove the given substring</b>	<b>27</b>
Source . . . . .	30
<b>7 Custom Building Cryptography Algorithms (Hybrid Cryptography)</b>	<b>31</b>
Source . . . . .	36
<b>8 Program to find all match of a regex in a string</b>	<b>37</b>
Source . . . . .	39
<b>9 KLA Tencor Interview Experience   Set 3</b>	<b>40</b>
Source . . . . .	43
<b>10 Check if a string can be formed from another string using given constraints</b>	<b>44</b>
Source . . . . .	46
<b>11 Largest connected component on a grid</b>	<b>47</b>
Source . . . . .	57
<b>12 Make array elements equal in Minimum Steps</b>	<b>58</b>
Source . . . . .	63
<b>13 PHP   ereg_replace() Function</b>	<b>64</b>
Source . . . . .	66

<b>14 Check if strings are rotations of each other or not   Set 2</b>	<b>67</b>
Source . . . . .	71
<b>15 DFA for Strings not ending with “THE”</b>	<b>72</b>
Source . . . . .	78
<b>16 Check if a string is substring of another</b>	<b>79</b>
Source . . . . .	84
<b>17 Program to replace a word with asterisks in a sentence</b>	<b>85</b>
Source . . . . .	86
<b>18 Dynamic Programming   Wildcard Pattern Matching   Linear Time and Constant Space</b>	<b>87</b>
Source . . . . .	93
<b>19 Pattern Searching using C++ library</b>	<b>94</b>
Source . . . . .	95
<b>20 Longest prefix which is also suffix</b>	<b>96</b>
Source . . . . .	108
<b>21 Splitting a Numeric String</b>	<b>109</b>
Source . . . . .	115
<b>22 Count of number of given string in 2D character array</b>	<b>116</b>
Source . . . . .	119
<b>23 Find minimum shift for longest common prefix</b>	<b>120</b>
Source . . . . .	122
<b>24 Frequency of a substring in a string</b>	<b>123</b>
Source . . . . .	127
<b>25 Count of occurrences of a “1(0+)1” pattern in a string</b>	<b>128</b>
Source . . . . .	132
<b>26 Find all the patterns of “1(0+)1” in a given string   SET 2(Regular Expression Approach)</b>	<b>133</b>
Source . . . . .	135
<b>27 Find all the patterns of “1(0+)1” in a given string   SET 1(General Approach)</b>	<b>136</b>
Source . . . . .	142
<b>28 Boyer Moore Algorithm   Good Suffix heuristic</b>	<b>143</b>
Source . . . . .	152
<b>29 is_permutation() in C++ and its application for anagram search</b>	<b>153</b>
Source . . . . .	155

<b>30 Match Expression where a single special character in pattern can match one or more characters</b>	<b>156</b>
Source . . . . .	163
<b>31 Maximum length prefix of one string that occurs as subsequence in another</b>	<b>164</b>
Source . . . . .	169
<b>32 Replace all occurrences of string AB with C without using extra space</b>	<b>170</b>
Source . . . . .	179
<b>33 Wildcard Pattern Matching</b>	<b>180</b>
Source . . . . .	187
<b>34 Find all occurrences of a given word in a matrix</b>	<b>188</b>
Source . . . . .	192
<b>35 Aho-Corasick Algorithm for Pattern Searching</b>	<b>193</b>
Source . . . . .	200
<b>36 kasai's Algorithm for Construction of LCP array from Suffix Array</b>	<b>201</b>
Source . . . . .	207
<b>37 Search a Word in a 2D Grid of characters</b>	<b>208</b>
Source . . . . .	211
<b>38 Z algorithm (Linear time pattern searching Algorithm)</b>	<b>212</b>
Source . . . . .	218
<b>39 Online algorithm for checking palindrome in a stream</b>	<b>219</b>
Source . . . . .	227
<b>40 Suffix Tree Application 6 – Longest Palindromic Substring</b>	<b>228</b>
Source . . . . .	242
<b>41 Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 4</b>	<b>243</b>
Source . . . . .	249
<b>42 Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 3</b>	<b>250</b>
Source . . . . .	256
<b>43 Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 2</b>	<b>257</b>
Source . . . . .	262
<b>44 Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 1</b>	<b>263</b>
Source . . . . .	265

<b>45 Suffix Tree Application 5 – Longest Common Substring</b>	<b>266</b>
Source . . . . .	278
<b>46 Generalized Suffix Tree 1</b>	<b>279</b>
Source . . . . .	290
<b>47 Suffix Tree Application 4 – Build Linear Time Suffix Array</b>	<b>291</b>
Source . . . . .	302
<b>48 Suffix Tree Application 3 – Longest Repeated Substring</b>	<b>303</b>
Source . . . . .	314
<b>49 Suffix Tree Application 2 – Searching All Patterns</b>	<b>315</b>
Source . . . . .	329
<b>50 Suffix Tree Application 1 – Substring Check</b>	<b>330</b>
Source . . . . .	340
<b>51 Ukkonen’s Suffix Tree Construction – Part 6</b>	<b>341</b>
Source . . . . .	351
<b>52 Ukkonen’s Suffix Tree Construction – Part 5</b>	<b>352</b>
Source . . . . .	364
<b>53 Ukkonen’s Suffix Tree Construction – Part 4</b>	<b>365</b>
Source . . . . .	372
<b>54 Ukkonen’s Suffix Tree Construction – Part 3</b>	<b>373</b>
Source . . . . .	382
<b>55 Ukkonen’s Suffix Tree Construction – Part 2</b>	<b>383</b>
Source . . . . .	389
<b>56 Ukkonen’s Suffix Tree Construction – Part 1</b>	<b>390</b>
Source . . . . .	402
<b>57 Pattern Searching using a Trie of all Suffixes</b>	<b>403</b>
Source . . . . .	411
<b>58 Anagram Substring Search (Or Search for all permutations)</b>	<b>412</b>
Source . . . . .	417
<b>59 Suffix Array   Set 2 (nLogn Algorithm)</b>	<b>418</b>
Source . . . . .	423
<b>60 Suffix Array   Set 1 (Introduction)</b>	<b>424</b>
Source . . . . .	428
<b>61 String matching where one string contains wildcard characters</b>	<b>429</b>
Source . . . . .	432

<b>62 Pattern Searching using Suffix Tree</b>	<b>433</b>
Source . . . . .	437
<b>63 Boyer Moore Algorithm for Pattern Searching</b>	<b>438</b>
Source . . . . .	444
<b>64 Pattern Searching   Set 6 (Efficient Construction of Finite Automata)</b>	<b>445</b>
Source . . . . .	448
<b>65 Finite Automata algorithm for Pattern Searching</b>	<b>449</b>
Source . . . . .	456
<b>66 Optimized Naive Algorithm for Pattern Searching</b>	<b>457</b>
Source . . . . .	460
<b>67 Rabin-Karp Algorithm for Pattern Searching</b>	<b>461</b>
Source . . . . .	469
<b>68 KMP Algorithm for Pattern Searching</b>	<b>470</b>
Source . . . . .	483
<b>69 Naive algorithm for Pattern Searching</b>	<b>484</b>
Source . . . . .	488

## Chapter 1

# Find Nth term of the series 1, 8, 54, 384...

Find Nth term of the series 1, 8, 54, 384... - GeeksforGeeks

Given a number N. The task is to write a program to find the Nth term in the below series:

1, 8, 54, 384...

**Examples:**

```
Input : 3
Output : 54
For N = 3
Nth term = ( 3*3 ) * 3!
           = 54
```

```
Input : 2
Output : 8
```

On observing carefully, the Nth term in the above series can be generalized as:

$$\text{Nth term} = (N * N) * (N!)$$

Below is the implementation of the above approach:

**C++**

```
// CPP program to find N-th term of the series:
// 1, 8, 54, 384...
```

```
#include <iostream>
using namespace std;

// calculate factorial of N
int fact(int N)
{
    int i, product = 1;
    for (i = 1; i <= N; i++)
        product = product * i;
    return product;
}

// calculate Nth term of series
int nthTerm(int N)
{
    return (N * N) * fact(N);
}

// Driver Function
int main()
{
    int N = 4;

    cout << nthTerm(N);

    return 0;
}
```

## Java

```
// Java program to find N-th term of the series:
// 1, 8, 54, 384...

import java.io.*;

// Main class for main method
class GFG {
    public static int fact(int N)
    {
        int i, product = 1;
        // Calculate factorial of N
        for (i = 1; i <= N; i++)
            product = product * i;
        return product;
    }
    public static int nthTerm(int N)
    {
        // By using above formula
```

```
        return (N * N) * fact(N);
    }

    public static void main(String[] args)
    {
        int N = 4; // 4th term is 384

        System.out.println(nthTerm(N));
    }
}
```

### Python 3

```
# Python 3 program to find
# N-th term of the series:
# 1, 8, 54, 384...

# calculate factorial of N
def fact(N):

    product = 1
    for i in range(1, N + 1):
        product = product * i
    return product

# calculate Nth term of series
def nthTerm(N):
    return (N * N) * fact(N)

# Driver Code
if __name__ == "__main__":
    N = 4
    print(nthTerm(N))

# This code is contributed
# by ChitraNayal
```

### C#

```
// C# program to find N-th
// term of the series:
// 1, 8, 54, 384...
using System;

class GFG
{
    public static int fact(int N)
    {
        int i, product = 1;

        // Calculate factorial of N
```



```
        for (i = 1; i <= N; i++)
            product = product * i;
        return product;
    }

    public static int nthTerm(int N)
    {
        // By using above formula
        return (N * N) * fact(N);
    }

    // Driver Code
    public static void Main(String[] args)
    {
        int N = 4; // 4th term is 384

        Console.WriteLine(nthTerm(N));
    }
}
```

// This code is contributed  
// by Kirti\_Mangal

## PHP

### Output:

384

**Time Complexity:**  $O(N)$

**Improved By :** [Kirti\\_Mangal](#), [ChitraNayal](#)

### Source

<https://www.geeksforgeeks.org/find-nth-term-of-the-series-1-8-54-384/>

## Chapter 2

# Find Nth term of the series 0, 2, 4, 8, 12, 18...

Find Nth term of the series 0, 2, 4, 8, 12, 18... - GeeksforGeeks

Given a number N. The task is to write a program to find the Nth term in the below series:

0, 2, 4, 8, 12, 18...

**Examples:**

```
Input: 3
Output: 4
For N = 3
Nth term = ( 3 + ( 3 - 1 ) * 3 ) / 2
          = 4
Input: 5
Output: 12
```

On observing carefully, the  $N^{\text{th}}$  term in the above series can be generalized as:

$$\text{Nth term} = ( N + ( N - 1 ) * N ) / 2$$

Below is the implementation of the above approach:

**C++**

```
// CPP program to find N-th term of the series:
// 0, 2, 4, 8, 12, 18...
#include <iostream>
```

```
using namespace std;

// calculate Nth term of series
int nthTerm(int N)
{
    return (N + N * (N - 1)) / 2;
}

// Driver Function
int main()
{
    int N = 5;

    cout << nthTerm(N);

    return 0;
}
```

### Java

```
// Java program to find N-th term of the series:
// 0, 2, 4, 8, 12, 18...

import java.io.*;

// Main class for main method
class GFG {
    public static int nthTerm(int N)
    {
        // By using above formula
        return (N + (N - 1) * N) / 2;
    }

    public static void main(String[] args)
    {
        int N = 5; // 5th term is 12

        System.out.println(nthTerm(N));
    }
}
```

### Python 3

```
# Python 3 program to find N-th term of the series:
# 0, 2, 4, 8, 12, 18.

# calculate Nth term of series
```

```
def nthTerm(N) :  
  
    return (N + N * (N - 1)) // 2  
  
# Driver Code  
if __name__ == "__main__" :  
  
    N = 5  
  
    print(nthTerm(N))  
  
# This code is contributed by ANKITRAI1
```

## PHP

```
<?php  
// PHP program to find  
// N-th term of the series:  
// 0, 2, 4, 8, 12, 18...  
  
// calculate Nth term of series  
function nthTerm($N)  
{  
    return (int)(( $\$N + \$N * (\$N - 1) / 2$ );  
}  
  
// Driver Code  
$N = 5;  
  
echo nthTerm($N);  
  
// This code is contributed by mits  
?>
```

## Output:

12

**Time Complexity:**  $O(1)$

**Improved By :** [Mithun Kumar](#), [ANKITRAI1](#)

## Source

<https://www.geeksforgeeks.org/find-nth-term-of-the-series-0-2-4-8-12-18/>

## Chapter 3

# Number of substrings of one string present in other

Number of substrings of one string present in other - GeeksforGeeks

Suppose we are given a string s1, we need to find total number of substring(including multiple occurrences of the same substring) of s1 which are present in string s2.

**Examples:**

```
Input : s1 = aab
        s2 = aaaab
Output : 6
Substrings of s1 are ["a", "a", "b", "aa",
"ab", "aab"]. These all are present in s2.
Hence, answer is 6.
```

```
Input :s1 = abcd
        s2 = swalencud
Output :3
```

The idea is to consider all substrings of s1 and check if it present in s2.

C++

```
// CPP program to count number of substrings of s1
// present in s2.
#include<iostream>
#include<string>
using namespace std;

int countSubstrs(string s1, string s2)
```

```
{
    int ans = 0;

    for (int i = 0; i < s1.length(); i++) {

        // s3 stores all substrings of s1
        string s3;
        for (int j = i; j < s1.length(); j++) {
            s3 += s1[j];

            // check the presence of s3 in s2
            if (s2.find(s3) != string::npos)
                ans++;
        }
    }
    return ans;
}

// Driver code
int main()
{
    string s1 = "aab", s2 = "aaaab";
    cout << countSubstrs(s1, s2);
    return 0;
}
```

#### Java

```
// Java program to count number of
// substrings of s1 present in s2.
import java.util.*;

class GFG
{
    static int countSubstrs(String s1,
                           String s2)
    {
        int ans = 0;

        for (int i = 0; i < s1.length(); i++)
        {

            // s3 stores all substrings of s1
            String s3 = "";
            char[] s4 = s1.toCharArray();
            for (int j = i; j < s1.length(); j++)
            {
```

```
        s3 += s4[j];

        // check the presence of s3 in s2
        if (s2.indexOf(s3) != -1)
            ans++;
    }
}
return ans;
}

// Driver code
public static void main(String[] args)
{
    String s1 = "aab", s2 = "aaaab";
    System.out.println(countSubstrs(s1, s2));
}
}

// This code is contributed by ChitraNayal
```

### Python 3

```
# Python 3 program to count number of substrings of s1
# present in s2.

# Function for counting no. of substring
# of s1 present in s2
def countSubstrs(s1, s2) :
    ans = 0
    for i in range(len(s1)) :
        s3 = ""

        # s3 stores all substrings of s1
        for j in range(i, len(s1)) :
            s3 += s1[j]

            # check the presence of s3 in s2
            if s2.find(s3) != -1 :
                ans += 1

    return ans

# Driver code
if __name__ == "__main__" :
    s1 = "aab"
    s2 = "aaaab"

    # function calling
    print(countSubstrs(s1, s2))
```

# This code is contributed by ANKITRAI1

**C#**

```
// C# program to count number of
// substrings of s1 present in s2.
using System;

class GFG
{
    static int countSubstrs(String s1,
    String s2)
    {
        int ans = 0;

        for (int i = 0; i < s1.Length; i++) { // s3 stores all substrings of s1 String s3 = ""; char[] s4
        = s1.ToCharArray(); for (int j = i; j < s1.Length; j++) { s3 += s4[j]; // check the presence
        of s3 in s2 if (s2.IndexOf(s3) != -1) ans++; } } return ans; } // Driver code public static void
        Main(String[] args) { String s1 = "aab", s2 = "aaaab"; Console.WriteLine(countSubstrs(s1,
        s2)); } } // This code is contributed // by Kirti_Mangal [tabby title="PHP"]
```

```
<?php
// PHP program to count number of
// substrings of s1 present in s2.

function countSubstrs($s1, $s2)
{
    $ans = 0;

    for ($i = 0; $i < strlen($s1); $i++)
    {

        // s3 stores all substrings of s1
        $s3 = "";
        for ($j = $i;
            $j < strlen($s1); $j++)
        {
            $s3 += $s1[$j];

            // check the presence of s3 in s2
            if (strpos($s2, $s3, 0) != -1)
                $ans++;
        }
    }
    return $ans;
}

// Driver code
```



```
$s1 = "aab";  
$s2 = "aaaab";  
echo countSubstrs($s1, $s2);  
  
// This code is contributed  
// by ChitraNayal  
?>
```

**Output:**

6

**Improved By :** [ANKITRAI1](#), [ChitraNayal](#), [Kirti\\_Mangal](#)

**Source**

<https://www.geeksforgeeks.org/number-of-substrings-of-one-string-present-in-other/>

## Chapter 4

# Find Nth term of series 1, 4, 15, 72, 420...

Find Nth term of series 1, 4, 15, 72, 420... - GeeksforGeeks

Given a number N. The task is to write a program to find the N<sup>th</sup> term in the below series:

1, 4, 15, 72, 420...

**Examples:**

Input: 3

Output: 15

For N = 3, we know that the factorial of 3 is 6

Nth term =  $6 * (3 + 2) / 2$   
= 15

Input: 6

Output: 2880

For N = 6, we know that the factorial of 6 is 720

Nth term =  $720 * (6 + 2) / 2$   
= 2880

The idea is to first find the factorial of the given number N, that is N!. Now the N-th term in the above series will be:

$$\text{N-th term} = N! * (N + 2) / 2$$

Below is the implementation of the above approach:

**C++**

```
// CPP program to find N-th term of the series:
// 1, 4, 15, 72, 420...
#include <iostream>
using namespace std;

// Function to find factorial of N
int factorial(int N)
{
    int fact = 1;

    for (int i = 1; i <= N; i++)
        fact = fact * i;

    // return factorial of N
    return fact;
}

// calculate Nth term of series
int nthTerm(int N)
{
    return (factorial(N) * (N + 2) / 2);
}

// Driver Function
int main()
{
    int N = 6;

    cout << nthTerm(N);

    return 0;
}
```

#### Java

```
// Java program to find N-th
// term of the series:
// 1, 4, 15, 72, 420
import java.util.*;
import java.lang.*;
import java.io.*;

class GFG
{
    // Function to find factorial of N
    static int factorial(int N)
    {
```

```
int fact = 1;

for (int i = 1; i <= N; i++)
    fact = fact * i;

// return factorial of N
return fact;
}

// calculate Nth term of series
static int nthTerm(int N)
{
    return (factorial(N) * (N + 2) / 2);
}

// Driver Code
public static void main(String args[])
{
    int N = 6;

    System.out.println(nthTerm(N));
}

// This code is contributed by Subhadeep
```

### Python3

```
# Python 3 program to find
# N-th term of the series:
# 1, 4, 15, 72, 420...

# Function for finding
# factorial of N
def factorial(N) :
    fact = 1
    for i in range(1, N + 1) :
        fact = fact * i

    # return factorial of N
    return fact

# Function for calculating
# Nth term of series
def nthTerm(N) :

    # return nth term
    return (factorial(N) * (N + 2) // 2)
```

```
# Driver code
if __name__ == "__main__" :
```

```
    N = 6
```

```
    # Function Calling
    print(nthTerm(N))
```

```
# This code is contributed
# by ANKITRAI1
```

**C#**

```
// C# program to find N-th
// term of the series:
// 1, 4, 15, 72, 420
using System;
```

```
class GFG
{
```

```
    // Function to find factorial of N
    static int factorial(int N)
```

```
    {
        int fact = 1;
```

```
        for (int i = 1; i <= N; i++) fact = fact * i; // return factorial of N return fact; } //
        calculate Nth term of series static int nthTerm(int N) { return (factorial(N) * (N + 2) / 2);
    } // Driver Code public static void Main() { int N = 6; Console.WriteLine(nthTerm(N)); } } //
    This code is contributed by ChitraNayal [tabby title="PHP"]
```

```
<?php
// PHP program to find
// N-th term of the series:
// 1, 4, 15, 72, 420...
```

```
// Function for finding
// factorial of N
```

```
function factorial($N)
{
```

```
    $fact = 1;
    for($i = 1; $i <= $N; $i++)
        $fact = $fact * $i;
```

```
    // return factorial of N
    return $fact;
}
```

```
// Function for calculating
```

```
// Nth term of series
function nthTerm($N)
{
    // return nth term
    return (factorial($N) *
            ($N + 2) / 2);
}

// Driver code
$N = 6;

// Function Calling
echo nthTerm($N);

// This code is contributed
// by mits
?>
```

**Output:**

2880

**Time complexity:**  $O(N)$

**Note:** Above code wouldn't not work for large values of N. To find the values for large N, use the concept of [Factorial for large numbers](#).

**Improved By :** [ANKITRAI1](#), [tufan\\_gupta2000](#), [Mithun Kumar](#), [ChitraNayal](#)

**Source**

<https://www.geeksforgeeks.org/find-nth-term-of-series-1-4-15-72-420/>

## Chapter 5

# Count Occurences of Anagrams

Count Occurences of Anagrams - GeeksforGeeks

Given a word and a text, return the count of the occurences of anagrams of the word in the text(For eg: anagrams of word for are for, ofr, rof etc.))

Examples:

Input : forxxorfxdofr  
      for

Output : 3

Explanation : Anagrams of the word for - for, orf, ofr appear in the text and hence the count is 3.

Input : aabaabaa  
      aaba

Output : 4

Explanation : Anagrams of the word aaba - aaba, abaa each appear twice in the text and hence the count is 4.

A **simple approach** is to traverse from start of the string considering substrings of length equal to the length of the given word and then check if this substring has all the characters of word.

```
// A Simple Java program to count anagrams of a
// pattern in a text.
import java.io.*;
import java.util.*;

public class GFG {
```

```
// Function to find if two strings are equal
static boolean araAnagram(String s1,
                          String s2)
{
    // converting strings to char arrays
    char[] ch1 = s1.toCharArray();
    char[] ch2 = s2.toCharArray();

    // sorting both char arrays
    Arrays.sort(ch1);
    Arrays.sort(ch2);

    // Check for equality of strings
    if (Arrays.equals(ch1, ch2))
        return true;
    else
        return false;
}

static int countAnagrams(String text, String word)
{
    int N = text.length();
    int n = word.length();

    // Initialize result
    int res = 0;

    for (int i = 0; i <= N - n; i++) {

        String s = text.substring(i, i + n);

        // Check if the word and substring are
        // anagram of each other.
        if (araAnagram(word, s))
            res++;
    }

    return res;
}

// Driver code
public static void main(String args[])
{
    String text = "forxxorfxdofr";
    String word = "for";
    System.out.print(countAnagrams(text, word));
}
}
```



**Output:**

3

An **Efficient Solution** is to use count array to check for anagrams, we can construct current count window from previous window in  $O(1)$  time using sliding window concept.

```
// An efficient Java program to count anagrams of a
// pattern in a text.
import java.io.*;
import java.util.*;

public class GFG {
    final static int MAX_CHAR = 256

    // Function to find if two strings are equal
    static boolean isCountZero(int[] count)
    {
        for (int i = 0; i < MAX_CHAR; i++)
            if (count[i] != 0)
                return false;
        return true;
    }

    static int countAnagrams(String text, String word)
    {
        int N = text.length();
        int n = word.length();

        // Check for first window. The idea is to
        // use single count array to match counts
        int[] count = new int[MAX_CHAR];
        for (int i = 0; i < n; i++)
            count[word.charAt(i)]++;
        for (int i = 0; i < n; i++)
            count[text.charAt(i)]--;

        // If first window itself is anagram
        int res = 0;
        if (isCountZero(count))
            res++;

        for (int i = n; i < N; i++) {

            // Add last character of current
            // window
```

```
        count[text.charAt(i)]--;

        // Remove first character of previous
        // window.
        count[text.charAt(i - n)]++;

        // If count array is 0, we found an
        // anagram.
        if (isCountZero(count))
            res++;
    }
    return res;
}

// Driver code
public static void main(String args[])
{
    String text = "forxxorfxdofr";
    String word = "for";
    System.out.print(countAnagrams(text, word));
}
}
```

**Output:**

3

**Source**

<https://www.geeksforgeeks.org/count-occurences-of-anagrams/>

## Chapter 6

# Minimum characters to be replaced to remove the given substring

Minimum characters to be replaced to remove the given substring - GeeksforGeeks

Given two strings **str1** and **str2**. The task is to find the minimum number of characters to be replaced by \$ in string **str1** such that **str1** does not contain string **str2** as any substring.

**Examples:**

```
Input: str1 = "intellect", str2 = "tell"
Output: 1
4th character of string "str1" can be replaced by $
such that "int$llect" it does not contain "tell"
as a substring.
```

```
Input: str1 = "google", str2 = "apple"
Output: 0
```

**Approach** is similar to [Searching for Patterns | Set 1 \(Naive Pattern Searching\)](#).

The idea is to find the leftmost occurrence of the string 'str2' in the string 'str1'. If all the characters of 'str1' match with 'str2', we will replace (or increment our answer with one) the last symbol of occurrence and increment the index of string 'str1', such that it checks again for the substring after the replaced character(that is index i will be equal to **i+length(b)-1**).

Below is the implementation of the above approach:

C++

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

// function to calculate minimum
// characters to replace
int replace(string A, string B)
{
    int n = A.length(), m = B.length();
    int count = 0, i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {

            // mismatch occurs
            if (A[i + j] != B[j])
                break;
        }

        // if all characters matched, i.e,
        // there is a substring of 'a' which
        // is same as string 'b'
        if (j == m) {
            count++;

            // increment i to index m-1 such that
            // minimum characters are replaced
            // in 'a'
            i += m - 1;
        }
    }

    return count;
}

// Driver Code
int main()
{
    string str1 = "aaaaaaaa";
    string str2 = "aaa";

    cout << replace(str1 , str2);

    return 0;
}
```

## Java

```
// Java implementation of
// above approach
import java.io.*;

// function to calculate minimum
// characters to replace
class GFG
{
static int replace(String A, String B)
{

    int n = A.length(), m = B.length();
    int count = 0, i, j;

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {

            // mismatch occurs
            if(i + j >= n)
                break;
            else if (A.charAt(i + j) != B.charAt(j))
                break;
        }

        // if all characters matched, i.e,
        // there is a substring of 'a' which
        // is same as string 'b'
        if (j == m)
        {
            count++;

            // increment i to index m-1 such that
            // minimum characters are replaced
            // in 'a'
            i += m - 1;
        }
    }

    return count;
}

// Driver Code
public static void main(String args[])
```

```
{
    String str1 = "aaaaaaaa";
    String str2 = "aaa";

    System.out.println(replace(str1 , str2));
}
}
```

// This code is contributed by Subhadeep

**Output:**

2

**Time Complexity:**  $O(len1 * len2)$ , where len1 is the length of first string and len2 is the length of second string.

Also, this problem can be solved directly by using Python's in-built function-**string1.count(string2)**

```
// Python program to find minimum numbers
// of characters to be replaced to
// remove the given substring
str1 = "aaaaaaaa"
str2 = "aaa"

# inbuilt function
answer = str1.count(str2)
print(answer)
```

**Output:**

2

Improved By : [tufan\\_gupta2000](#)

**Source**

<https://www.geeksforgeeks.org/minimum-characters-to-be-replaced-to-remove-the-given-substring/>

## Chapter 7

# Custom Building Cryptography Algorithms (Hybrid Cryptography)

Custom Building Cryptography Algorithms (Hybrid Cryptography) - GeeksforGeeks

Cryptography can be defined as an art of encoding and decoding the patterns (in the form of messages).

Cryptography is a very straightforward concept which deals with manipulating the strings (or text) to make them unreadable for the intermediate person. It has a very effective way to **encrypt or decrypts** the text coming from the other parties. Some of the examples are, Caesar Cipher, Viginere Cipher, Columnner Cipher, DES, AES and the list continues. To develop custom cryptography algorithm, hybrid encryption algorithms can be used.

**Hybrid Encryption is a concept in cryptography which combines/merge one/two cryptography algorithms to generate more effective encrypted text.**

Example:

### **FibBil Cryptography Algorithm**

#### ***Problem Statement:***

Program to generate an encrypted text, by computing Fibonacci Series, adding the terms of Fibonacci Series with each plaintext letter, until the length of the key.

#### ***Algorithm:***

**For Encryption:** Take an input plain text and key from the user, reverse the plain text and concatenate the plain text with the key, Copy the string into an array. After copying, separate the array elements into two parts, EvenArray, and OddArray in which even index of an array will be placed in EvenArray and same for OddArray. Start generating the Fibonacci Series  $F(i)$  up-to-the length of the key<sub>j</sub> such that  $c=i+j$  where c is cipher text

with mod 26. Append all the  $c^{\text{th}}$  elements in a CipherString and, so Encryption Done!. When sum up concept is use, it highlights of implementing Caesar Cipher.

**For Decryption:** Vice Versa of the Encryption Algorithm

**Example for the Algorithm:**

**Input:** *hello*

**Key:** *abcd*

**Output:** *riobkxezg*

Reverse the input, olleh, append this with the key i.e. ollehabcd.

EvenString: leac

OddString: olhbd

As key length is 4, 4 times loop will be generated including FibNum 0, which is ignored.

**For EvenArray Ciphers:**

**FibNum: 1**

In Even Array for l and FibNum 1 cip is k

In Even Array for e and FibNum 1 cip is d

In Even Array for a and FibNum 1 cip is z

In Even Array for c and FibNum 1 cip is b

**FibNum: 2**

In Even Array for l and FibNum 2 cip is j

In Even Array for e and FibNum 2 cip is c

In Even Array for a and FibNum 2 cip is y

In Even Array for c and FibNum 2 cip is a

**FibNum: 3** (Final Computed letters)

**In Even Array for l and FibNum 3 cip is i**

**In Even Array for e and FibNum 3 cip is b**

**In Even Array for a and FibNum 3 cip is x**

**In Even Array for c and FibNum 3 cip is z**

**For OddArray Ciphers**

**FibNum: 1**

In Odd Array for o and FibNum 1 cip is p

In Odd Array for l and FibNum 1 cip is m

In Odd Array for h and FibNum 1 cip is i

In Odd Array for b and FibNum 1 cip is c

In Odd Array for d and FibNum 1 cip is e

**FibNum: 2**

In Odd Array for o and FibNum 2 cip is q

In Odd Array for l and FibNum 2 cip is n

In Odd Array for h and FibNum 2 cip is j

In Odd Array for b and FibNum 2 cip is d

In Odd Array for d and FibNum 2 cip is f

**FibNum: 3** (Final Computed letters)

**In Odd Array for o and FibNum 3 cip is r**

**In Odd Array for l and FibNum 3 cip is o**

**In Odd Array for h and FibNum 3 cip is k**



**In Odd Array for b and FibNum 3 cip is e**

**In Odd Array for d and FibNum 3 cip is g**

Arrange EvenArrayCiphers and OddArrayCiphers in their index order, so final String Cipher will be, **riobkxezg**

***Program:***

```
import java.util.*;
import java.lang.*;

class GFG {

    public static void main(String[] args)
    {
        String pass = "hello";
        String key = "abcd";
        System.out.println(encryptText(pass, key));
    }

    public static String encryptText(String password, String key)
    {
        int a = 0, b = 1, c = 0, m = 0, k = 0, j = 0;
        String cipher = "", temp = "";

        // Declare a password string
        StringBuffer pw = new StringBuffer(password);

        // Reverse the String
        pw = pw.reverse();
        pw = pw.append(key);

        // For future Purpose
        temp = pw.toString();
        char stringArray[] = temp.toCharArray();
        String evenString = "", oddString = "";

        // Declare EvenArray for storing
        // even index of stringArray
        char evenArray[];

        // Declare OddArray for storing
        // odd index of stringArray
        char oddArray[];

        // Storing the positions in their respective arrays
        for (int i = 0; i < stringArray.length; i++) {
            if (i % 2 == 0) {
                oddString = oddString + Character.toString(stringArray[i]);
            }
        }
    }
}
```

```

    }
    else {
        evenString = evenString + Character.toString(stringArray[i]);
    }
}
evenArray = new char[evenString.length()];
oddArray = new char[oddString.length()];

// Generate a Fibonacci Series
// Upto the Key Length
while (m <= key.length()) {
    // As it always starts with 1
    if (m == 0)
        m = 1;

    else {

        // Logic For Fibonacci Series
        a = b;
        b = c;
        c = a + b;
        for (int i = 0; i < evenString.length(); i++) {
            // Caesar Cipher Algorithm Start for even positions
            int p = evenString.charAt(i);
            int cip = 0;
            if (p == '0' || p == '1' || p == '2' || p == '3' || p == '4'
                || p == '5' || p == '6'
                || p == '7' || p == '8' || p == '9') {
                cip = p - c;
                if (cip < '0')
                    cip = cip + 9;
            }
            else {
                cip = p - c;
                if (cip < 'a') {
                    cip = cip + 26;
                }
            }
            evenArray[i] = (char)cip;
            /* Caesar Cipher Algorithm End*/
        }
        for (int i = 0; i < oddString.length(); i++) {
            // Caesar Cipher Algorithm Start for odd positions
            int p = oddString.charAt(i);
            int cip = 0;
            if (p == '0' || p == '1' || p == '2' || p == '3' || p == '4'
                || p == '5' || p == '6'
                || p == '7' || p == '8' || p == '9') {

```

```

        cip = p + c;
        if (cip > '9')
            cip = cip - 9;
    }
    else {
        cip = p + c;
        if (cip > 'z') {
            cip = cip - 26;
        }
    }
    oddArray[i] = (char)cip;
    // Caesar Cipher Algorithm End
}

    m++;
}

}

// Storing content of even and
// odd array to the string array
for (int i = 0; i < stringArray.length; i++) {
    if (i % 2 == 0) {
        stringArray[i] = oddArray[k];
        k++;
    }
    else {
        stringArray[i] = evenArray[j];
        j++;
    }
}

// Generating a Cipher Text
// by stringArray (Caesar Cipher)
for (char d : stringArray) {
    cipher = cipher + d;
}

// Return the Cipher Text
return cipher;
}
}

```

**Output:**

riobkxezg

*Conclusion:*

Hybrid Algorithms for the cryptography are effective and so, it is not very easy to detect the pattern and decode the message. Here, the algorithm is a combination of mathematical function and Caesar Cipher, so as to implement Hybrid Cryptography Algorithm.

### **Source**

<https://www.geeksforgeeks.org/custom-building-cryptography-algorithms-hybrid-cryptography/>

## Chapter 8

# Program to find all match of a regex in a string

Program to find all match of a regex in a string - GeeksforGeeks

**Prerequisite:** [smatch | Regex \(Regular Expressions\) in C++](#)

Given a regex, the task is to find all regex matches in a string.

- Without using iterator:

```
// C++ program to find all the matches
#include <bits/stdc++.h>
using namespace std;
int main()
{
    string subject("My GeeksforGeeks is my "
                  "GeeksforGeeks none of your GeeksforGeeks");

    // Template instantiations for
    // extracting the matching pattern.
    smatch match;
    regex r("GeeksforGeeks");
    int i = 1;
    while (regex_search(subject, match, r)) {
        cout << "\nMatched string is " << match.str(0) << endl
              << "and it is found at position "
              << match.position(0)<<endl;
        i++;

        // suffix to find the rest of the string.
        subject = match.suffix().str();
    }
}
```

```
    return 0;
}
```

**Output:**

```
Matched string is GeeksforGeeks
and it is found at position 3
```

```
Matched string is GeeksforGeeks
and it is found at position 7
```

```
Matched string is GeeksforGeeks
and it is found at position 14
```

**Note:** Above code is running perfectly fine but the problem is input string will be lost.

- **Using iterator:**

Object can be constructed by calling the constructor with three parameters: a string iterator indicating the starting position of the search, a string iterator indicating the ending position of the search, and the regex object. Construct another iterator object using the default constructor to get an end-of-sequence iterator.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    string subject("geeksforgeeksabcdefghg"
                  "eeksforgeeksabcdgeeksforgeeks");

    // regex object.
    regex re("geeks(for)geeks");

    // finding all the match.
    for (sregex_iterator it = sregex_iterator(subject.begin(), subject.end(), re);
         it != sregex_iterator(); it++) {
        smatch match;
        match = *it;
        cout << "\nMatched string is = " << match.str(0)
              << "\nand it is found at position "
              << match.position(0) << endl;
        cout << "Capture " << match.str(1)
              << " at position " << match.position(1) << endl;
    }
    return 0;
}
```

**Output:**

```
Matched string is = geeksforgeeks  
and it is found at position 0  
Capture for at position 5
```

```
Matched string is = geeksforgeeks  
and it is found at position 21  
Capture for at position 26
```

```
Matched string is = geeksforgeeks  
and it is found at position 38  
Capture for at position 43
```

## Source

<https://www.geeksforgeeks.org/program-to-find-all-match-of-a-regex-in-a-string/>

## Chapter 9

# KLA Tencor Interview Experience | Set 3

KLA Tencor Interview Experience | Set 3 - GeeksforGeeks

### Round 1

Online test on HackerRank contains 2 Coding Questions

1. Given a image represented as 2d array of 0 and 1, find the size of the biggest cluster in the image.

Cluster one or more adjunct cell with 1 is cluster. Even in a single cell which surrounded by all 0 will be clustered.

Adjacent cell : cell on left, right, top and bottom diagonal cells are not considered for cluster.

Size-of-Cluster: number of 1 in the cluster

input:

4

5

10001

00110

10000

11110

output:

5

2. Given a 2D char array[m][n] and a word w[k], you need to find all the occurrences of w in array could be appeared in row(left, right) col(top, bottom) and in any diagonals.

Input:

5

6

abcdef

ahijkl

abccde

bbcuvx



cddwww

abc

output:

(0, 0, horizontal)

(2, 0, horizontal)

(2, 0, vertical)

(1, 0, diagonal)

## Round 2

This is a 1-hour telephonic interview.

1. Tell me about yourself.
2. Discussion on constructor/instructor
3. Memory leak.
4. Virtual function some more cpp questions.
5. Discussion on core dump, corruption, how handle all these scenarios.
6. Discussion on mallow(), new().
7. Find the k most frequent words from a file.  
<https://www.geeksforgeeks.org/find-the-k-most-frequent-words-from-a-file/>

## Round 3

This is a 1-hour telephonic interview.

1. Discussion on my current project in depth.
2. Leaders in an array  
<https://www.geeksforgeeks.org/leaders-in-an-array/>
3. Find the Number Occurring Odd Number of Times  
<https://www.geeksforgeeks.org/find-the-number-occurring-odd-number-of-times/>
4. Design a contact app for android.(mostly focus on efficient algorithm).

## Round 4

Face To Face Interview on Programming Questions Explain on Paper (1-hour)

1. Discussion on my current project in depth.
2. Clone a linked list with next and random pointer (all possible approach).  
<https://www.geeksforgeeks.org/a-linked-list-with-next-and-arbit-pointer/>
3. Find the middle of a given linked list  
<https://www.geeksforgeeks.org/write-a-c-function-to-print-the-middle-of-the-linked-list/>
4. Reverse a linked list  
<https://www.geeksforgeeks.org/reverse-a-linked-list/>
5. Given a number N you need make it to 0 by subtracting K or 1 from N, but condition is you need select K in such a way that after subtracting K the result should be factored of N.  
example N = 10 that first K=5 after subtracting K from N 10-5=5 hence 5 is factors of 10.  
Find minimum number of substation operation to make it 0.
6. Some more questions on array and linked list.

## Round 5

Face To Face Interview on Programming Questions Explain on Paper (1-hour)

This round was taken by a senior folk.

1. Given a image in the form of 2D array of numbers. You need to corrupt that images and return.

Condition for corruption is the element at index  $[x][y]$  should contain average of surrounding numbers.

Example.

1234

6789 –here in place of 7  $\rightarrow$  4

2345

2. Next Greater Element

<https://www.geeksforgeeks.org/next-greater-element/>

### Round 6

Face To Face Interview on Design and OOPS(1- hour).

This round was taken by a manager.

1. Design class diagram for coffee machine.(with all possible classes object, functions and type of data)

Most focus on object interaction.

### Round 7

This round was taken by a senior manager.

1. Tell me about your self, your family and all?.

2. What is the one thing that can make you stay with KLA Tencor?

3. Design ola/uber cost estimation functionality Focus on Factory design pattern.

4. More HR related questions.

### Round 8

This round was based on Behavioral Skills taken by senior HR Manager.

1. Tell me about your self.

2. Why KLA Tencor.

3. More HR related questions.

I prepared mostly from GeeksforGeeks and I want to say thanks to the content writers of Geeks for providing the best solutions. This is one of the best sites.

Note-

1. Mostly Focus on algorithm, How efficient you can write algorithm.

2. Also Focus on system design questions. For getting some Idea or to start with system design question refer to below given link.

[https://www.youtube.com/watch?v=UzLMhgg3\\_Wc&list=PLrmLmBdmllps7GJJWW9I7N0P0rB0C3eY2](https://www.youtube.com/watch?v=UzLMhgg3_Wc&list=PLrmLmBdmllps7GJJWW9I7N0P0rB0C3eY2)

3. To begin with design pattern refer below given link to start with.

<https://www.youtube.com/watch?v=rI4kdGLaUiQ&list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd>

4. Ask questions at the end of your interview to the interviewers.

5. Before start writing the code try to explain your algorithm.

## **Source**

<https://www.geeksforgeeks.org/kla-tencor-interview-experience-set-3/>

## Chapter 10

# Check if a string can be formed from another string using given constraints

Check if a string can be formed from another string using given constraints - GeeksforGeeks

Given two strings S1 and S2(all characters are in lower-case). The task is to check if S2 can be formed from S1 using given constraints:

1. Characters of S2 is there in S1 if there are two 'a' in S2, then S1 should have two 'a' also.
2. If any character of S2 is not present in S1, check if the previous two ASCII characters are there in S1. e.g., if 'e' is there in S2 and not in S1, then 'c' and 'd' can be used from S1 to make 'e'.

**Note:** All characters from S1 can be used only once.

**Examples:**

**Input:** S= abbat, W= cat

**Output:** YES

'c' is formed from 'a' and 'b', 'a' and 't' is present in S1.

**Input:** S= abbt, W= cat

**Output:** NO

'c' is formed from 'a' and 'b', but to form the next character

'a' in S2, there is no more unused 'a' left in S1.

**Approach:** The above problem can be solved using hashing. The count of all the characters in S1 is stored in a hash-table. Traverse in the string, and check if the character in S2 is there in the hash-table, reduce the count of that particular character in the hash-table. If the character is not there in the hash-table, check if the previous two ASCII characters are there in the hash-table, then reduce the count of the previous two ASCII characters in the

hash-table. If all the characters can be formed from S1 using the given constraints, the string S2 can be formed from S1, else it cannot be formed.

Below is the implementation of the above approach:

```
// CPP program to Check if a given
// string can be formed from another
// string using given constraints
#include <bits/stdc++.h>
using namespace std;

// Function to check if S2 can be formed of S1
bool check(string S1, string S2)
{
    // length of strings
    int n1 = S1.size();
    int n2 = S2.size();

    // hash-table to store count
    unordered_map<int, int> mp;

    // store count of each character
    for (int i = 0; i < n1; i++) {
        mp[S1[i]]++;
    }

    // traverse and check for every character
    for (int i = 0; i < n2; i++) {

        // if the character of s2 is present in s1
        if (mp[S2[i]]) {
            mp[S2[i]]--;
        }

        // if the character of s2 is not present in
        // S1, then check if previous two ASCII characters
        // are present in S1
        else if (mp[S2[i] - 1] && mp[S2[i] - 2]) {

            mp[S2[i] - 1]--;
            mp[S2[i] - 2]--;
        }
        else {
            return false;
        }
    }

    return true;
}
```

```
// Driver Code
int main()
{
    string S1 = "abbat";
    string S2 = "cat";

    // Calling function to check
    if (check(S1, S2))
        cout << "YES";
    else
        cout << "NO";
}
```

**Output:**

YES

**Source**

<https://www.geeksforgeeks.org/check-if-a-string-can-be-formed-from-another-string-using-given-constraints/>

## Chapter 11

# Largest connected component on a grid

Largest connected component on a grid - GeeksforGeeks

Given a grid with different colors in a different cell, each color represented by a different number. The task is to find out the largest connected component on the grid. Largest component grid refers to a maximum set of cells such that you can move from any cell to any other cell in this set by only moving between side-adjacent cells from the set.

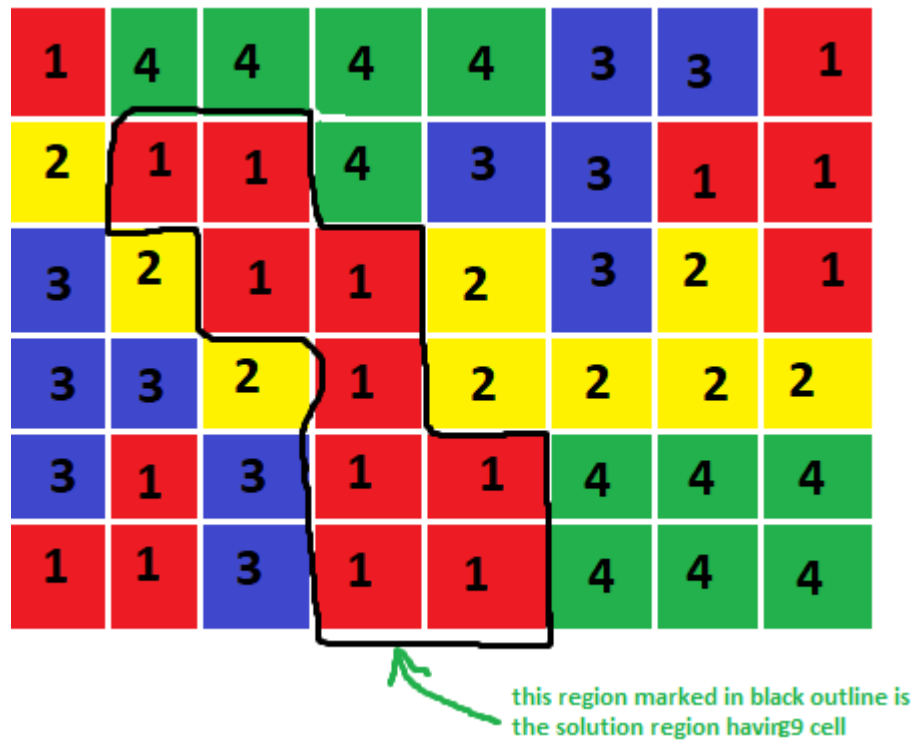
**Examples:**

Input :

1	4	4	4	4	3	3	1
2	1	1	4	3	3	1	1
3	2	1	1	2	3	2	1
3	3	2	1	2	2	2	2
3	1	3	1	1	4	4	4
1	1	3	1	1	4	4	4

Grid of different colors

Output : 9

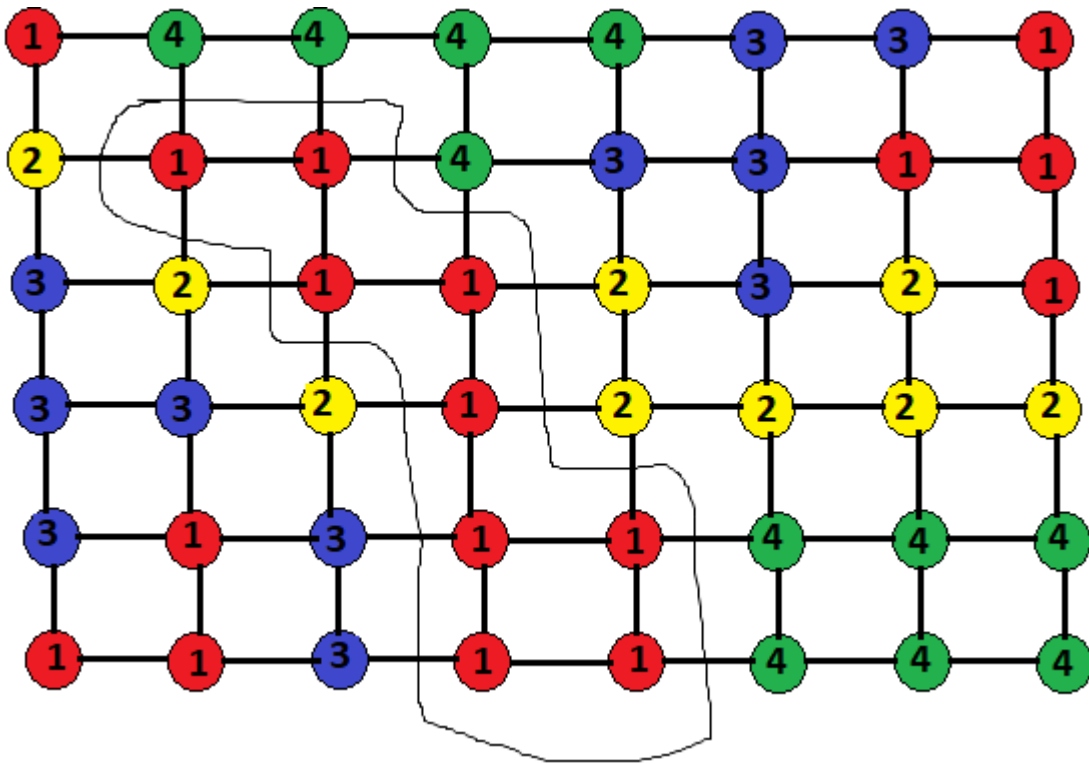


Largest connected component of grid

### Approach :

The approach is to visualize the given grid as a graph with each cell representing a separate node of the graph and each node connected to four other nodes which are to immediately up, down, left, and right of that grid. Now doing a [BFS](#) search for every node of the graph, find *all the nodes connected to the current node with same color value as the current node*. Here is the graph for above example :





Graph representation of grid

At every cell  $(i, j)$ , a BFS can be done. The possible moves from a cell will be either to **right, left, top or bottom**. Move to only those cells which are in range and are of the same color. If the same nodes have been visited previously, then the largest component value of the grid is stored in `result[][]` array. Using memoization, reduce the number of BFS on any cell. `visited[][]` array is used to mark if the cell has been visited previously and count stores the count of the connected component when a BFS is done for every cell. Store the maximum of the count and print the resultant grid using `result[][]` array.

Below is the illustration of the above approach:

C++

```
// CPP program to print the largest
// connected component in a grid
#include <bits/stdc++.h>
using namespace std;

const int n = 6;
const int m = 8;
```

```
// stores information about which cell
// are already visited in a particular BFS
int visited[n][m];

// result stores the final result grid
int result[n][m];

// stores the count of cells in the largest
// connected component
int COUNT;

// Function checks if a cell is valid i.e it
// is inside the grid and equal to the key
bool is_valid(int x, int y, int key, int input[n][m])
{
    if (x < n && y < m && x >= 0 && y >= 0) {
        if (visited[x][y] == false && input[x][y] == key)
            return true;
        else
            return false;
    }
    else
        return false;
}

// BFS to find all cells in
// connection with key = input[i][j]
void BFS(int x, int y, int i, int j, int input[n][m])
{
    // terminating case for BFS
    if (x != y)
        return;

    visited[i][j] = 1;
    COUNT++;

    // x_move and y_move arrays
    // are the possible movements
    // in x or y direction
    int x_move[] = { 0, 0, 1, -1 };
    int y_move[] = { 1, -1, 0, 0 };

    // checks all four points connected with input[i][j]
    for (int u = 0; u < 4; u++)
        if (is_valid(i + y_move[u], j + x_move[u], x, input))
            BFS(x, y, i + y_move[u], j + x_move[u], input);
}
```

```
// called every time before a BFS
// so that visited array is reset to zero
void reset_visited()
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            visited[i][j] = 0;
}

// If a larger connected component
// is found this function is called
// to store information about that component.
void reset_result(int key, int input[n][m])
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (visited[i][j] && input[i][j] == key)
                result[i][j] = visited[i][j];
            else
                result[i][j] = 0;
        }
    }
}

// function to print the result
void print_result(int res)
{
    cout << "The largest connected "
          << "component of the grid is :" << res << "\n";

    // prints the largest component
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (result[i][j])
                cout << result[i][j] << " ";
            else
                cout << ". ";
        }
        cout << "\n";
    }
}

// function to calculate the largest connected
// component
void computeLargestConnectedGrid(int input[n][m])
{
    int current_max = INT_MIN;
```

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        reset_visited();
        COUNT = 0;

        // checking cell to the right
        if (j + 1 < m)
            BFS(input[i][j], input[i][j + 1], i, j, input);

        // updating result
        if (COUNT >= current_max) {
            current_max = COUNT;
            reset_result(input[i][j], input);
        }
        reset_visited();
        COUNT = 0;

        // checking cell downwards
        if (i + 1 < n)
            BFS(input[i][j], input[i + 1][j], i, j, input);

        // updating result
        if (COUNT >= current_max) {
            current_max = COUNT;
            reset_result(input[i][j], input);
        }
    }
}
print_result(current_max);
}
// Drivers Code
int main()
{
    int input[n][m] = { { 1, 4, 4, 4, 4, 3, 3, 1 },
                        { 2, 1, 1, 4, 3, 3, 1, 1 },
                        { 3, 2, 1, 1, 2, 3, 2, 1 },
                        { 3, 3, 2, 1, 2, 2, 2, 2 },
                        { 3, 1, 3, 1, 1, 4, 4, 4 },
                        { 1, 1, 3, 1, 1, 4, 4, 4 } };

    // function to compute the largest
    // connected component in the grid
    computeLargestConnectedGrid(input);
    return 0;
}
```

**Java**

```
// Java program to print the largest
// connected component in a grid
import java.util.*;
import java.lang.*;
import java.io.*;

class GFG
{
    static final int n = 6;
    static final int m = 8;

    // stores information about which cell
    // are already visited in a particular BFS
    static final int visited[] [] = new int [n] [m];

    // result stores the final result grid
    static final int result[] [] = new int [n] [m];

    // stores the count of
    // cells in the largest
    // connected component
    static int COUNT;

    // Function checks if a cell
    // is valid i.e it is inside
    // the grid and equal to the key
    static boolean is_valid(int x, int y,
                           int key,
                           int input[] [])
    {
        if (x < n && y < m &&
            x >= 0 && y >= 0)
        {
            if (visited[x][y] == 0 &&
                input[x][y] == key)
                return true;
            else
                return false;
        }
        else
            return false;
    }

    // BFS to find all cells in
    // connection with key = input[i][j]
    static void BFS(int x, int y, int i,
                   int j, int input[] [])
    {
```

```
// terminating case for BFS
if (x != y)
    return;

visited[i][j] = 1;
COUNT++;

// x_move and y_move arrays
// are the possible movements
// in x or y direction
int x_move[] = { 0, 0, 1, -1 };
int y_move[] = { 1, -1, 0, 0 };

// checks all four points
// connected with input[i][j]
for (int u = 0; u < 4; u++)
    if ((is_valid(i + y_move[u],
        j + x_move[u], x, input)) == true)
        BFS(x, y, i + y_move[u],
            j + x_move[u], input);
}

// called every time before
// a BFS so that visited
// array is reset to zero
static void reset_visited()
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            visited[i][j] = 0;
}

// If a larger connected component
// is found this function is
// called to store information
// about that component.
static void reset_result(int key,
                        int input[][])
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (visited[i][j] == 1 &&
                input[i][j] == key)
                result[i][j] = visited[i][j];
            else
                result[i][j] = 0;
        }
    }
}
```

```
    }
  }
}

// function to print the result
static void print_result(int res)
{
    System.out.println ("The largest connected " +
                        "component of the grid is :" +
                        res );

    // prints the largest component
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (result[i][j] != 0)
                System.out.print(result[i][j] + " ");
            else
                System.out.print(". ");
        }
        System.out.println();
    }
}

// function to calculate the
// largest connected component
static void computeLargestConnectedGrid(int input[][])
{
    int current_max = Integer.MIN_VALUE;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            reset_visited();
            COUNT = 0;

            // checking cell to the right
            if (j + 1 < m)
                BFS(input[i][j], input[i][j + 1],
                    i, j, input);

            // updating result
            if (COUNT >= current_max)
            {
                current_max = COUNT;
                reset_result(input[i][j], input);
            }
        }
    }
}
```

```
    }
    reset_visited();
    COUNT = 0;

    // checking cell downwards
    if (i + 1 < n)
        BFS(input[i][j],
            input[i + 1][j], i, j, input);

    // updating result
    if (COUNT >= current_max)
    {
        current_max = COUNT;
        reset_result(input[i][j], input);
    }
}
}
print_result(current_max);
}
// Driver Code
public static void main(String args[])
{
    int input[][] = {{1, 4, 4, 4, 4, 3, 3, 1},
                     {2, 1, 1, 4, 3, 3, 1, 1},
                     {3, 2, 1, 1, 2, 3, 2, 1},
                     {3, 3, 2, 1, 2, 2, 2, 2},
                     {3, 1, 3, 1, 1, 4, 4, 4},
                     {1, 1, 3, 1, 1, 4, 4, 4}};

    // function to compute the largest
    // connected component in the grid
    computeLargestConnectedGrid(input);
}
}
```

// This code is contributed by Subhadeep

**Output:**

The largest connected component of the grid is :9

```
. . . . .
. 1 1 . . .
. . 1 1 . . .
. . . 1 . . .
. . . 1 1 . . .
. . . 1 1 . . .
```



**Improved By :** [tufan\\_gupta2000](#)

**Source**

<https://www.geeksforgeeks.org/largest-connected-component-on-a-grid/>

## Chapter 12

# Make array elements equal in Minimum Steps

Make array elements equal in Minimum Steps - GeeksforGeeks

Given an array of **N** elements where the first element is a non zero positive number **M**, and the rest  $N - 1$  elements are 0, the task is to calculate the minimum number of steps required to make the entire array equal while abiding by the following rules:

1. The **i<sup>th</sup>** element can be increased by one if and only if **i-1<sup>th</sup>** element is strictly greater than the **i<sup>th</sup>** element
2. If the **i<sup>th</sup>** element is being increased by one then the **i+1<sup>th</sup>** cannot be increased at the same time.(i.e consecutive elements cannot be increased at the same time)
3. Multiple elements can be incremented simultaneously in a single step.

**Examples:**

Input :  $N = 3, M = 4$

Output : 8

Explanation:

array is 4 0 0

In 4 steps element at index 1 is increased, so the array becomes **{4, 4, 0}**. In the next 4 steps the element at index 3 is increased so array becomes **{4, 4, 4}**  
Thus,  $4 + 4 = 8$  operations are required to make all the array elements equal

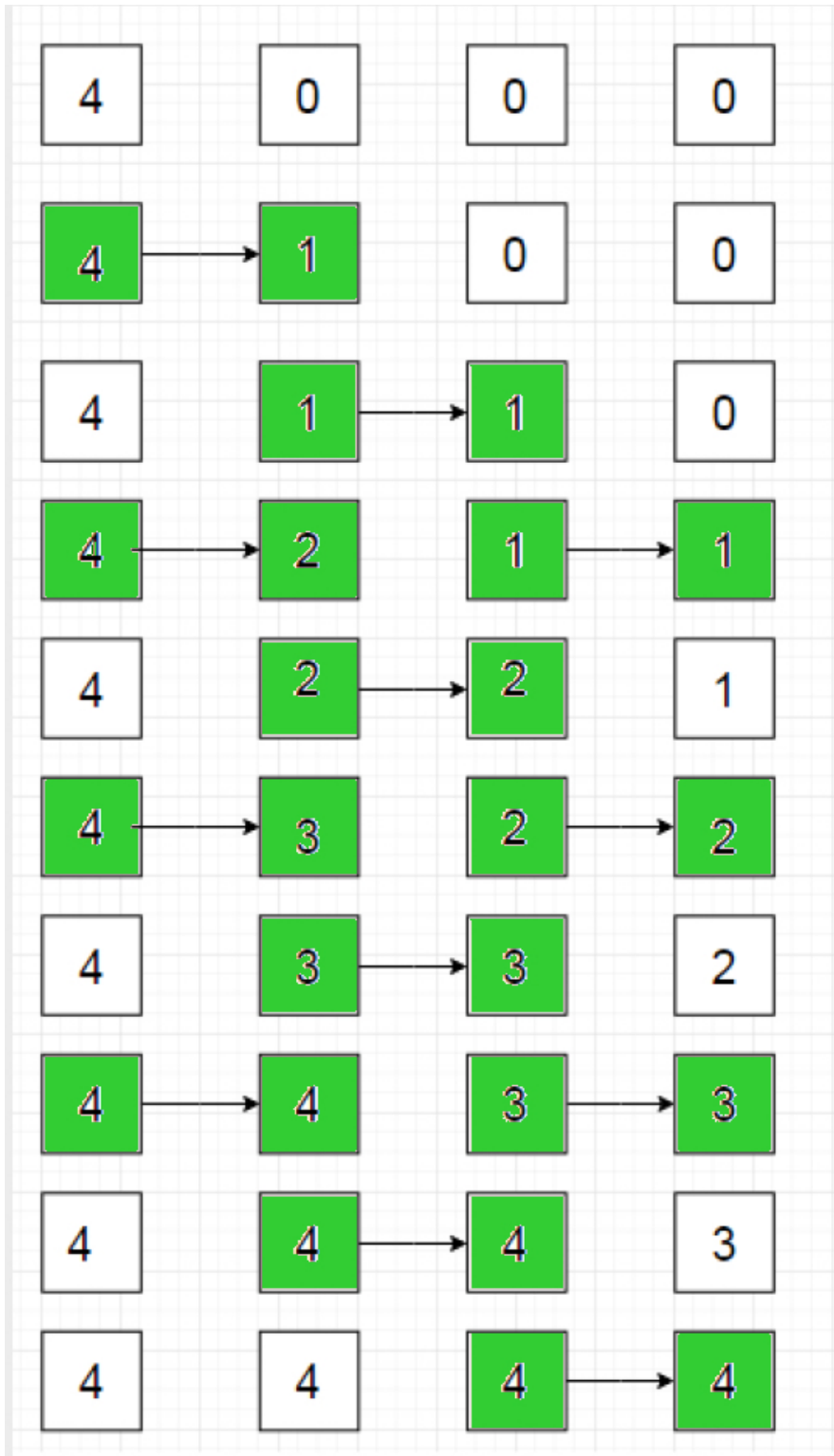
Input :  $N = 4, M = 4$

Output : 9

**Explanation:**

The steps are shown in the flowchart given below

Refer to the flowchart given below.



**Approach:**

To maximise the Number of Increments per Step, more number of Unbalances are created ( $\text{array}[i] > \text{array}[i+1]$ ),

Step 1, element 0 > element 1 so element 1 is incremented,

Step 2, element 1 > element 2 so element 2 is incremented by 1

Step 3, element 0 > element 1 and element 2 > element 3 so element 1 & 3 are incremented by 1

Step 4, element 1 > element 2 element 3 > element 4 so element 2 & 4 are incremented

Step 5, element 0 > element 1; element 2 > element 3 ; element 4 > element 5; so element 1, 3, & 5 are incremented.

and so on...

Consider the following array,

5 0 0 0 0

- 1) 5 1 0 0 0
- 2) 5 1 1 0 0
- 3) 5 2 1 1 0
- 4) 5 2 2 1 1
- 5) 5 3 2 2 1
- 6) 5 3 3 2 2
- 7) 5 4 3 3 2
- 8) 5 4 4 3 3
- 9) 5 5 4 4 3
- 10) 5 5 5 4 4
- 11) 5 5 5 5 4
- 12) 5 5 5 5 5
- 13) 5 5 5 5 5

Notice that after an unbalance is created (i.e  $\text{array}[i] > \text{array}[i+1]$ ) the element gets incremented by one in alternate steps. In step 1 element 1 gets incremented to 1, in step 2 element 2 gets incremented to 1, in step 3 element 3 gets incremented to 1, so in step  $n-1$ ,  $n-1^{\text{th}}$  element will become 1. After that  $n-1^{\text{th}}$  element is increased by 1 on alternate steps until it reaches the value at element 0. Then the entire array becomes equal.

So the pattern followed by the last element is

(0, 0, 0, ..., 0) till  $(N - 4)^{\text{th}}$  element becomes 1 which is  **$n-4$  steps**

and after that,

(0, 0, 1, 1, 2, 2, 3, 3, 4, 4, ...  $M - 1$ ,  $M - 1$ ,  $M$ ) which is  **$2 * m + 1$  steps.**

So the Final Result becomes  $(N - 3) + 2 * M$

There are a few corner cases which need to be handled, viz. When  $N = 1$ , array has only a single element, so the number of steps required = 0. and When  $N = 2$ , number of steps required equal to  $M$

C++

```
// C++ program to make the array elements equal in minimum steps

#include <bits/stdc++.h>
using namespace std;

// Returns the minumum steps required to make an array of N
// elements equal, where the first array element equals M
int steps(int N, int M)
{
    // Corner Case 1: When N = 1
    if (N == 1)
        return 0;

    // Corner Case 2: When N = 2
    else if (N == 2) // corner case 2
        return M;

    return 2 * M + (N - 3);
}

// Driver Code
int main()
{
    int N = 4, M = 4;
    cout << steps(N, M);
    return 0;
}
```

## Java

```
// Java program to make the array elements
// equal in minimum steps

import java.io.*;

class GFG {

    // Returns the minumum steps required
    // to make an array of N elements equal,
    // where the first array element equals M
    static int steps(int N, int M)
    {
        // Corner Case 1: When N = 1
        if (N == 1)
            return 0;

        // Corner Case 2: When N = 2
        else if (N == 2) // corner case 2
```

```
        return M;

        return 2 * M + (N - 3);
    }

    // Driver Code
    public static void main (String[] args)
    {
        int N = 4, M = 4;
        System.out.print( steps(N, M));
    }
}

// This code is contributed by anuj_67.
```

### Python3

```
# Python program to make
# the array elements equal
# in minimum steps

# Returns the minumum steps
# required to make an array
# of N elements equal, where
# the first array element
# equals M
def steps(N, M):

    # Corner Case 1: When N = 1
    if (N == 1):
        return 0

    # Corner Case 2: When N = 2
    elif(N == 2):
        return M

    return 2 * M + (N - 3)

# Driver Code
N = 4
M = 4
print(steps(N,M))

# This code is contributed
# by Shivi_Aggarwal.
```

### C#

```
// C# program to make the array
// elements equal in minimum steps
using System;

class GFG
{
    // Returns the minumum steps
    // required to make an array
    // of N elements equal, where
    // the first array element
    // equals M
    static int steps(int N, int M)
    {
        // Corner Case 1: When N = 1
        if (N == 1)
            return 0;

        // Corner Case 2: When N = 2
        else if (N == 2) // corner case 2
            return M;

        return 2 * M + (N - 3);
    }

    // Driver Code
    public static void Main ()
    {
        int N = 4, M = 4;
        Console.WriteLine(steps(N, M));
    }
}

// This code is contributed by anuj_67.
```

**Output:**

9

Improved By : [vt\\_m](#), [Shivi\\_Aggarwal](#)

**Source**

<https://www.geeksforgeeks.org/make-array-elements-equal-in-minimum-steps/>

## Chapter 13

# PHP | `ereg_replace()` Function

PHP | `ereg_replace()` Function - GeeksforGeeks

The `ereg_replace()` is an inbuilt function in PHP and is used to search a string pattern in an other string. If pattern is found in the original string then it will replace matching text with a replacement string. You may refer to the article on [Regular Expression](#) for basic understanding of pattern matching using regular expressions.

**Syntax:**

```
string ereg_replace ( $string_pattern, $replace_string, $original_string )
```

**Parameters Used:** This function accepts three mandatory parameters and all of these parameters are described below.

- ***\$string\_pattern:*** This parameter specifies the pattern to be searched in the `$original_string`. Its can be used with both array and string type which is parenthesized substrings.
- ***\$replace\_string:*** This parameter specifies the string by which the matching text will be replaced and it can be used with both array and string type. The replacement contain substring in the form of `\digit`, which replaces the text matching digit'th parenthesized substring and `\0` produce entire contents string.
- ***\$original\_string:*** This parameter specifies the input string and can be of both array and string type.

**Return Value:** This function returns a modified string or array if matches found. If matches not found in the original string then it will return unchanged original string or array.

**Note:** The `ereg_replace()` function is *case sensitive* in PHP. This function was *deprecated* in PHP 5.3.0, and *removed* in PHP 7.0.0.

Examples:



```
Input: $original_string = "Geeksforgeeks PHP article.";
       $string_pattern = "(.*)PHP(.*)";
       $replace_string = " You should read \\1all\\2";
Output: You should read Geeksforgeeks all article.
Explanation: Within the parenthesis "\\1" and "\\2" to access
             the part of string and replace with 'PHP' to 'all'.
```

```
Input: $original_string = "Geeksforgeeks is no:one computer
                           science portal.";
       $replace_string = '1';
       $original_string = ereg_replace('one', $replace_string,
                                       $original_string);
Output: Geeksforgeeks is no:1 computer science portal.
```

Below programs illustrate the *ereg\_replace()* function.

**Program 1:**

```
<?php

// Original input string
$original_string = "Write any topic .";

// Pattern to be searched
$string_pattern = "(.*)any(.*)";

// Replace string
$replace_string = " own yours own \\1biography\\2";

echo ereg_replace($patternstrVal, $replacesstrVal, $stringVal);

?>
```

Output:

Write own yours own biography topic.

**Note:** While using an integer value as the replacement parameter, we do not get expected result as the function interpret the number to ordinal value of character.

**Program 2:**

```
<?php

// Original input string
```

```
$original_string = "India To Become World's Fifth  
                    Largest Economy In 2018.";

// Replace string
$replace_string = 5;

// This function call will not show the expected output as the
// function interpret the number to ordinal value of character.
echo ereg_replace('Fifth',$replace_string, $original_string);

$original_string = "India To Become World's Fifth  
                    Largest Economy In 2018.";

// Replace String
$replace_string = '5';

// This function call will show
// the correct expected output
echo ereg_replace('Fifth',$replace_string, $original_string);

?>
```

Output:

```
India To Become World's  Largest Economy In 2018.
India To Become World's 5 Largest Economy In 2018.
```

**Reference:** <http://php.net/manual/en/function.ereg-replace.php>

## Source

[https://www.geeksforgeeks.org/php-ereg\\_replace-function/](https://www.geeksforgeeks.org/php-ereg_replace-function/)

## Chapter 14

# Check if strings are rotations of each other or not | Set 2

Check if strings are rotations of each other or not | Set 2 - GeeksforGeeks

Given two strings s1 and s2, check whether s2 is a rotation of s1.

**Examples:**

Input : ABACD, CDABA  
Output : True

Input : GEEKS, EKSGE  
Output : True

We have discussed an approach in [earlier](#) post which handles substring match as a pattern. In this post, we will be going to use **KMP algorithm's lps** (longest proper prefix which is also suffix) construction, which will help in finding the longest match of the prefix of string b and suffix of string a. By which we will know the **rotating point**, from this point match the characters. If all the characters are matched, then it is a rotation, else not.

Below is the basic implementation of the above approach.

**Java**

```
// Java program to check if two strings are rotations
// of each other.
import java.util.*;
import java.lang.*;
import java.io.*;
class stringMatching {
```

```
public static boolean isRotation(String a, String b)
{
    int n = a.length();
    int m = b.length();
    if (n != m)
        return false;

    // create lps[] that will hold the longest
    // prefix suffix values for pattern
    int lps[] = new int[n];

    // length of the previous longest prefix suffix
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to n-1
    while (i < n) {
        if (a.charAt(i) == b.charAt(len)) {
            lps[i] = ++len;
            ++i;
        }
        else {
            if (len == 0) {
                lps[i] = 0;
                ++i;
            }
            else {
                len = lps[len - 1];
            }
        }
    }

    i = 0;

    // match from that rotating point
    for (int k = lps[n - 1]; k < m; ++k) {
        if (b.charAt(k) != a.charAt(i++))
            return false;
    }
    return true;
}

// Driver code
public static void main(String[] args)
{
    String s1 = "ABACD";
    String s2 = "CDABA";
```

```
        System.out.println(isRotation(s1, s2) ? "1" : "0");
    }
}
```

## C#

```
// C# program to check if
// two strings are rotations
// of each other.
using System;

class GFG
{
    public static bool isRotation(string a,
                                  string b)
    {
        int n = a.Length;
        int m = b.Length;
        if (n != m)
            return false;

        // create lps[] that will
        // hold the longest prefix
        // suffix values for pattern
        int []lps = new int[n];

        // length of the previous
        // longest prefix suffix
        int len = 0;
        int i = 1;

        // lps[0] is always 0
        lps[0] = 0;

        // the loop calculates
        // lps[i] for i = 1 to n-1
        while (i < n)
        {
            if (a[i] == b[len])
            {
                lps[i] = ++len;
                ++i;
            }
            else
            {
                if (len == 0)
                {

```

```
        lps[i] = 0;
        ++i;
    }
    else
    {
        len = lps[len - 1];
    }
}

i = 0;

// match from that
// rotating point
for (int k = lps[n - 1]; k < m; ++k)
{
    if (b[k] != a[i++])
        return false;
}
return true;
}

// Driver code
public static void Main()
{
    string s1 = "ABACD";
    string s2 = "CDABA";

    Console.WriteLine(isRotation(s1, s2) ?
        "1" : "0");
}

// This code is contributed
// by anuj_67.
```

**Output:**

1

**Time Complexity :**  $O(n)$

**Auxiliary Space :**  $O(n)$

**Improved By :** [vt\\_m](#)

## **Source**

<https://www.geeksforgeeks.org/check-strings-rotations-not-set-2/>

## Chapter 15

# DFA for Strings not ending with “THE”

DFA for Strings not ending with "THE" - GeeksforGeeks

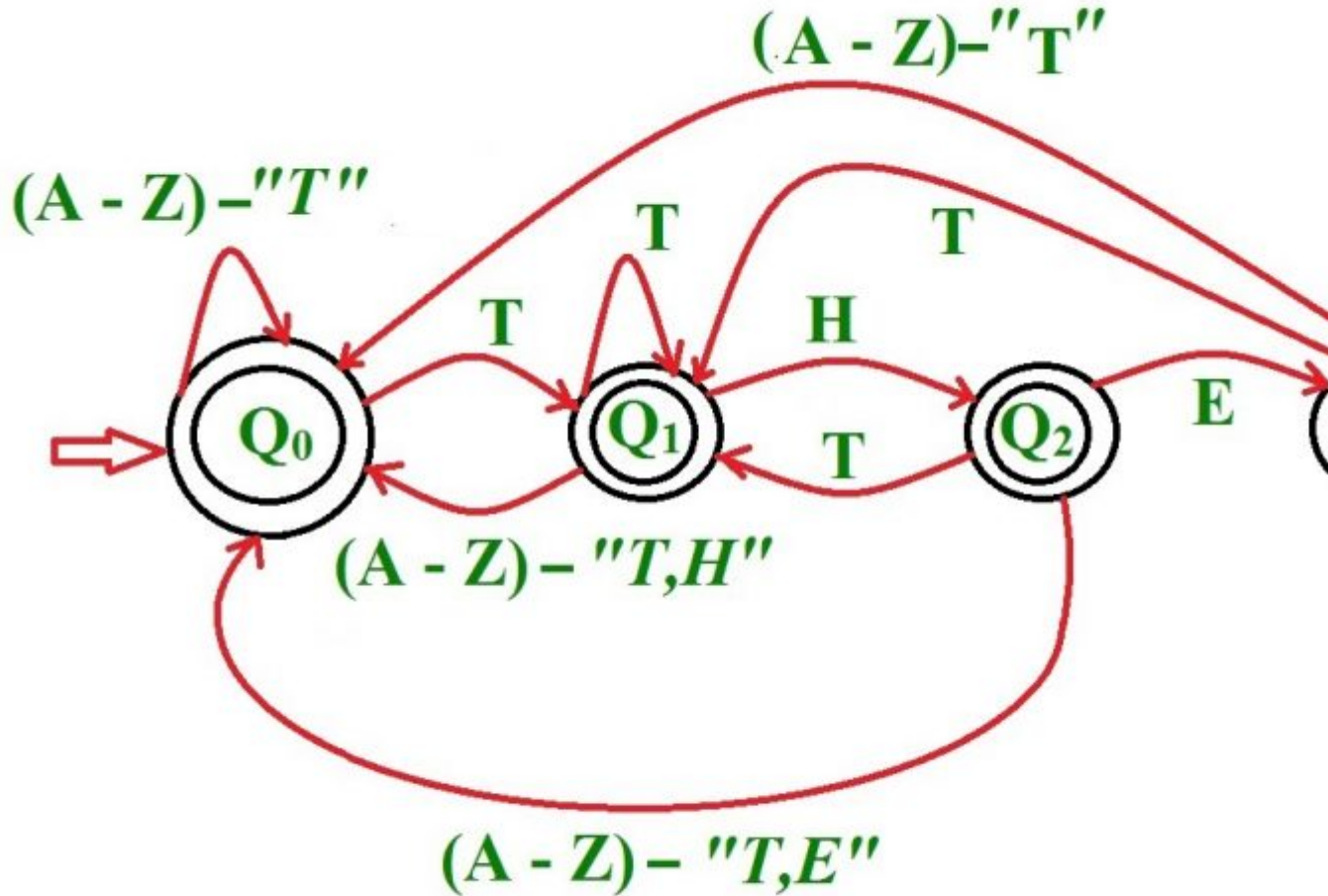
**Problem** – Accept Strings that not ending with substring “THE”. Check if a given string is ending with “the” or not. The different forms of “the” which are avoided in the end of the string are:

"THE", "ThE", "THe", "tHE", "thE", "The", "tHe" and "the"

All those strings that are ending with any of the above mentioned forms of “the” are not accepted.

**Deterministic finite automata (DFA) of strings that not ending with “THE”** –  
The initial and starting state in this dfa is  $Q_0$





**Approach used in the program –**

In this program, consider the 4 states to be 0, 1, 2 and 3. Now let us take a variable named DFA which will be initially 0. Whenever any transition takes place, it will update the value of DFA with the number associated with new state.

**Example :** If a transition occurs from state 0 to state 1 then the value of DFA will be updated to 1. If a transition occurs from state 2 to state 3 then the value of dfa will be updated to 3. In this way, apply this algorithm on entire string and if in the end, then reach state 0, 1 or 2 then our string will be accepted otherwise not.

Input : XYzabCthe

Output : NOT ACCEPTED

Input : Themaliktth

Output : ACCEPTED

C++

```
// CPP program to implement DFS that accepts
// all string that do not end with "THE"
#include <stdio.h>
#include <string.h>

// dfa tells the number associated
// with the present state
int dfa = 0;

// This function is for
// the starting state (zeroth) of DFA
void start(char c)
{
    // On receiving 'T' or 't' goto first state (1)
    if (c == 't' || c == 'T')
        dfa = 1;
}

// This function is for the first state of DFA
void state1(char c)
{
    // On receiving 'T' or 't' goto first state (1)
    if (c == 't' || c == 'T')
        dfa = 1;

    // On receiving 'H' or 'h' goto second state (2)
    else if (c == 'h' || c == 'H')
        dfa = 2;

    // else goto starting state (0)
    else
        dfa = 0;
}

// This function is for the second state of DFA
void state2(char c)
{
    // On receiving 'E' or 'e' goto third state (3)
    // else goto starting state (0)
    if (c == 'e' || c == 'E')
        dfa = 3;
    else
        dfa = 0;
}
```

```
// This function is for the third state of DFA
void state3(char c)
{
    // On receiving 'T' or 't' goto first state (1)
    // else goto starting state (0)
    if (c == 't' || c == 'T')
        dfa = 1;
    else
        dfa = 0;
}

bool isAccepted(char str[])
{
    // store length of string
    int len = strlen(str);

    for (int i=0; i < len; i++) {
        if (dfa == 0)
            start(str[i]);

        else if (dfa == 1)
            state1(str[i]);

        else if (dfa == 2)
            state2(str[i]);

        else
            state3(str[i]);
    }

    return (dfa != 3);
}

// driver code
int main()
{
    char str[] = "forTHEgeeks";
    if (isAccepted(str) == true)
        printf("ACCEPTED\n");
    else
        printf("NOT ACCEPTED\n");
    return 0;
}
```

## PHP

<?php

```
// PHP program to implement DFS
// that accepts all string that
// do not end with "THE"

// dfa tells the number associated
// with the present state
$dfa = 0;

// This function is for the
// starting state (zeroth) of DFA
function start($c)
{
    global $dfa;
    // On receiving 'T' or 't'
    // goto first state (1)
    if ($c == 't' || $c == 'T')
        $dfa = 1;
}

// This function is for
// the first state of DFA
function state1($c)
{
    global $dfa;
    // On receiving 'T' or 't'
    // goto first state (1)
    if ($c == 't' || $c == 'T')
        $dfa = 1;

    // On receiving 'H' or 'h'
    // goto second state (2)
    else if ($c == 'h' || $c == 'H')
        $dfa = 2;

    // else goto starting state (0)
    else
        $dfa = 0;
}

// This function is for
// the second state of DFA
function state2($c)
{
    global $dfa;
    // On receiving 'E' or 'e'
    // goto third state (3) else
    // goto starting state (0)
    if ($c == 'e' || $c == 'E')
```

```

        $dfa = 3;
    else
        $dfa = 0;
}

// This function is for
// the third state of DFA
function state3($c)
{
    global $dfa;
    // On receiving 'T' or 't'
    // goto first state (1) else
    // goto starting state (0)
    if ($c == 't' || $c == 'T')
        $dfa = 1;
    else
        $dfa = 0;
}

function isAccepted($str)
{
    global $dfa;
    // store length of string
    $len = strlen($str);

    for ($i=0; $i < $len; $i++)
    {
        if ($dfa == 0)
            start($str[$i]);

        else if ($dfa == 1)
            state1($str[$i]);

        else if ($dfa == 2)
            state2($str[$i]);

        else
            state3($str[$i]);
    }

    return ($dfa != 3);
}

// Driver Code
$str = "forTHEgeeks";
if (isAccepted($str) == true)
    echo "ACCEPTED\n";
else

```

```
    echo "NOT ACCEPTED\n";

// This code is contributed by m_kit
?>
```

**Output :**

ACCEPTED

The time complexity of this program is  $O(n)$

**Improved By :** [jit\\_t](#)

**Source**

<https://www.geeksforgeeks.org/dfa-for-strings-not-ending-with-the/>

## Chapter 16

# Check if a string is substring of another

Check if a string is substring of another - GeeksforGeeks

Given two strings s1 and s2, find if s1 is substring of s2. If yes, return index of first occurrence, else return -1.

**Examples :**

Input : s1 = "for", s2 = "geeksforgeeks"

Output : 5

String "for" is present as a substring of s2.

Input : s1 = "practice", s2 = "geeksforgeeks"

Output : -1.

A **simple solution** is to one by one check every index of s2. For every index, check if s1 is present.

**C++**

```
// CPP program to check if a string is
// substring of other.
#include <bits/stdc++.h>
using namespace std;

// Returns true if s2 is substring of s1
int isSubstring(string s1, string s2)
{
    int M = s1.length();
```

```
int N = s2.length();

/* A loop to slide pat[] one by one */
for (int i = 0; i <= N - M; i++) {
    int j;

    /* For current index i, check for pattern match */
    for (j = 0; j < M; j++)
        if (s2[i + j] != s1[j])
            break;

    if (j == M)
        return i;
}

return -1;
}

/* Driver program to test above function */
int main()
{
    string s1 = "for";
    string s2 = "geeksforgeeks";
    int res = isSubstring(s1, s2);
    if (res == -1)
        cout << "Not present";
    else
        cout << "Present at index " << res;
    return 0;
}
```

## Java

```
//Java program to check if a string is
//substring of other.
class GFG {

    // Returns true if s2 is substring of s1
    static int isSubstring(String s1, String s2)
    {
        int M = s1.length();
        int N = s2.length();

        /* A loop to slide pat[] one by one */
        for (int i = 0; i <= N - M; i++) {
            int j;

            /* For current index i, check for
```



```
        pattern match */
        for (j = 0; j < M; j++)
            if (s2.charAt(i + j) != s1.charAt(j))
                break;

        if (j == M)
            return i;
    }

    return -1;
}

/* Driver program to test above function */
public static void main(String args[])
{
    String s1 = "for";
    String s2 = "geeksforgeeks";

    int res = isSubstring(s1, s2);

    if (res == -1)
        System.out.println("Not present");
    else
        System.out.println("Present at index "
                           + res);
}
}
```

// This code is contributed by JaideepPyne.

### Python 3

```
# Python 3 program to check if
# a string is substring of other.

# Returns true if s2 is substring of s1
def isSubstring(s1, s2):
    M = len(s1)
    N = len(s2)

    # A loop to slide pat[] one by one
    for i in range(N - M + 1):

        # For current index i,
        # check for pattern match
        for j in range(M):
            if (s2[i + j] != s1[j]):
                break

        if j + 1 == M :
            return i
```

```
return -1

# Driver Code
if __name__ == "__main__":
    s1 = "for"
    s2 = "geeksforgeeks"
    res = isSubstring(s1, s2)
    if res == -1 :
        print("Not present")
    else:
        print("Present at index " + str(res))

# This code is contributed by Chitranayal

C#

//C# program to check if a string is
//substring of other.
using System;
class GFG {

    // Returns true if s2 is substring of s1
    static int isSubstring(string s1, string s2)
    {
        int M = s1.Length;
        int N = s2.Length;

        /* A loop to slide pat[] one by one */
        for (int i = 0; i <= N - M; i++) {
            int j;

            /* For current index i, check for
            pattern match */
            for (j = 0; j < M; j++)
                if (s2[i + j] != s1[j])
                    break;

            if (j == M)
                return i;
        }

        return -1;
    }

    /* Driver program to test above function */
    public static void Main()
    {
        string s1 = "for";
        string s2 = "geeksforgeeks";
```

```
        int res = isSubstring(s1, s2);

        if (res == -1)
            Console.WriteLine("Not present");
        else
            Console.WriteLine("Present at index "
                               + res);
    }
}

// This code is contributed by nitin mittal.
```

## PHP

```
<?php
// PHP program to check if a
// string is substring of other.

// Returns true if s2
// is substring of s1
function isSubstring($s1, $s2)
{
    $M = strlen($s1);
    $N = strlen($s2);

    // A loop to slide
    // pat[] one by one
    for ($i = 0; $i <= $N - $M; $i++)
    {
        $j = 0;

        // For current index i,
        // check for pattern match
        for (; $j < $M; $j++)
            if ($s2[$i + $j] != $s1[$j])
                break;

        if ($j == $M)
            return $i;
    }

    return -1;
}

// Driver Code
$s1 = "for";
$s2 = "geeksforgeeks";
```

```
$res = isSubstring($s1, $s2);  
if ($res == -1)  
    echo "Not present";  
else  
    echo "Present at index " . $res;  
  
// This code is contributed by mits  
?>
```

**Output:**

Present at index 5

**Time complexity :**  $O(m * n)$  where m and n are lengths of s1 and s2 respectively.

An **efficient solution** is to use a  $O(n)$  searching algorithm like [KMP algorithm](#), [Z algorithm](#), etc.

**Language implementations :**

- [Java Substring](#)
- [substr in C++](#)
- [Python find](#)

**Improved By :** [jaideeppyne1997](#), [nitin mittal](#), [Mithun Kumar](#), [ChitraNayal](#)

**Source**

<https://www.geeksforgeeks.org/check-string-substring-another/>

## Chapter 17

# Program to replace a word with asterisks in a sentence

Program to replace a word with asterisks in a sentence - GeeksforGeeks

For the given sentence as input, censor a specific word with asterisks ' \* '.

**Example :**

**Input :** word = "computer"

text = "GeeksforGeeks is a computer science portal for geeks. People who love computer and computer codes can contribute their valuables/ideas on computer codes/structures on here."

**Output :** GeeksforGeeks is a \*\*\*\*\* science portal for geeks. People who love \*\*\*\*\* and \*\*\*\*\* codes can contribute their valuables/ideas on \*\*\*\*\* codes/structures on here.

The idea is to first split given sentence into different words. Then traverse the word list. For every word in the word list, check if it matches with given word. If yes, then replace the word with stars in the list. Finally merge the words of list and print.

```
# Python Program to censor a word
# with asterisks in a sentence

# Function takes two parameter
def censor(text, word):

    # Break down sentence by ' ' spaces
    # and store each individual word in
    # a different list
    word_list = text.split()
```

```
# A new string to store the result
result = ''

# Creating the censor which is an asterisks
# "*" text of the length of censor word
stars = '*' * len(word)

# count variable to
# access our word_list
count = 0

# Iterating through our list
# of extracted words
index = 0;
for i in word_list:

    if i == word:

        # changing the censored word to
        # created asterisks censor
        word_list[index] = stars
        index += 1

# join the words
result =' '.join(word_list)

return result

# Driver code
if __name__ == '__main__':

    extract = "GeeksforGeeks is a computer science portal for geeks.\
        I am pursuing my major in computer science. "
    cen = "computer"
    print(censor(extract, cen))
```

#### Output :

```
GeeksforGeeks is a ***** science portal for geeks.
I am pursuing my major in ***** science.
```

#### Source

<https://www.geeksforgeeks.org/program-censor-word-asterisks-sentence/>

## Chapter 18

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

Dynamic Programming | Wildcard Pattern Matching | Linear Time and Constant Space -  
GeeksforGeeks

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 '\*'

'?' – matches any single character

'\*' – Matches any sequence of characters (including the empty sequence)

**Prerequisite :** [Dynamic Programming](#) | [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
```

Input =            **ba** aab **ab**  
Pattern = \*\*\*\*\* **ba** \*\*\*\*\* **ab**  
Output : true

                            No matching text  
Input =            **b**aaabab  
                            ~~~~~~  
Pattern =           **a** \*       **ab**  
Output : false

Input =            **ba** aab **ab**  
Pattern =           **ba** \*       **a?**  
Output : true

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.

We have discussed a solution [here](#) which has  $O(m \times n)$  time and  $O(m \times n)$  space complexity.

For applying the optimization, we will at first note the **BASE CASE** which involves :

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 | (STEP BY STEP)

Step – (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  $\text{index\_txt}$  be the marker to point at the character of text on which we encounter '\*' in pattern.

Let  $\text{index\_pat}$  be the marker to point at the position of '\*' in the pattern.

**NOTE : WE WILL TRAVERSE THE GIVEN STRING AND PATTERN USING A WHILE LOOP**

Step – (2) : At any instant if we observe that  $\text{txt}[i] == \text{pat}[j]$ , then we increment both  $i$  and  $j$  as no operation needs to be performed in this case.



Step – (3) : If we encounter  $\text{pat}[j] == '?'$ , then it resembles the case mentioned in step – (2) as '?' has the property to match with any single character.

Step – (4) : If we encounter  $\text{pat}[j] == '*'$ , then we update the value of  $\text{index\_txt}$  and  $\text{index\_pat}$  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).

Step – (5) : Now if  $\text{txt}[i] == \text{pat}[j]$ , and we have encountered a '\*' before, then it means that '\*' included the empty sequence, else if  $\text{txt}[i] != \text{pat}[j]$ , 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,  $j = \text{index\_pat} + 1$ ,  $i = \text{index\_txt} + 1$  (as '\*' can capture other characters as well),  $\text{index\_txt}++$  (as current character in text is matched).

Step – (6) : If step – (5) is not valid, that means  $\text{txt}[i] != \text{pat}[j]$ , also we have not encountered a '\*' that means it is not possible for the pattern to match the string. (return false).

Step – (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  $\text{pat}[j] == '*'$ , 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  $\text{txt}[i] == \text{pat}[j]$ , 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 -> '\*')

**EIGHTH 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 above optimized approach.**

```
// C++ program to implement wildcard
// pattern matching algorithm
#include <bits/stdc++.h>
using namespace std;

// Function that matches input text
// with given wildcard pattern
bool strmatch(char txt[], char pat[],
              int n, int m)
{
    // empty pattern can only
    // match with empty string.
    // Base Case :
    if (m == 0)
        return (n == 0);

    // step-1 :
    // initialize markers :
    int i = 0, j = 0, index_txt = -1,
        index_pat = -1;

    while (i < n) {

        // For step - (2, 5)
        if (txt[i] == pat[j]) {
            i++;
```

```
        j++;
    }

    // For step - (3)
    else if (j < m && pat[j] == '?') {
        i++;
        j++;
    }

    // For step - (4)
    else if (j < m && pat[j] == '*') {
        index_txt = i;
        index_pat = j;
        j++;
    }

    // For step - (5)
    else if (index_pat != -1) {
        j = index_pat + 1;
        i = index_txt + 1;
        index_txt++;
    }

    // For step - (6)
    else {
        return false;
    }
}

// For step - (7)
while (j < m && pat[j] == '*') {
    j++;
}

// Final Check
if (j == m) {
    return true;
}

return false;
}

// Driver code
int main()
{
    char str[] = "baaabab";
    char pattern[] = "*****ba*****ab";
    // char pattern[] = "ba*****ab";
}
```

```
// char pattern[] = "ba*ab";
// char pattern[] = "a*ab";

if (strmatch(str, pattern,
            strlen(str), strlen(pattern)))
    cout << "Yes" << endl;
else
    cout << "No" << endl;

char pattern2[] = "a*****ab";
if (strmatch(str, pattern2,
            strlen(str), strlen(pattern2)))
    cout << "Yes" << endl;
else
    cout << "No" << endl;

return 0;
}
```

#### Output:

Yes  
No

Time complexity of above solution is  $O(m)$ . Auxiliary space used is  $O(1)$ .

#### Source

<https://www.geeksforgeeks.org/dynamic-programming-wildcard-pattern-matching-linear-time-constant-space/>

## Chapter 19

# Pattern Searching using C++ library

Pattern Searching using C++ library - GeeksforGeeks

Given a text `txt[0..n-1]` and a pattern `pat[0..m-1]`, write a function that prints all occurrences of `pat[]` in `txt[]`. You may assume that  $n > m$ .

Examples:

```
Input : txt[] = "geeks for geeks"
        pat[] = "geeks"
Output : Pattern found at index 0
        Pattern found at index 10
```

```
Input : txt[] = "aaaa"
        pat[] = "aa"
Output : Pattern found at index 0
        Pattern found at index 1
        Pattern found at index 2
```

The idea is to use [find\(\) in C++ string class](#).

```
// CPP program to print all occurrences of a pattern
// in a text
#include <bits/stdc++.h>
using namespace std;

void printOccurrences(string txt, string pat)
{
    int found = txt.find(pat);
```

```
        while (found != string::npos) {
            cout << "Pattern found at index " << found << endl;
            found = txt.find(pat, found + 1);
        }
    }

int main()
{
    string txt = "aaaa", pat = "aa";
    printOccurrences(txt, pat);
    return 0;
}
```

#### Output:

```
Pattern found at index 0
Pattern found at index 1
Pattern found at index 2
```

#### Source

<https://www.geeksforgeeks.org/pattern-searching-using-c-library/>

## Chapter 20

# Longest prefix which is also suffix

Longest prefix which is also suffix - GeeksforGeeks

Given a string s, find length of the longest prefix which is also 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

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

Below is a solution of above approach!

C++

```
// CPP program to find length of the longest
// prefix which is also suffix
#include <bits/stdc++.h>
using namespace std;
```



```
// Function to find largest prefix which is also a suffix
int largest_prefix_suffix(const std::string &str) {

    int n = str.length();

    if(n < 2) {
        return 0;
    }

    int len = 0;
    int i = n/2;

    while(i < n) {
        if(str[i] == str[len]) {
            ++len;
            ++i;
        } else {
            if(len == 0) { // no prefix
                ++i;
            } else { // search for shorter prefixes
                --len;
            }
        }
    }

    return len;
}

// Driver code
int main() {

    string s = "blablabla";

    cout << largest_prefix_suffix(s);

    return 0;
}
```

## Java

```
// Java program to find length of the longest
// prefix which is also suffix
class GFG {

    static int longestPrefixSuffix(String s)
    {
```

```
int n = s.length();

if(n < 2) {
    return 0;
}

int len = 0;
int i = n/2;

while(i < n) {
    if(s.charAt(i) == s.charAt(len)) {
        ++len;
        ++i;
    }
    else
    {
        if(len == 0) { // no prefix
            ++i;
        }
        else
        {
            // search for shorter prefixes
            --len;
        }
    }
}

return len;
}

// Driver code
public static void main (String[] args)
{
    String s = "blablabla";
    System.out.println(longestPrefixSuffix(s));
}

// This code is contributed by Anant Agarwal.
```

### Python3

```
# 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

s = "blablabla"
print(longestPrefixSuffix(s))

# This code is contributed by Nikita Tiwari.
```

## C#

```
// C# program to find length of the longest
// prefix which is also suffix
using System;

class GFG {

    static int longestPrefixSuffix(String s)
    {
        int n = s.Length;

        if(n < 2)
            return 0;

        int len = 0;
        int i = n / 2;

        while(i < n) {
            if(s[i] == s[len]) {
                ++len;
                ++i;
            }
            else {
                if(len == 0) {
```

```
        // no prefix
        ++i;
    }
    else {

        // search for shorter prefixes
        --len;
    }
}

return len;
}

// Driver code
public static void Main ()
{
    String s = "blablabla";

    Console.WriteLine(longestPrefixSuffix(s));
}

// This code is contributed by vt_m.
```

**Output:**

3

**Efficient Solution :** The idea is to use preprocessing algorithm of [KMP search](#). In the preprocessing algorithm, we build lps array which stores following values.

lps[i] = the longest proper prefix of pat[0..i]  
which is also a suffix of pat[0..i].

**C++**

```
// Efficient CPP program to find length of
// the longest prefix which is also suffix
#include<bits/stdc++.h>
using namespace std;

// 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/
int longestPrefixSuffix(string s)
{
    int n = s.length();

    int lps[n];
    lps[0] = 0; // lps[0] is always 0

    // length of the previous
    // longest prefix suffix
    int len = 0;

    // the loop calculates lps[i]
    // for i = 1 to n-1
    int i = 1;
    while (i < n)
    {
        if (s[i] == s[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider
            // the example. AAACAAAA
            // and i = 7. The idea is
            // similar to search step.
            if (len != 0)
            {
                len = lps[len-1];

                // Also, note that we do
                // not increment i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }

    int res = lps[n-1];

    // Since we are looking for
```

```
    // non overlapping parts.
    return (res > n/2)? n/2 : res;
}

// Driver program to test above function
int main()
{
    string s = "abcbab";
    cout << longestPrefixSuffix(s);
    return 0;
}
```

### Java

```
// Efficient Java program to find length of
// the longest prefix which is also suffix

class GFG
{
    // 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/
    static int longestPrefixSuffix(String s)
    {
        int n = s.length();

        int lps[] = new int[n];

        // lps[0] is always 0
        lps[0] = 0;

        // length of the previous
        // longest prefix suffix
        int len = 0;

        // the loop calculates lps[i]
        // for i = 1 to n-1
        int i = 1;
        while (i < n)
        {
            if (s.charAt(i) == s.charAt(len))
            {
                len++;
                lps[i] = len;
                i++;
            }
        }
    }
}
```

```
    }

    // (pat[i] != pat[len])
    else
    {
        // This is tricky. Consider
        // the example. AAACAAAA
        // and i = 7. The idea is
        // similar to search step.
        if (len != 0)
        {
            len = lps[len-1];

            // Also, note that we do
            // not increment i here
        }

        // if (len == 0)
        else
        {
            lps[i] = 0;
            i++;
        }
    }
}

int res = lps[n-1];

// Since we are looking for
// non overlapping parts.
return (res > n/2)? n/2 : res;
}

// Driver program
public static void main (String[] args)
{
    String s = "abcaab";
    System.out.println(longestPrefixSuffix(s));
}

// This code is contributed by Anant Agarwal.
```

### Python3

```
# 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 = 1
    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.
    if(res > n/2) :
```



```
        return n//2
    else :
        return res

# Driver program to test above function
s = "abcbab"
print(longestPrefixSuffix(s))

# This code is contributed
# by Nikita Tiwari.

C#

// Efficient C# program to find length of
// the longest prefix which is also suffix
using System;

class GFG {

    // 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/
    static int longestPrefixSuffix(string s)
    {
        int n = s.Length;

        int []lps = new int[n];

        // lps[0] is always 0
        lps[0] = 0;

        // length of the previous
        // longest prefix suffix
        int len = 0;

        // the loop calculates lps[i]
        // for i = 1 to n-1
        int i = 1;
        while (i < n)
        {
            if (s[i] == s[len])
            {
                len++;
            }
        }
    }
}
```

```
        lps[i] = len;
        i++;
    }

    // (pat[i] != pat[len])
    else
    {

        // This is tricky. Consider
        // the example. AAACAAAA
        // and i = 7. The idea is
        // similar to search step.
        if (len != 0)
        {
            len = lps[len-1];

            // Also, note that we do
            // not increment i here
        }

        // if (len == 0)
        else
        {
            lps[i] = 0;
            i++;
        }
    }
}

int res = lps[n-1];

// Since we are looking for
// non overlapping parts.
return (res > n/2) ? n/2 : res;
}

// Driver program
public static void Main ()
{
    string s = "abcbab";

    Console.WriteLine(longestPrefixSuffix(s));
}

// This code is contributed by vt_m.
```

## PHP

```
<?php
// Efficient PHP 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/
function longestPrefixSuffix($s)
{
    $n = strlen($s);

    $lps[$n] = NULL;

    // lps[0] is always 0
    $lps[0] = 0;

    // length of the previous
    // longest prefix suffix
    $len = 0;

    // the loop calculates lps[i]
    // for i = 1 to n-1
    $i = 1;
    while ($i < $n)
    {
        if ($s[$i] == $s[$len])
        {
            $len++;
            $lps[$i] = $len;
            $i++;
        }

        // (pat[i] != pat[len])
        else
        {
            // This is tricky. Consider
            // the example. AAACAAAA
            // and i = 7. The idea is
            // similar to search step.
            if ($len != 0)
            {
                $len = $lps[$len-1];

                // Also, note that we do
                // not increment i here
            }
        }
    }
}
```

```
        }

        // if (len == 0)
        else
        {
            $lps[$i] = 0;
            $i++;
        }
    }
}

$res = $lps[$n-1];

// Since we are looking for
// non overlapping parts.
return ($res > $n/2)? $n/2 : $res;
}

// Driver Code
$s = "abcbab";
echo longestPrefixSuffix($s);

// This code is contributed by nitin mittal
?>
```

**Output:**

2

Please refer computeLPSArray() of [KMP search](#) for explanation.

Time Complexity :  $O(n)$

Auxiliary Space :  $O(n)$

Improved By : [nitin mittal](#)

**Source**

<https://www.geeksforgeeks.org/longest-prefix-also-suffix/>

## Chapter 21

# Splitting a Numeric String

Splitting a Numeric String - GeeksforGeeks

Given a numeric string (length  $\leq 32$ ), split it into two or more integers( if possible), such that

1) Difference between current and previous number is 1.

2) No number contains leading zeroes

If it is possible to separate a given numeric string then print “**Possible**” followed by the first number of the increasing sequence, else print “**Not Possible**”.

**Examples:**

Input : 1234

Output : Possible 1

Explanation: String can be split as "1", "2", "3", "4"

Input : 99100

Output :Possible 99

Explanation: String can be split as "99", "100"

Input : 101103

Output : Not Possible

Explanation: It is not possible to split this string under given constraint.

**Approach :** The idea is to take a substring from index 0 to any index i (i starting from 1) of the numeric string and convert it to long data type. Add 1 to it and convert the increased number back to string. Check if the next occurring substring is equal to the increased one. If yes, then carry on the procedure else increase the value of i and repeat the steps.

**Java**

```
// Java program to split a numeric
// string in an Increasing
// sequence if possible
import java.io.*;
import java.util.*;

class GFG {

    // Function accepts a string and
    // checks if string can be split.
    public static void split(String str)
    {
        int len = str.length();

        // if there is only 1 number
        // in the string then
        // it is not possible to split it
        if (len == 1) {
            System.out.println("Not Possible");
            return;
        }

        String s1 = "", s2 = "";
        long num1, num2;

        for (int i = 0; i <= len / 2; i++) {

            int flag = 0;

            // storing the substring from
            // 0 to i+1 to form initial
            // number of the increasing sequence
            s1 = str.substring(0, i + 1);
            num1 = Long.parseLong((s1));
            num2 = num1 + 1;

            // convert string to integer
            // and add 1 and again convert
            // back to string s2
            s2 = Long.toString(num2);
            int k = i + 1;
            while (flag == 0) {
                int l = s2.length();

                // if s2 is not a substring
                // of number than not possible
                if (k + l > len) {
                    flag = 1;
                }
            }
        }
    }
}
```

```
        break;
    }

    // if s2 is the next substring
    // of the numeric string
    if ((str.substring(k, k + 1).equals(s2))) {
        flag = 0;

        // Incearse num2 by 1 i.e the
        // next number to be looked for
        num2++;
        k = k + 1;

        // check if string is fully
        // traversed then break
        if (k == len)
            break;
        s2 = Long.toString(num2);
        l = s2.length();
        if (k + 1 > len) {

            // If next string doesnot occurs
            // in a given numeric string
            // then it is not possible
            flag = 1;
            break;
        }
    }

    else
        flag = 1;
}

// if the string was fully traversed
// and conditions were satisfied
if (flag == 0) {
    System.out.println("Possible"
        + " " + s1);
    break;
}

// if conditions failed to hold
else if (flag == 1 && i > len / 2 - 1) {
    System.out.println("Not Possible");
    break;
}
}
}
```

```
// Driver Code
public static void main(String args[])
{
    Scanner in = new Scanner(System.in);
    String str = "99100";

    // Call the split function
    // for splitting the string
    split(str);
}
}
```

**C#**

```
// C# program to split a numeric
// string in an Increasing
// sequence if possible
using System;

class GFG
{
    // Function accepts a
    // string and checks if
    // string can be split.
    static void split(string str)
    {
        int len = str.Length;

        // if there is only 1
        // number in the string
        // then it is not possible
        // to split it
        if (len == 1)
        {
            Console.WriteLine("Not Possible");
            return;
        }

        string s1 = "", s2 = "";
        long num1, num2;

        for (int i = 0; i < len / 2; i++)
        {
            int flag = 0;
```



```
// storing the substring
// from 0 to i+1 to form
// initial number of the
// increasing sequence
s1 = str.Substring(0, i + 1);
num1 = Convert.ToInt64(s1);
num2 = num1 + 1;

// convert string to integer
// and add 1 and again convert
// back to string s2
s2 = num2.ToString();
int k = i + 1;
while (flag == 0)
{
    int l = s2.Length;

    // if s2 is not a substring
    // of number than not possible
    if (k + l > len)
    {
        flag = 1;
        break;
    }

    // if s2 is the next
    // substring of the
    // numeric string
    if ((str.Substring(k, l).Equals(s2)))
    {
        flag = 0;

        // Increase num2 by 1 i.e
        // the next number to be
        // looked for
        num2++;
        k = k + l;

        // check if string is fully
        // traversed then break
        if (k == len)
            break;
        s2 = num2.ToString();
        l = s2.Length;
        if (k + l > len)
        {

            // If next string doesnot
```

```
        // occurs in a given numeric
        // string then it is not
        // possible
        flag = 1;
        break;
    }
}

else
    flag = 1;
}

// if the string was fully
// traversed and conditions
// were satisfied
if (flag == 0)
{
    Console.WriteLine("Possible" +
        " " + s1);

    break;
}

// if conditions
// failed to hold
else if (flag == 1 &&
        i > len / 2 - 1)
{
    Console.WriteLine("Not Possible");
    break;
}
}

}

// Driver Code
static void Main()
{
    string str = "99100";

    // Call the split function
    // for splitting the string
    split(str);
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

**Output:**

Possible 99

Improved By : [manishshaw1](#)

### Source

<https://www.geeksforgeeks.org/splitting-numeric-string/>

## Chapter 22

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

Count of number of given string in 2D character array - GeeksforGeeks

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

We have discussed simpler problem to [find if a word exists or not in a matrix](#).

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

**Algorithm :**

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

**C**

```
// C code for finding count
// of string in a given 2D
// character array.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(*a))

// utility function to search
// complete string from any
// given index of 2d char array
int internalSearch(char *needle, int row,
                  int col, char **hay,
                  int row_max, int col_max)
{
    int found = 0;

    if (row >= 0 && row <= row_max && col >= 0 &&
        col <= col_max && *needle == hay[row][col])
    {
        char match = *needle++;

        hay[row][col] = 0;

        if (*needle == 0) {
            found = 1;
        }
    }
}
```

```
    } else {

        // through Backtrack searching
        // in every directions
        found += internalSearch(needle, row,
                                col+1, hay,
                                row_max, col_max);
        found += internalSearch(needle, row, col-1,
                                hay, row_max, col_max);
        found += internalSearch(needle, row+1, col,
                                hay, row_max, col_max);
        found += internalSearch(needle, row-1, col,
                                hay, row_max, col_max);
    }

    hay[row][col] = match;
}

return found;
}

// Function to search the string in 2d array
int searchString(char *needle, int row, int col,
                char **str, int row_count, int col_count)
{
    int found = 0;
    int r, c;

    for (r = 0; r < row_count; ++r) {
        for (c = 0; c < col_count; ++c) {
            found += internalSearch(needle, r, c, str,
                                    row_count - 1, col_count - 1);
        }
    }

    return found;
}

// Driver code
int main(void){

    char needle[] = "MAGIC";
    char *input[] = {
        "BBABBM",
        "CBMBBA",
        "IBABBG",
        "GOZBBI",
        "ABBBBC",
    }
```

```
        "MCIGAM"
    };
    char *str[ARRAY_SIZE(input)];
    int i;
    for (i = 0; i < ARRAY_SIZE(input); ++i) {
        str[i] = malloc(strlen(input[i]));
        strcpy(str[i], input[i]);
    }

    printf("count: %d\n", searchString(needle, 0, 0,
        str, ARRAY_SIZE(str), strlen(str[0])));

    return 0;
}
```

Output:

3

## Source

<https://www.geeksforgeeks.org/find-count-number-given-string-present-2d-character-array/>

## Chapter 23

# Find minimum shift for longest common prefix

Find minimum shift for longest common prefix - GeeksforGeeks

You are given two string str1 and str2 of same length. In a single shift you can rotate one string (str2) by 1 element such that its 1st element becomes the last and second one becomes the first like “abcd” will change to “bcda” after one shift operation. You have to find the minimum shift operation required to get common prefix of maximum length from str1 and str2.

Examples:

```
Input : str1[] = "geeks",  
        str2 = "dgeek"  
Output : Shift = 1,  
        Prefix = geek
```

```
Input : str1[] = "practicegeeks",  
        str2 = "coderpractice"  
Output : Shift = 5  
        Prefix = practice
```

**Naive Approach :** Shift second string one by one and keep track the length of longest prefix for each shift, there are total of  $n$  shifts and for each shift finding the length of common prefix will take  $O(n)$  time. Hence, overall time complexity for this approach is  $O(n^2)$ .

**Better Approach :** If we will add second string at the end of itself that is **str2 = str2 + str2** then there is no need of finding prefix for each shift separately. Now, after adding str2 to itself we have to only find the longest prefix of str1 present in str2 and the starting position of that prefix in str2 will give us the actual number of shift required. For finding longest prefix we can use [KMP pattern search algorithm](#).

So, in this way our time-complexity will reduce to  $O(n)$  only.



```
// CPP program to find longest common prefix
// after rotation of second string.
#include <bits/stdc++.h>
using namespace std;

// function for KMP search
void KMP(int m, int n, string str2, string str1)
{
    int pos = 0, len = 0;

    // preprocessing of longest proper prefix
    int p[m + 1];
    int k = 0;
    p[1] = 0;

    for (int i = 2; i <= n; i++) {
        while (k > 0 && str1[k] != str1[i - 1])
            k = p[k];
        if (str1[k] == str1[i - 1])
            ++k;
        p[i] = k;
    }

    // find out the longest prefix and position
    for (int j = 0, i = 0; i < m; i++) {
        while (j > 0 && str1[j] != str2[i])
            j = p[j];
        if (str1[j] == str2[i])
            j++;

        // for new position with longer prefix in str2
        // update pos and len
        if (j > len) {
            len = j;
            pos = i - j + 1;
        }
    }

    // print result
    cout << "Shift = " << pos << endl;
    cout << "Prefix = " << str1.substr(0, len);
}

// driver function
int main()
{
    string str1 = "geeksforgeeks";
    string str2 = "forgeeksgeeks";
```

```
    int n = str1.size();  
    str2 = str2 + str2;  
    KMP(2 * n, n, str2, str1);  
    return 0;  
}
```

Output:

```
Shift = 8  
Prefix = geeksforgeeks
```

### Source

<https://www.geeksforgeeks.org/find-minimum-shift-longest-common-prefix/>

## Chapter 24

# Frequency of a substring in a string

Frequency of a substring in a string - GeeksforGeeks

Given a input string and a substring. Find the frequency of occurrences of substring in given string.

Examples:

```
Input : man (pattern)
        dhimanman (string)
Output : 2
```

```
Input : nn (pattern)
        Banana (String)
Output : 0
```

```
Input : man (pattern)
        dhimanman (string)
Output : 2
```

```
Input : aa (pattern)
        aaaaaa (String)
Output : 4
```

A **simple solution** is to match characters one by one. And whenever we see a complete match, we increment count. Below is simple solution based on [Naive pattern searching](#).

```
// Simple C++ program to count occurrences
// of pat in txt.
```

```
#include<bits/stdc++.h>
using namespace std;

int countFreq(string &pat, string &txt)
{
    int M = pat.length();
    int N = txt.length();
    int res = 0;

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        /* For current index i, check for
           pattern match */
        int j;
        for (j = 0; j < M; j++)
            if (txt[i+j] != pat[j])
                break;

        // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        if (j == M)
        {
            res++;
            j = 0;
        }
    }
    return res;
}

/* Driver program to test above function */
int main()
{
    string txt = "dhimanman";
    string pat = "man";
    cout << countFreq(pat, txt);
    return 0;
}
```

Output :

2

Time Complexity :  $O(M * N)$

An **efficient solution** is to use [KMP algorithm](#).

```
// Java program to count occurrences of pattern
```

```
// in a text.
class KMP_String_Matching
{
    int KMPSearch(String pat, String txt)
    {
        int M = pat.length();
        int N = txt.length();

        // create lps[] that will hold the longest
        // prefix suffix values for pattern
        int lps[] = new int[M];
        int j = 0; // index for pat[]

        // Preprocess the pattern (calculate lps[]
        // array)
        computeLPSArray(pat,M,lps);

        int i = 0; // index for txt[]
        int res = 0;
        int next_i = 0;

        while (i < N)
        {
            if (pat.charAt(j) == txt.charAt(i))
            {
                j++;
                i++;
            }
            if (j == M)
            {
                // When we find pattern first time,
                // we iterate again to check if there
                // exists more pattern
                j = lps[j-1];
                res++;

                // We start i to check for more than once
                // appearance of pattern, we will reset i
                // to previous start+1
                if (lps[j] != 0)
                    i = ++next_i;
                j = 0;
            }

            // mismatch after j matches
            else if (i < N && pat.charAt(j) != txt.charAt(i))
            {
                // Do not match lps[0..lps[j-1]] characters,
```

```
        // they will match anyway
        if (j != 0)
            j = lps[j-1];
        else
            i = i+1;
    }
}
return res;
}

void computeLPSArray(String pat, int M, int lps[])
{
    // length of the previous longest prefix suffix
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
        if (pat.charAt(i) == pat.charAt(len))
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0)
            {
                len = lps[len-1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = len;
                i++;
            }
        }
    }
}

// Driver program to test above function
```

```
public static void main(String args[])
{
    String txt = "geeksforgeeks";
    String pat = "eeks";
    int ans = new KMP_String_Matching().KMPSearch(pat,txt);
    System.out.println(ans);
}
```

Output:

2

Time Complexity :  $O(M + N)$

### Source

<https://www.geeksforgeeks.org/frequency-substring-string/>

## Chapter 25

# Count of occurrences of a “1(0+)1” pattern in a string

Count of occurrences of a “1(0+)1” pattern in a string - GeeksforGeeks

Given an alphanumeric string, find the number of times a pattern 1(0+)1 occurs in the given string. Here, (0+) signifies the presence of non empty sequence of consecutive 0's.

Examples:

Input : 1001010001

Output : 3

First sequence is in between 0th and 3rd index.

Second sequence is in between 3rd and 5th index.

Third sequence is in between 5th and 9th index.

So total number of sequences comes out to be 3.

Input : 1001ab010abc01001

Output : 2

First sequence is in between 0th and 3rd index.

Second valid sequence is in between 13th and 16th index. So total number of sequences comes out to be 2.

The idea to solve this problem is to first find a ‘1’ and keep moving forward in the string and check as mentioned below:

1. If any character other than ‘0’ and ‘1’ is obtained then it means pattern is not valid. So we go on in the search of next ‘1’ from this index and repeat these steps again.
2. If a ‘1’ is seen, then check for the presence of ‘0’ at previous position to check the validity of sequence.



Below is the implementation of above idea:

C++

```
// C++ program to calculate number of times
// the pattern occurred in given string
#include<iostream>
using namespace std;

// Returns count of occurrences of "1(0+)1"
// int str.
int countPattern(string str)
{
    int len = str.size();
    bool oneSeen = 0;

    int count = 0; // Initialize result
    for (int i = 0; i < len ; i++)
    {
        // if 1 encountered for first time
        // set oneSeen to 1
        if (str[i] == '1' && oneSeen == 0)
            oneSeen = 1;

        // Check if there is any other character
        // other than '0' or '1'. If so then set
        // oneSeen to 0 to search again for new
        // pattern
        if (str[i] != '0' && str[i] != '1')
            oneSeen = 0;

        // Check if encountered '1' forms a valid
        // pattern as specified
        if (str[i] == '1' && oneSeen == 1)
            if (str[i - 1] == '0')
                count++;
    }

    return count;
}

// Driver program to test above function
int main()
{
    string str = "100001abc101";
    cout << countPattern(str);
    return 0;
}
```

## Java

```
//Java program to calculate number of times
//the pattern occurred in given string
public class GFG
{
    // Returns count of occurrences of "1(0+)1"
    // int str.
    static int countPattern(String str)
    {
        int len = str.length();
        boolean oneSeen = false;

        int count = 0; // Initialize result
        for(int i = 0; i < len ; i++)
        {
            char getChar = str.charAt(i);

            // if 1 encountered for first time
            // set oneSeen to 1
            if(getChar == '1' && oneSeen == false)
                oneSeen = true;

            // Check if there is any other character
            // other than '0' or '1'. If so then set
            // oneSeen to 0 to search again for new
            // pattern
            else if(getChar != '0' && str.charAt(i) != '1')
                oneSeen = false;

            // Check if encountered '1' forms a valid
            // pattern as specified
            else if (getChar == '1' && oneSeen == true){
                if (str.charAt(i - 1) == '0')
                    count++;
            }
        }
        return count;
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        String str = "100001abc101";
        System.out.println(countPattern(str));
    }
}
```

// This code is contributed by Sumit Ghosh

### Python

```
# Python program to calculate number of times
# the pattern occurred in given string

# Returns count of occurrences of "1(0+)1"
def countPattern(s):
    length = len(s)
    oneSeen = False

    count = 0    # Initialize result
    for i in range(length):

        # if 1 encountered for first time
        # set oneSeen to 1
        if (s[i] == '1' and oneSeen == 0):
            oneSeen = True

        # Check if there is any other character
        # other than '0' or '1'. If so then set
        # oneSeen to 0 to search again for new
        # pattern
        if (s[i] != '0' and s[i] != '1'):
            oneSeen = False

        # Check if encountered '1' forms a valid
        # pattern as specified
        if (s[i] == '1' and oneSeen):
            if (s[i - 1] == '0'):
                count += 1

    return count

# Driver code
s = "100001abc101"
print countPattern(s)

# This code is contributed by Sachin Bisht
```

Output:

2

**Time Complexity:**  $O(N)$ , where  $N$  is the length of input string.

## **Source**

<https://www.geeksforgeeks.org/count-of-occurrences-of-a-101-pattern-in-a-string/>

## Chapter 26

# Find all the patterns of “1(0+)1” in a given string | SET 2(Regular Expression Approach)

Find all the patterns of "1(0+)1" in a given string | SET 2(Regular Expression Approach)  
- GeeksforGeeks

In [Set 1](#), we have discussed general approach for counting the patterns of the form 1(0+)1 where (0+) represents any non-empty consecutive sequence of 0's. In this post, we will discuss [regular expression](#) approach to count the same.

### Examples:

Input : 1101001  
Output : 2

Input : 100001abc101  
Output : 2

Below is one of the regular expression for above pattern

10+1

Hence, whenever we found a match, we increase counter for counting the pattern. As last character of a match will always '1', we have to again start searching from that index.

```
//Java program to count the patterns
// of the form 1(0+)1 using Regex

import java.util.regex.Matcher;
import java.util.regex.Pattern;

class GFG
{
    static int patternCount(String str)
    {
        // regular expression for the pattern
        String regex = "10+1";

        // compiling regex
        Pattern p = Pattern.compile(regex);

        // Matcher object
        Matcher m = p.matcher(str);

        // counter
        int counter = 0;

        // whenever match found
        // increment counter
        while(m.find())
        {
            // As last character of current match
            // is always one, starting match from that index
            m.region(m.end()-1, str.length());

            counter++;
        }

        return counter;
    }

    // Driver Method
    public static void main (String[] args)
    {
        String str = "1001ab010abc01001";
        System.out.println(patternCount(str));
    }
}
```

Output:

**Related Articles :**

- [Regular Expression Java](#)
- [Quantifiers](#)
- [Extracting each word from a String using Regex](#)
- [Check if a given string is a valid number \(Integer or Floating Point\)](#)
- [Print first letter of each word in a string using regex](#)

**Source**

<https://www.geeksforgeeks.org/find-patterns-101-given-string-set-2regular-expression-approach/>

## Chapter 27

# Find all the patterns of “1(0+)1” in a given string | SET 1(General Approach)

Find all the patterns of "1(0+)1" in a given string | SET 1(General Approach) - Geeks-forGeeks

A string contains patterns of the form 1(0+)1 where (0+) represents any non-empty consecutive sequence of 0's. Count all such patterns. The patterns are allowed to overlap.

**Note :** It contains digits and lowercase characters only. The string is not necessarily a binary. 100201 is not a valid pattern.

One approach to solve the problem is discussed here, other using Regular expressions is given in [Set 2](#)

**Examples:**

Input : 1101001

Output : 2

Input : 100001abc101

Output : 2

Let size of input string be n.

1. Iterate through index '0' to 'n-1'.
2. If we encounter a '1', we iterate till the elements are '0'.
3. After the stream of zeros ends, we check whether we encounter a '1' or not.
4. Keep on doing this till we reach the end of string.

Below is the implementation of the above method.

C++



```
/* Code to count 1(0+)1 patterns in a string */
#include <bits/stdc++.h>
using namespace std;

/* Function to count patterns */
int patternCount(string str)
{
    /* Variable to store the last character*/
    char last = str[0];

    int i = 1, counter = 0;
    while (i < str.size())
    {
        /* We found 0 and last character was '1',
        state change*/
        if (str[i] == '0' && last == '1')
        {
            while (str[i] == '0')
                i++;

            /* After the stream of 0's, we got a '1',
            counter incremented*/
            if (str[i] == '1')
                counter++;
        }

        /* Last character stored */
        last = str[i];
        i++;
    }

    return counter;
}

/* Driver Code */
int main()
{
    string str = "1001ab010abc01001";
    cout << patternCount(str) << endl;
    return 0;
}
```

## Java

```
// Java Code to count 1(0+)1
// patterns in a string
import java.io.*;
```

```
class GFG
{
    // Function to count patterns
    static int patternCount(String str)
    {
        /* Variable to store the last character*/
        char last = str.charAt(0);

        int i = 1, counter = 0;
        while (i < str.length())
        {
            /* We found 0 and last character was '1',
            state change*/
            if (str.charAt(i) == '0' && last == '1')
            {
                while (str.charAt(i) == '0')
                    i++;

                // After the stream of 0's, we
                // got a '1', counter incremented
                if (str.charAt(i) == '1')
                    counter++;
            }

            /* Last character stored */
            last = str.charAt(i);
            i++;
        }

        return counter;
    }

    // Driver Code
    public static void main (String[] args)
    {
        String str = "1001ab010abc01001";
        System.out.println(patternCount(str));
    }
}

// This code is contributed by vt_m.
```

### Python3

```
# Python3 code to count 1(0+)1 patterns in a

# Function to count patterns
```

```
def patternCount(str):

    # Variable to store the last character
    last = str[0]

    i = 1; counter = 0
    while (i < len(str)):

        # We found 0 and last character was '1',
        # state change
        if (str[i] == '0' and last == '1'):
            while (str[i] == '0'):
                i += 1

            # After the stream of 0's, we got a '1',
            # counter incremented
            if (str[i] == '1'):
                counter += 1

        # Last character stored
        last = str[i]
        i += 1

    return counter

# Driver Code
str = "1001ab010abc01001"
ans = patternCount(str)
print (ans)

# This code is contributed by saloni1297
```

C#

```
// C# Code to count 1(0 + )1
// patterns in a string
using System;

class GFG
{
    // Function to count patterns
    static int patternCount(String str)
    {
        // Variable to store the
        // last character
        char last = str[0];
```

```
int i = 1, counter = 0;
while (i < str.Length)
{
    // We found 0 and last
    // character was '1',
    // state change
    if (str[i] == '0' && last == '1')
    {
        while (str[i] == '0')
            i++;

        // After the stream of 0's, we
        // got a '1', counter incremented
        if (str[i] == '1')
            counter++;
    }

    // Last character stored
    last = str[i];
    i++;
}

return counter;
}

// Driver Code
public static void Main ()
{
    String str = "1001ab010abc01001";
    Console.Write(patternCount(str));
}

// This code is contributed by nitin mittal
```

## PHP

```
<?php
// PHP Code to count 1(0+)1 patterns
// in a string

// Function to count patterns
function patternCount($str)
{
    // Variable to store the
```

```
// last character
$last = $str[0];

$i = 1;
$counter = 0;
while ($i < strlen($str))
{

    // We found 0 and last character
    // was '1', state change
    if ($str[$i] == '0' && $last == '1')
    {
        while ($str[$i] == '0')
            $i++;

        // After the stream of 0's,
        // we got a '1', counter
        // incremented
        if ($str[$i] == '1')
            $counter++;
    }

    /* Last character stored */
    $last = $str[$i];
    $i++;
}

return $counter;
}

// Driver Code
$str = "1001ab010abc01001";
echo patternCount($str) ;

// This code is contributed by nitin mittal
?>
```

Output :

2

Improved By : [nitin mittal](#)

## **Source**

<https://www.geeksforgeeks.org/find-patterns-101-given-string/>

## Chapter 28

# Boyer Moore Algorithm | Good Suffix heuristic

Boyer Moore Algorithm | Good Suffix heuristic - GeeksforGeeks

We have already discussed [Bad character heuristic](#) variation of Boyer Moore algorithm. In this article we will discuss **Good Suffix** heuristic for pattern searching. Just like bad character heuristic, a preprocessing table is generated for good suffix heuristic.

### Good Suffix Heuristic

Let  $t$  be substring of text  $T$  which is matched with substring of pattern  $P$ . Now we shift pattern until :

- 1) Another occurrence of  $t$  in  $P$  matched with  $t$  in  $T$ .
- 2) A prefix of  $P$ , which matches with suffix of  $t$
- 3)  $P$  moves past  $t$

#### Case 1: Another occurrence of $t$ in $P$ matched with $t$ in $T$

Pattern  $P$  might contain few more occurrences of  $t$ . In such case, we will try to shift the pattern to align that occurrence with  $t$  in text  $T$ . For example-

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| T | A | B | A | A | B | A | B | A | C | B | A  |
| P | C | A | B | A | B |   |   |   |   |   |    |

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| T | A | B | A | A | B | A | B | A | C | B | A  |
| P |   |   | C | A | B | A | B |   |   |   |    |

Figure – Case 1

**Explanation:** In above example, we have got a substring  $t$  of text  $T$  matched with pattern  $P$  (in green) before mismatch at index 2. Now we will search for occurrence of  $t$  (“AB”) in  $P$ . We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times to align  $t$  in  $P$  with  $t$  in  $T$ . This is weak rule of original Boyer Moore and not much effective, we will discuss a **Strong Good Suffix rule** shortly.

**Case 2: A prefix of  $P$ , which matches with suffix of  $t$  in  $T$**

It is not always likely that we will find occurrence of  $t$  in  $P$ . Sometimes there is no occurrence at all, in such cases sometime we can search for some **suffix of  $t$**  matching with some **prefix of  $P$**  and try to align them by shifting  $P$ . For example –



| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| T | A | A | B | A | B | A | B | A | C | B | A  |
| P | A | B | B | A | B |   |   |   |   |   |    |

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| T | A | A | B | A | B | A | B | A | C | B | A  |
| P |   |   |   | A | B | B | A | B |   |   |    |

Figure – Case 2

**Explanation:** In above example, we have got t (“BAB”) matched with P (in green) at index 2-4 before mismatch . But because there exist no occurrence of t in P we will search for some some prefix of P which match with some suffix of t. We have found prefix “AB” (in yellow background) starting at index 0 which matches not with whole t but suffix of t “AB” starting at index 3. So now we will shift pattern 3 times to align prefix with suffix.

**Case 3: P moves past t**

If above two cases are not satisfied, we will shift the pattern past the t. For example –

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| T | A | A | C | A | B | A | B | A | C | B | A  |
| P | C | B | A | A | B |   |   |   |   |   |    |

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| T | A | B | A | A | B | A | B | A | C | B | A  |
| P |   |   |   |   |   | C | B | A | A | B |    |

Figure – Case 3

**Explanation:** If above example, there exist no occurrence of  $t$  (“AB”) in  $P$  and also there is no prefix in  $P$  which matches with suffix of  $t$ . So in that case we can never find any perfect match before index 4, so we will shift the  $P$  past the  $t$  ie. to index 5.

#### Strong Good suffix Heuristic

Suppose substring  $q = P[i \text{ to } n]$  got matched with  $t$  in  $T$  and  $c = P[i-1]$  is the mismatching character. Now unlike case 1 we will search for  $t$  in  $P$  which is not preceded by character  $c$ . The closest such occurrence is then aligned with  $t$  in  $T$  by shifting pattern  $P$ . For example –

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| T | A | A | B | A | B | A | B | A | C | B | A  | C  | A  | B  | B  | C  | A  | B  |
| P | A | A | C | C | A | C | C | A | C |   |    |    |    |    |    |    |    |    |

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| T | A | A | B | A | B | A | B | A | C | B | A  | C  | A  | B  | B  | C  | A  | B  |
| P |   |   |   |   |   |   | A | A | C | C | A  | C  | C  | A  | C  |    |    |    |

Figure – strong suffix rule

**Explanation:** In above example,  $q = P[7 \text{ to } 8]$  got matched with  $t$  in  $T$ . The mismatching character  $c$  is “C” at position  $P[6]$ . Now if we start searching  $t$  in  $P$  we will get first occurrence of  $t$  starting at position 4. But this occurrence is preceded by “C” which is equal to  $c$ , so we will skip this and carry on searching. At position 1 we got another occurrence of  $t$  (in yellow background). This occurrence is preceded by “A” (in blue) which is not equivalent to  $c$ . So we will shift pattern  $P$  6 times to align this occurrence with  $t$  in  $T$ . We are doing this because we already know that character  $c = \text{“C”}$  causes the mismatch. So any occurrence of  $t$  preceded by  $c$  will again causes mismatch when aligned with  $t$ , so that's why it is better to skip this.

### Preprocessing for Good suffix heuristic

As a part of preprocessing, an array **shift** is created. Each entry **shift[i]** contain the distance pattern will shift if mismatch occur at position **i-1**. That is, the suffix of pattern starting at position **i** is matched and a mismatch occur at position **i-1**. Preprocessing is done separately for strong good suffix and case 2 discussed above.

#### 1) Preprocessing for Strong Good Suffix

Before discussing preprocessing, let us first discuss the idea of border. A **border** is a substring which is both proper suffix and proper prefix. For example, in string “ccacc”, “c” is a border, “cc” is a border because it appears in both end of string but “cca” is not a border.

As a part of preprocessing an array **bpos** (border position) is calculated. Each entry **bpos[i]** contains the starting index of border for suffix starting at index **i** in given pattern  $P$ .

The suffix beginning at position  $m$  has no border, so **bpos[m]** is set to  $m+1$  where  $m$  is the length of the pattern.

The shift position is obtained by the borders which cannot be extended to the left. Following is the code for preprocessing –

```
void preprocess_strong_suffix(int *shift, int *bpos,
                             char *pat, int m)
{
    int i = m, j = m+1;
    bpos[i] = j;
    while(i > 0)
    {
        while(j <= m && pat[i-1] != pat[j-1])
        {
            if (shift[j] == 0)
                shift[j] = j-i;
            j = bpos[j];
        }
        i--; j--;
        bpos[i] = j;
    }
}
```

**Explanation:** Consider pattern **P** = “ABBABAB”, **m** = 7.

- The widest border of suffix “AB” beginning at position **i** = 5 is (nothing) starting at position 7 so **bpos**[5] = 7.
- At position **i** = 2 the suffix is “BABAB”. The widest border for this suffix is “BAB” starting at position 4, so **j** = **bpos**[2] = 4.

```
i--;
j--;
bpos[ i ] = j
```

But if character **#** at position **i-1** do not match with character **?** at position **j-1** then we continue our search to the right. Now we know that –

- a. Border width will be smaller than the border starting at position **j** ie. smaller than **x...**
- b. Border has to begin with **#** and end with **?** or could be empty (no border exist).

With above two facts we will continue our search in sub string **x...** from position **j** to **m**. The next border should be at **j** = **bpos**[**j**]. After updating **j**, we again compare character at position **j-1** (?) with **#** and if they are equal then we got our border otherwise we continue our search to right **until j>m**. This process is shown by code –

```
while(j <= m && pat[i-1] != pat[j-1])
{
    j = bpos[j];
}
i--; j--;
bpos[i]=j;
```

In above code look at these conditions –

```
pat[i-1] != pat[j-1]
```

This is the condition which we discussed in case 2. When the character preceding the occurrence of  $t$  in pattern  $P$  is different than mismatching character in  $P$ , we stop skipping the occurrences and shift the pattern. So here  $P[i] == P[j]$  but  $P[i-1] != P[j-1]$  so we shift pattern from  $i$  to  $j$ . So  $\text{shift}[j] = j-i$  is recorder for  $j$ . So whenever any mismatch occur at position  $j$  we will shift the pattern  $\text{shift}[j+1]$  positions to the right.

In above code the following condition is very important –

```
if (shift[j] == 0 )
```

This condition prevent modification of  $\text{shift}[j]$  value from suffix having same border. For example, Consider pattern  $P = \text{"addbdddcd"}$ , when we calculate  $\text{bpos}[i-1]$  for  $i = 4$  then  $j = 7$  in this case. we will be eventually setting value of  $\text{shift}[7] = 3$ . Now if we calculate  $\text{bpos}[i-1]$  for  $i = 1$  then  $j = 7$  and we will be setting value  $\text{shift}[7] = 6$  again if there is no test  $\text{shift}[j] == 0$ . This mean if we have a mismatch at position 6 we will shift pattern  $P$  3 positions to right not 6 position.

## 2) Preprocessing for Case 2

In the preprocessing for case 2, for each suffix the **widest border of the whole pattern** that is contained in that suffix is determined.

The starting position of the widest border of the pattern at all is stored in  $\text{bpos}[0]$

In the following preprocessing algorithm, this value  $\text{bpos}[0]$  is stored initially in all free entries of array  $\text{shift}$ . But when the suffix of the pattern becomes shorter than  $\text{bpos}[0]$ , the algorithm continues with the next-wider border of the pattern, i.e. with  $\text{bpos}[j]$ .

Following is the C implementation of search algorithm –

```
/* C program for Boyer Moore Algorithm with
   Good Suffix heuristic to find pattern in
   given text string */
```

```
#include <stdio.h>
#include <string.h>
```

```
// preprocessing for strong good suffix rule
void preprocess_strong_suffix(int *shift, int *bpos,
                             char *pat, int m)
{
    // m is the length of pattern
    int i=m, j=m+1;
    bpos[i]=j;

    while(i>0)
    {
        /*if character at position i-1 is not equivalent to
        character at j-1, then continue searching to right
        of the pattern for border */
        while(j<=m && pat[i-1] != pat[j-1])
        {
            /* the character preceding the occurrence of t in
            pattern P is different than mismatching character in P,
            we stop skipping the occurrences and shift the pattern
            from i to j */
            if (shift[j]==0)
                shift[j] = j-i;

            //Update the position of next border
            j = bpos[j];
        }
        /* p[i-1] matched with p[j-1], border is found.
        store the beginning position of border */
        i--;j--;
        bpos[i] = j;
    }
}

//Preprocessing for case 2
void preprocess_case2(int *shift, int *bpos,
                     char *pat, int m)
{
    int i, j;
    j = bpos[0];
    for(i=0; i<=m; i++)
    {
        /* set the border position of first character of pattern
        to all indices in array shift having shift[i] = 0 */
        if(shift[i]==0)
            shift[i] = j;

        /* suffix become shorter than bpos[0], use the position of
        next widest border as value of j */
        if (i==j)
```

```
        j = bpos[j];
    }
}

/*Search for a pattern in given text using
  Boyer Moore algorithm with Good suffix rule */
void search(char *text, char *pat)
{
    // s is shift of the pattern with respect to text
    int s=0, j;
    int m = strlen(pat);
    int n = strlen(text);

    int bpos[m+1], shift[m+1];

    //initialize all occurrence of shift to 0
    for(int i=0;i<m+1;i++) shift[i]=0;

    //do preprocessing
    preprocess_strong_suffix(shift, bpos, pat, m);
    preprocess_case2(shift, bpos, pat, m);

    while(s <= n-m)
    {
        j = m-1;

        /* Keep reducing index j of pattern while characters of
           pattern and text are matching at this shift s*/
        while(j >= 0 && pat[j] == text[s+j])
            j--;

        /* If the pattern is present at current shift, then index j
           will become -1 after the above loop */
        if (j<0)
        {
            printf("pattern occurs at shift = %d\n", s);
            s += shift[0];
        }
        else
        {
            /*pat[i] != pat[s+j] so shift the pattern
              shift[j+1] times */
            s += shift[j+1];
        }
    }
}

//Driver
```

```
int main()
{
    char text[] = "ABAAAABAACD";
    char pat[] = "ABA";
    search(text, pat);
    return 0;
}
```

Output:

```
pattern occurs at shift = 0
pattern occurs at shift = 5
```

### References

- <http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm>

### Source

<https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/>



## Chapter 29

# is\_permutation() in C++ and its application for anagram search

is\_permutation() in C++ and its application for anagram search - GeeksforGeeks

is\_permutations() is used to check if two containers like string and vector are permutation of each other. It accepts three parameters, the first two parameters are begin and end positions of first object and third parameter begin position of second object.

```
// C++ program to demonstrate working of
// is_permutation()
#include <bits/stdc++.h>
using namespace std;

// Driver program to test above
int main()
{
    vector<int> v1{1, 2, 3, 4};
    vector<int> v2{2, 3, 1, 4};

    // v1 and v2 are permutation of each other
    if (is_permutation(v1.begin(), v1.end(), v2.begin()))
        cout << "True\n";
    else
        cout << "False\n";

    // v1 and v3 are NOT permutation of each other
    vector<int> v3{5, 3, 1, 4};
    if (is_permutation(v1.begin(), v1.end(), v3.begin()))
        cout << "True\n";
```

```
    else
        cout << "False\n";

    return 0;
}
```

Output :

```
True
False
```

### Application :

Given a pattern and a text, find all occurrences of pattern and its anagrams in text.

Examples:

```
Input : text = "forxxorfxdofr"
        pat = "for"
Output : 3
There are three anagrams of "for"
in text.
```

```
Input : word = "aabaabaa"
        text = "aaba"
Output : 4
```

We have discussed a (n) solution [her](#). But in this post it is done using `is_permutation()`. Although the complexity is higher than [previously discussed method](#), but the purpose is to explain application of `is_permutation()`.

Let size of pattern to be searched be `pat_len`. The idea is to traverse given text and for every window of size `pat_len`, check if it is a permutation of given pattern or not.

```
// C++ program to count all permutation of
// given text
#include<bits/stdc++.h>
using namespace std;

// Function to count occurrences of anagrams of pat
int countAnagrams(string text, string pat)
{
    int t_len = text.length();
    int p_len = pat.length();
```

```
// Start traversing the text
int count = 0; // Initialize result
for (int i=0; i<=t_len-p_len; i++)

    // Check if substring text[i..i+p_len]
    // is a permutation of pat[].
    // Three parameters are :
    // 1) Beginning position of current window in text
    // 2) End position of current window in text
    // 3) Pattern to be matched with current window
    if (is_permutation(text.begin()+i,
                       text.begin()+i+p_len,
                       pat.begin()))
        count++;

return count;
}

// Driver code
int main()
{
    string str = "forxxorfxdofr";
    string pat = "for";
    cout << countAnagrams(str, pat) << endl;
    return 0;
}
```

Output:

3

## Source

[https://www.geeksforgeeks.org/is\\_permutation-c-application-anagram-search/](https://www.geeksforgeeks.org/is_permutation-c-application-anagram-search/)

## Chapter 30

# Match Expression where a single special character in pattern can match one or more characters

Match Expression where a single special character in pattern can match one or more characters - GeeksforGeeks

Given two string, in which one is pattern (Pattern) and other is searching expression. Searching expression contains '#’.

The # works in following way:

1. A # matches with one or more characters.
2. A # matches all characters before a pattern match is found. For example if pat = “A#B”, and text is “ACCB”, then # would match only with “CC” and pattern is considered as not found.

### Examples :

```
Input  : str = "ABABABA"
        pat = "A#B#A"
```

```
Output : yes
```

```
Input  : str = "ABCCB"
        pat = "A#B"
```

```
Output : yes
```

```
Input  : str = "ABCABCCE"
```

```
        pat = "A#C#"
Output : yes
```

```
Input  : str = "ABCABCCE"
        pat = "A#C"
Output : no
```

We can observe that whenever we encounter '#', we have to consider as many characters till the next character of pattern will not be equal to the current character of given string. Firstly, we check if the current character of pattern is equal to '#'-

a) If not then we check whether the current character of string and pattern are same or not, if same, then increment both counters else return false from here only. No need for further checking.

b) If yes, then we have to find the position of a character in text that matches with next character of pattern.

C++

```
// C++ program for pattern matching
// where a single special character
// can match one more characters
#include<bits/stdc++.h>

using namespace std;

// Returns true if pat matches with text
int regexMatch(string text, string pat)
{
    int lenText = text.length();
    int letPat = pat.length();

    // i is used as an index in pattern
    // and j as an index in text
    int i = 0, j = 0;

    // Traverse through pattern
    while (i < letPat)
    {
        // If current character of
        // pattern is not '#'
        if (pat[i] != '#')
        {
            // If does not match with text
            if (pat[i] != text[j])
                return false;

            // If matches, increment i and j
        }
    }
}
```

```
        i++;
        j++;
    }

    // Current character is '#'
    else
    {
        // At least one character
        // must match with #
        j++;

        // Match characters with # until
        // a matching character is found.
        while (text[j] != pat[i + 1])
            j++;

        // Matching with # is over,
        // move ahead in pattern
        i++;
    }
}

return (j == lenText);
}

// Driver code
int main()
{
    string str = "ABABABA";
    string pat = "A#B#A";
    if (regexMatch(str, pat))
        cout << "yes";
    else
        cout << "no";
    return 0;
}
```

## Java

```
// Java program for pattern matching
// where a single special character
// can match one more characters

import java.util.*;
import java.lang.*;
import java.io.*;

class GFG
```

```
{
    // Returns true if pat
    // matches with text.
    public static boolean regexMatch
        (String text, String pat)
    {
        int lenText = text.length();
        int lenPat = pat.length();

        char[] Text = text.toCharArray();
        char[] Pat = pat.toCharArray();

        // i is used as an index in pattern
        // and j as an index in text.
        int i = 0, j = 0;

        // Traverse through pattern
        while (i < lenPat)
        {
            // If current character of
            // pattern is not '#'
            if (Pat[i] != '#')
            {
                // If does not match with text.
                if (Pat[i] != Text[j])
                    return false;

                // If matches, increment i and j
                i++;
                j++;
            }

            // Current character is '#'
            else
            {
                // At least one character
                // must match with #
                j++;

                // Match characters with # until
                // a matching character is found.
                while (Text[j] != Pat[i + 1])
                    j++;

                // Matching with # is over,
                // move ahead in pattern
                i++;
            }
        }
    }
}
```

```
    }

    return (j == lenText);
}

// Driver code
public static void main (String[] args)
{
    String str = "ABABABA";
    String pat = "A#B#A";
    if (regexMatch(str, pat))
        System.out.println("yes");
    else
        System.out.println("no");
}
}

// This code is contributed by Mr. Somesh Awasthi
```

## C#

```
// C# program for pattern matching
// where a single special character
// can match one more characters
using System;

class GFG
{
    // Returns true if pat
    // matches with text.
    public static bool regexMatch
        (String text, String pat)
    {
        int lenText = text.Length;
        int lenPat = pat.Length;

        char []Text = text.ToCharArray();
        char []Pat = pat.ToCharArray();

        // i is used as an index in pattern
        // and j as an index in text.
        int i = 0, j = 0;

        // Traverse through pattern
        while (i < lenPat)
        {
            // If current character
            // of pattern is not '#'

```



```
        if (Pat[i] != '#')
        {
            // If does not match with text.
            if (Pat[i] != Text[j])
                return false;

            // If matches, increment i and j
            i++;
            j++;
        }

        // Current character is '#'
        else
        {
            // At least one character
            // must match with #
            j++;

            // Match characters with # until
            // a matching character is found.
            while (Text[j] != Pat[i + 1])
                j++;

            // Matching with # is over,
            // move ahead in pattern
            i++;
        }
    }

    return (j == lenText);
}

// Driver code
public static void Main ()
{
    String str = "ABABABA";
    String pat = "A#B#A";
    if (regexMatch(str, pat))
        Console.Write("yes");
    else
        Console.Write("no");
}

// This code is contributed by nitin mittal
```

**PHP**

```
<?php
// PHP program for pattern matching
// where a single special character
// can match one more characters

// Returns true if pat
// matches with text
function regexMatch($text, $pat)
{
    $lenText = strlen($text);
    $letPat = strlen($pat);

    // i is used as an index in pattern
    // and j as an index in text
    $i = 0; $j = 0;

    // Traverse through pattern
    while ($i < $letPat)
    {

        // If current character of
        // pattern is not '#'
        if ($pat[$i] != '#')
        {

            // If does not match with text
            if ($pat[$i] != $text[$j])
                return false;

            // If matches, increment i and j
            $i++;
            $j++;
        }

        // Current character is '#'
        else
        {

            // At least one character
            // must match with #
            $j++;

            // Match characters with # until
            // a matching character is found.
            while ($text[$j] != $pat[$i + 1])
                $j++;

            // Matching with # is over,
```

```
        // move ahead in pattern
        $i++;
    }
}

return ($j == $lenText);
}

// Driver code
$str = "ABABABA";
$pat = "A#B#A";
if (regexMatch($str, $pat))
    echo "yes";
else
    echo "no";

// This code is contributed by nitin mittal
?>
```

Output:

yes

Improved By : [nitin mittal](#)

## Source

<https://www.geeksforgeeks.org/match-expression-where-a-single-special-character-in-pattern-can-match-one-or-more-characters/>

## Chapter 31

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

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

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
digger
biggerdiagram
Prefix "dig" of s is longest subsequence in t.
```

```
Input : s = "geeksforgeeks"
        t = "agbcdfeitk"
Output : 4
```

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 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:

C++

```
// C++ program to find maximum
// length prefix of one string
// occur as subsequence in another
// string.
#include<bits/stdc++.h>
using namespace std;

// Return the maximum length
// prefix which is subsequence.
int maxPrefix(char s[], char t[])
{
    int count = 0;

    // Iterating string T.
    for (int i = 0; i < strlen(t); i++)
    {
        // If end of string S.
        if (count == strlen(s))
            break;

        // If character match,
        // increment counter.
        if (t[i] == s[count])
            count++;
    }

    return count;
}

// Driven Code
int main()
{
    char S[] = "digger";
    char T[] = "biggerdiagram";

    cout << maxPrefix(S, T)
         << endl;

    return 0;
}
```

## Java

```
// Java program to find maximum
// length prefix of one string
// occur as subsequence in another
// string.
public class GFG {

    // Return the maximum length
    // prefix which is subsequence.
    static int maxPrefix(String s,
                          String t)
    {
        int count = 0;

        // Iterating string T.
        for (int i = 0; i < t.length(); i++)
        {
            // If end of string S.
            if (count == t.length())
                break;

            // If character match,
            // increment counter.
            if (t.charAt(i) == s.charAt(count))
                count++;
        }

        return count;
    }

    // Driver Code
    public static void main(String args[])
    {
        String S = "digger";
        String T = "biggerdiagram";

        System.out.println(maxPrefix(S, T));
    }
}
// This code is contributed by Sumit Ghosh
```

## Python 3

```
# 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.
```

**C#**

```
// C# program to find maximum
// length prefix of one string
// occur as subsequence in
// another string.
using System;

class GFG
{
    // Return the maximum length prefix
    // which is subsequence.
    static int maxPrefix(String s,
                          String t)
```

```
{
    int count = 0;

    // Iterating string T.
    for (int i = 0; i < t.Length; i++)
    {
        // If end of string S.
        if (count == t.Length)
            break;

        // If character match,
        // increment counter.
        if (t[i] == s[count])
            count++;
    }

    return count;
}

// Driver Code
public static void Main()
{
    String S = "digger";
    String T = "biggerdiagram";

    Console.Write(maxPrefix(S, T));
}

// This code is contributed by nitin mittal
```

## PHP

```
<?php
// PHP program to find maximum
// length prefix of one string
// occur as subsequence in another
// string.

// Return the maximum length
// prefix which is subsequence.
function maxPrefix($s, $t)
{
    $count = 0;

    // Iterating string T.
    for ($i = 0; $i < strlen($t); $i++)
    {
```



```
// If end of string S.
if ($count == strlen($s))
    break;

// If character match,
// increment counter.
if ($t[$i] == $s[$count])
    $count++;
}

return $count;
}

// Driver Code
{
    $S = "digger";
    $T = "biggerdiagram";

    echo maxPrefix($S, $T) ;

    return 0;
}

// This code is contributed by nitin mittal.
?>
```

**Output :**

3

**Improved By :** [nitin mittal](#)

**Source**

<https://www.geeksforgeeks.org/maximum-length-prefix-one-string-occurs-subsequence-another/>

## Chapter 32

# Replace all occurrences of string AB with C without using extra space

Replace all occurrences of string AB with C without using extra space - GeeksforGeeks

Given a string **str** that may contain one more occurrences of “AB”. Replace all occurrences of “AB” with “C” in str.

Examples:

Input : str = "helloABworld"

Output : str = "helloCworld"

Input : str = "fghABsdfABysu"

Output : str = "fghCsdfCysu"

A **simple solution** is to find all occurrences of “AB”. For every occurrence, replace it with C and move all characters one position back.

C++

```
// C++ program to replace all occurrences of "AB"
// with "C"
#include <bits/stdc++.h>

void translate(char* str)
{
    if (str[0] == '\0')
        return;
```

```
// Start traversing from second character
for (int i=1; str[i] != ''; i++)
{
    // If previous character is 'A' and
    // current character is 'B'
    if (str[i-1]=='A' && str[i]=='B')
    {
        // Replace previous character with
        // 'C' and move all subsequent
        // characters one position back
        str[i-1] = 'C';
        for (int j=i; str[j]!=''; j++)
            str[j] = str[j+1];
    }
}
return;
}

// Driver code
int main()
{
    char str[] = "helloABworldABGfG";
    translate(str);
    printf("The modified string is :\n");
    printf("%s", str);
}
```

## Java

```
// Java program to replace all
// occurrences of "AB" with "C"
import java.io.*;

class GFG {

    static void translate(char str[])
    {
        // Start traversing from second character
        for (int i = 1; i < str.length; i++)
        {
            // If previous character is 'A' and
            // current character is 'B'
            if (str[i - 1] == 'A' && str[i] == 'B')
            {
                // Replace previous character with
                // 'C' and move all subsequent
                // characters one position back
            }
        }
    }
}
```

```
        str[i - 1] = 'C';
        int j;
        for (j = i; j < str.length - 1; j++)
            str[j] = str[j + 1];
        str[j] = ' ';
    }
}
return;
}

// Driver code
public static void main(String args[])
{
    String st = "helloABworldABGfG";
    char str[] = st.toCharArray();
    translate(str);
    System.out.println("The modified string is :");
    System.out.println(str);
}

// This code is contributed by Nikita Tiwari.
```

### Python3

```
# Python 3 program to replace all
# occurrences of "AB" with "C"

def translate(st) :

    # Start traversing from second chracter
    for i in range(1, len(st)) :

        # If previous character is 'A'
        # and current character is 'B'
        if (st[i - 1] == 'A' and st[i] == 'B') :

            # Replace previous character with
            # 'C' and move all subsequent
            # characters one position back
            st[i - 1] = 'C'

            for j in range(i, len(st) - 1) :
                st[j] = st[j + 1]

            st[len(st) - 1] = ' '
```

```
        return

# Driver code
st = list("helloABworldABGfG")
translate(st)

print("The modified string is :")
print(''.join(st))

# This code is contributed by Nikita Tiwari.
```

**C#**

```
// C# program to replace all
// occurrences of "AB" with "C"
using System;

class GFG {

    static void translate(char []str)
    {

        // Start traversing from second chracter
        for (int i = 1; i < str.Length; i++)
        {
            // If previous character is 'A' and
            // current character is 'B'
            if (str[i - 1] == 'A' && str[i] == 'B')
            {
                // Replace previous character with
                // 'C' and move all subsequent
                // characters one position back
                str[i - 1] = 'C';
                int j;
                for (j = i; j < str.Length - 1; j++)
                    str[j] = str[j + 1];
                str[j] = ' ';
            }
        }
        return;
    }

    // Driver code
    public static void Main()
    {
        String st = "helloABworldABGfG";
```

```
        char []str = st.ToCharArray();
        translate(str);
        Console.WriteLine("The modified string is :");
        Console.Write(str);
    }
}
```

// This code is contributed by Nitin Mittal.

Output :

```
The modified string is :
helloCworldCGfG
```

Time Complexity :  $O(n^2)$   
Auxiliary Space :  $O(1)$

An **efficient solution** is to keep track of two indexes, one for modified string (**i** in below code) and other for original string (**j** in below code). If we find “AB” at current index j, we increment j by 2 and i by 1. Otherwise we increment both and copy character from j to i.

Below is implementation of above idea.

**C++**

```
// Efficient C++ program to replace all occurrences
// of "AB" with "C"
#include <bits/stdc++.h>

void translate(char* str)
{
    int len = strlen(str);
    if (len < 2)
        return;

    int i = 0; // Index in modified string
    int j = 0; // Index in original string

    // Traverse string
    while (j < len-1)
    {
        // Replace occurrence of "AB" with "C"
        if (str[j] == 'A' && str[j+1] == 'B')
        {
            // Increment j by 2
            j = j + 2;
        }
        else
        {
            str[i] = str[j];
            i++;
            j++;
        }
    }
    str[i] = '\0';
}
```

```
        str[i++] = 'C';
        continue;
    }
    str[i++] = str[j++];
}

if (j == len-1)
    str[i++] = str[j];

// add a null character to terminate string
str[i] = '\0';
}

// Driver code
int main()
{
    char str[] = "helloABworldABGfG";
    translate(str);
    printf("The modified string is :\n");
    printf("%s", str);
}
```

## Java

```
// Efficient Java program to replace
// all occurrences of "AB" with "C"
import java.io.*;

class GFG {

    static void translate(char str[])
    {
        int len = str.length;
        if (len < 2)
            return;

        // Index in modified string
        int i = 0;

        // Index in original string
        int j = 0;

        // Traverse string
        while (j < len - 1)
        {
            // Replace occurrence of "AB" with "C"
            if (str[j] == 'A' && str[j + 1] == 'B')
            {
```

```
        // Increment j by 2
        j = j + 2;
        str[i++] = 'C';
        continue;
    }
    str[i++] = str[j++];
}

if (j == len - 1)
    str[i++] = str[j];

// add a null character to terminate string
str[i] = ' ';
str[len - 1] = ' ';
}

// Driver code
public static void main(String args[])
{
    String st="helloABworldABGfG";
    char str[] = st.toCharArray();
    translate(str);
    System.out.println("The modified string is :");
    System.out.println(str);
}

// This code is contributed
// by Nikita Tiwari.
```

### Python3

```
# Python 3 program to replace all
# occurrences of "AB" with "C"

def translate(st) :
    l = len(st)

    if (l < 2) :
        return

    i = 0 # Index in modified string
    j = 0 # Index in original string

    # Traverse string
    while (j < l - 1) :
```



```
# Replace occurrence of "AB" with "C"
if (st[j] == 'A' and st[j + 1] == 'B') :

    # Increment j by 2
    j += 2
    st[i] = 'C'
    i += 1
    continue

st[i] = st[j]
i += 1
j += 1

if (j == l - 1) :
    st[i] = st[j]
    i += 1

# add a null character to
# terminate string
st[i] = ' '
st[l-1] = ' '

# Driver code
st = list("helloABworldABGfG")
translate(st)

print("The modified string is :")
print(''.join(st))

# This code is contributed by Nikita Tiwari.
```

## C#

```
// Efficient C# program to replace
// all occurrences of "AB" with "C"
using System;

class GFG {

    static void translate(char []str)
    {
        int len = str.Length;
        if (len < 2)
            return;

        // Index in modified string
        int i = 0;
```

```
// Index in original string
int j = 0;

// Traverse string
while (j < len - 1)
{
    // Replace occurrence of "AB" with "C"
    if (str[j] == 'A' && str[j + 1] == 'B')
    {
        // Increment j by 2
        j = j + 2;
        str[i++] = 'C';
        continue;
    }
    str[i++] = str[j++];
}

if (j == len - 1)
    str[i++] = str[j];

// add a null character to
// terminate string
str[i] = ' ';
str[len - 1] = ' ';
}

// Driver code
public static void Main()
{
    String st="helloABworldABGfG";
    char []str = st.ToCharArray();
    translate(str);
    Console.WriteLine("The modified string is :");
    Console.WriteLine(str);
}

// This code is contributed by nitin mittal.
```

Output:

The modified string is :  
helloCworldCGfG

Time Complexity :  $O(n)$

Auxiliary Space :  $O(1)$

**Improved By :** [nitin mittal](#)

### Source

<https://www.geeksforgeeks.org/replace-occurrences-string-ab-c-without-using-extra-space/>

## Chapter 33

# Wildcard Pattern Matching

Wildcard Pattern Matching - GeeksforGeeks

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
```

Input = ba aab ab  
 Pattern = \*\*\*\*\*ba\*\*\*\*\*ab  
 Output : true

Input = baaabab  
 Pattern = a \* ab  
 Output : false

No matching text

Input = ba aab ab  
 Pattern = ba \* a?  
 Output : true

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 arise

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
```

Below is the implementation of above Dynamic Programming approach.

**C++**

```
// 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 n, int m)
{
    // empty pattern can only match with
    // empty string
    if (m == 0)
        return (n == 0);

    // lookup table for storing results of
    // subproblems
    bool lookup[n + 1][m + 1];

    // initialize lookup table to false
    memset(lookup, false, sizeof(lookup));

    // empty pattern can match with empty string
    lookup[0][0] = true;

    // Only '*' can match with empty string
    for (int j = 1; j <= m; j++)
        if (pattern[j - 1] == '*')
            lookup[0][j] = lookup[0][j - 1];

    // fill the table in bottom-up fashion
    for (int i = 1; i <= n; i++)
    {
        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[j - 1] == '*')
                lookup[i][j] = lookup[i][j - 1] ||
                               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] == '?' ||

```

```
        str[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];
}

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;
}
```

## Java

```
// Java program to implement wildcard
// pattern matching algorithm
import java.util.Arrays;
public class GFG{

    // Function that matches input str with
    // given wildcard pattern
```



```
static boolean strmatch(String str, String pattern,
                        int n, int m)
{
    // empty pattern can only match with
    // empty string
    if (m == 0)
        return (n == 0);

    // lookup table for storing results of
    // subproblems
    boolean[][] lookup = new boolean[n + 1][m + 1];

    // initialize lookup table to false
    for(int i = 0; i < n + 1; i++)
        Arrays.fill(lookup[i], false);

    // empty pattern can match with empty string
    lookup[0][0] = true;

    // Only '*' can match with empty string
    for (int j = 1; j <= m; j++)
        if (pattern.charAt(j - 1) == '*')
            lookup[0][j] = lookup[0][j - 1];

    // fill the table in bottom-up fashion
    for (int i = 1; i <= n; i++)
    {
        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.charAt(j - 1) == '*')
                lookup[i][j] = lookup[i][j - 1] ||
                    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.charAt(j - 1) == '?' ||
                    str.charAt(i - 1) == pattern.charAt(j - 1))
                lookup[i][j] = lookup[i - 1][j - 1];
        }
    }
}
```

```

        // If characters don't match
        else lookup[i][j] = false;
    }
}

return lookup[n][m];
}

public static void main(String args[])
{
    String str = "baaabab";
    String pattern = "*****ba*****ab";
    // String pattern = "ba*****ab";
    // String pattern = "ba*ab";
    // String pattern = "a*ab";
    // String pattern = "a*****ab";
    // String pattern = "*a*****ab";
    // String pattern = "ba*ab****";
    // String pattern = "****";
    // String pattern = "*";
    // String pattern = "aa?ab";
    // String pattern = "b*b";
    // String pattern = "a*a";
    // String pattern = "baaabab";
    // String pattern = "?baaabab";
    // String pattern = "*baaaba*";

    if (strmatch(str, pattern, str.length(),
                pattern.length()))
        System.out.println("Yes");
    else
        System.out.println("No");
}
}
// This code is contributed by Sumit Ghosh

```

Output :

Yes

Time complexity of above solution is  $O(m \times n)$ . Auxiliary space used is also  $O(m \times n)$ .

#### Further Improvements:

We can improve space complexity by making use of the fact that we only use the result from last row.

One more improvement is to 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.

### Source

<https://www.geeksforgeeks.org/wildcard-pattern-matching/>

## Chapter 34

# Find all occurrences of a given word in a matrix

Find all occurrences of a given word in a matrix - GeeksforGeeks

Given a 2D grid of characters and a word, find all occurrences of given word in 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 solution should print all coordinates if a cycle is found. i.e.

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

Input:

```
mat[ROW][COL] = { {'B', 'N', 'E', 'Y', 'S'},
                   {'H', 'E', 'D', 'E', 'S'},
                   {'S', 'G', 'N', 'D', 'E'}
                 };
```

Word = "DES"

Output:

```
D(1, 2) E(1, 1) S(2, 0)
D(1, 2) E(1, 3) S(0, 4)
D(1, 2) E(1, 3) S(1, 4)
D(2, 3) E(1, 3) S(0, 4)
D(2, 3) E(1, 3) S(1, 4)
D(2, 3) E(2, 4) S(1, 4)
```

Input:

```
char mat[ROW][COL] = { {'B', 'N', 'E', 'Y', 'S'},
                       {'H', 'E', 'D', 'E', 'S'},
                       {'S', 'G', 'N', 'D', 'E'} };
char word[] = "BNEGSHBN";
```

Output:

B(0, 0) N(0, 1) E(1, 1) G(2, 1) S(2, 0) H(1, 0)  
B(0, 0) N(0, 1)

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | B | N | E | Y | S |
| 1 | H | E | D | E | S |
| 2 | S | G | N | D | E |

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | B | N | E | Y | S |
| 1 | H | E | D | E | S |
| 2 | S | G | N | D | E |

We strongly recommend you to minimize your browser and try this yourself first.

This is mainly an extension of [this](#) post. Here with locations path is also printed.

The problem can be easily solved by applying DFS() on each occurrence of first character of the word in the matrix. A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursive call for 8 neighbours only.

```
// Program to find all occurrences of the word in
// a matrix
```

```
#include <bits/stdc++.h>
using namespace std;

#define ROW 3
#define COL 5

// check whether given cell (row, col) is a valid
// cell or not.
bool isValid(int row, int col, int prevRow, int prevCol)
{
    // return true if row number and column number
    // is in range
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL) &&
           !(row == prevRow && col == prevCol);
}

// These arrays are used to get row and column
// numbers of 8 neighbours of a given cell
int rowNum[] = {-1, -1, -1, 0, 0, 1, 1, 1};
int colNum[] = {-1, 0, 1, -1, 1, -1, 0, 1};

// A utility function to do DFS for a 2D boolean
// matrix. It only considers the 8 neighbours as
// adjacent vertices
void DFS(char mat[][COL], int row, int col,
         int prevRow, int prevCol, char* word,
         string path, int index, int n)
{
    // return if current character doesn't match with
    // the next character in the word
    if (index > n || mat[row][col] != word[index])
        return;

    //append current character position to path
    path += string(1, word[index]) + "(" + to_string(row)
            + ", " + to_string(col) + ") ";

    // current character matches with the last character
    // in the word
    if (index == n)
    {
        cout << path << endl;
        return;
    }

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
```

```

        if (isvalid(row + rowNum[k], col + colNum[k],
                    prevRow, prevCol))

            DFS(mat, row + rowNum[k], col + colNum[k],
                row, col, word, path, index+1, n);
    }

    // The main function to find all occurrences of the
    // word in a matrix
    void findWords(char mat[][COL], char* word, int n)
    {
        // traverse through the all cells of given matrix
        for (int i = 0; i < ROW; ++i)
            for (int j = 0; j < COL; ++j)

                // occurrence of first character in matrix
                if (mat[i][j] == word[0])

                    // check and print if path exists
                    DFS(mat, i, j, -1, -1, word, "", 0, n);
    }

    // Driver program to test above function
    int main()
    {
        char mat[ROW][COL] = { {'B', 'N', 'E', 'Y', 'S'},
                                {'H', 'E', 'D', 'E', 'S'},
                                {'S', 'G', 'N', 'D', 'E'}
                                };

        char word[] = "DES";

        findWords(mat, word, strlen(word) - 1);

        return 0;
    }

```

Output :

```

D(1, 2) E(1, 1) S(2, 0)
D(1, 2) E(1, 3) S(0, 4)
D(1, 2) E(1, 3) S(1, 4)
D(2, 3) E(1, 3) S(0, 4)
D(2, 3) E(1, 3) S(1, 4)
D(2, 3) E(2, 4) S(1, 4)

```

## **Source**

<https://www.geeksforgeeks.org/find-all-occurrences-of-the-word-in-a-matrix/>



## Chapter 35

# Aho-Corasick Algorithm for Pattern Searching

Aho-Corasick Algorithm for Pattern Searching - GeeksforGeeks

Given an input text and an array of  $k$  words, `arr[]`, find all occurrences of all words in the input text. Let  $n$  be the length of text and  $m$  be the total number characters in all words, i.e.  $m = \text{length}(\text{arr}[0]) + \text{length}(\text{arr}[1]) + \dots + \text{length}(\text{arr}[k-1])$ . Here  $k$  is total numbers of input words.

Example:

```
Input: text = "ahishers"
       arr[] = {"he", "she", "hers", "his"}
```

Output:

```
Word his appears from 1 to 3
Word he appears from 4 to 5
Word she appears from 3 to 5
Word hers appears from 4 to 7
```

If we use a linear time searching algorithm like **KMP**, then we need to one by one search all words in `text[]`. This gives us total time complexity as  $O(n + \text{length}(\text{word}[0]) + O(n + \text{length}(\text{word}[1]) + O(n + \text{length}(\text{word}[2]) + \dots + O(n + \text{length}(\text{word}[k-1]))$ . This time complexity can be written as  $O(n * k + m)$ .

**Aho-Corasick Algorithm** finds all words in  $O(n + m + z)$  time where  $z$  is total number of occurrences of words in text. The Aho-Corasick string matching algorithm formed the basis of the original Unix command `fgrep`.

1. **Preprocessing** : Build an automaton of all words in `arr[]` The automaton has mainly three functions:

**Go To :** This function simply follows edges of Trie of all words in `arr[]`. It is represented as 2D array `g[][]` where we store next state for current state and character.

**Failure :** This function stores all edges that are followed when current character doesn't have edge in Trie. It is represented as 1D array `f[]` where we store next state for current state.

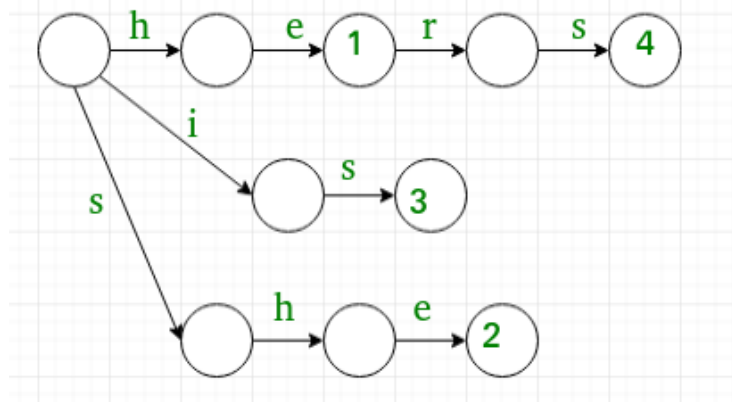
**Output :** Stores indexes of all words that end at current state. It is represented as 1D array `o[]` where we store indexes of all matching words as a bitmap for current state.

2. **Matching :** Traverse the given text over built automaton to find all matching words.

#### Preprocessing:

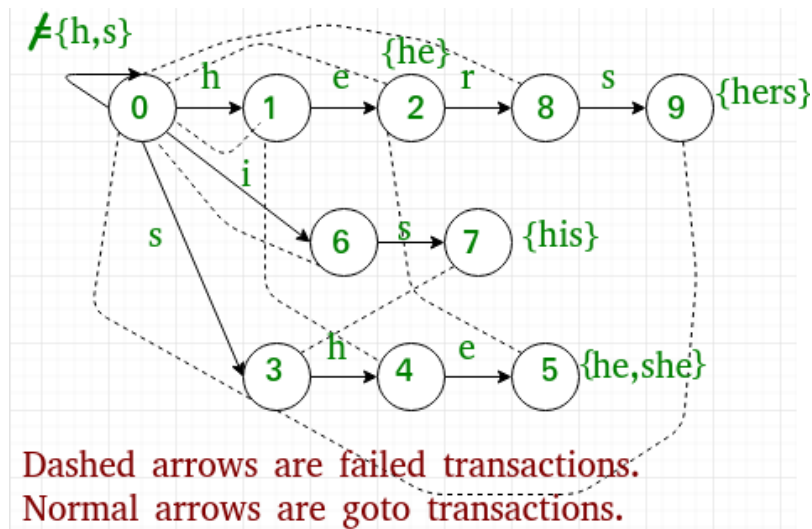
1. We first Build a [Trie](#) (or Keyword Tree) of all words.

**Trie for Arr[] = { he , she, his , hers }**



This part fills entries in `goto g[][]` and `output o[]`.

2. Next we extend Trie into an automaton to support linear time matching.



This part fills entries in failure `f[]` and output `o[]`.

#### Go to :

We build [Trie](#). And for all characters which don't have an edge at root, we add an edge back to root.

#### Failure :

For a state `s`, we find the longest proper suffix which is a proper prefix of some pattern. This is done using Breadth First Traversal of Trie.

#### Output :

For a state `s`, indexes of all words ending at `s` are stored. These indexes are stored as bitwise map (by doing bitwise OR of values). This is also computing using Breadth First Traversal with Failure.

Below is C++ implementation of Aho-Corasick Algorithm

```
// C++ program for implementation of Aho Corasick algorithm
// for string matching
using namespace std;
#include <bits/stdc++.h>

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 500;

// Maximum number of characters in input alphabet
const int MAXC = 26;

// OUTPUT FUNCTION IS IMPLEMENTED USING out[]
// Bit i in this mask is one if the word with index i
// appears when the machine enters this state.
int out[MAXS];
```

```
// FAILURE FUNCTION IS IMPLEMENTED USING f[]
int f[MAXS];

// GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][]
int g[MAXS][MAXC];

// Builds the string matching machine.
// arr - array of words. The index of each keyword is important:
//       "out[state] & (1 << i)" is > 0 if we just found word[i]
//       in the text.
// Returns the number of states that the built machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(string arr[], int k)
{
    // Initialize all values in output function as 0.
    memset(out, 0, sizeof out);

    // Initialize all values in goto function as -1.
    memset(g, -1, sizeof g);

    // Initially, we just have the 0 state
    int states = 1;

    // Construct values for goto function, i.e., fill g[][]
    // This is same as building a Trie for arr[]
    for (int i = 0; i < k; ++i)
    {
        const string &word = arr[i];
        int currentState = 0;

        // Insert all characters of current word in arr[]
        for (int j = 0; j < word.size(); ++j)
        {
            int ch = word[j] - 'a';

            // Allocate a new node (create a new state) if a
            // node for ch doesn't exist.
            if (g[currentState][ch] == -1)
                g[currentState][ch] = states++;

            currentState = g[currentState][ch];
        }

        // Add current word in output function
        out[currentState] |= (1 << i);
    }
}
```

```
// For all characters which don't have an edge from
// root (or state 0) in Trie, add a goto edge to state
// 0 itself
for (int ch = 0; ch < MAXC; ++ch)
    if (g[0][ch] == -1)
        g[0][ch] = 0;

// Now, let's build the failure function

// Initialize values in fail function
memset(f, -1, sizeof f);

// Failure function is computed in breadth first order
// using a queue
queue<int> q;

// Iterate over every possible input
for (int ch = 0; ch < MAXC; ++ch)
{
    // All nodes of depth 1 have failure function value
    // as 0. For example, in above diagram we move to 0
    // from states 1 and 3.
    if (g[0][ch] != 0)
    {
        f[g[0][ch]] = 0;
        q.push(g[0][ch]);
    }
}

// Now queue has states 1 and 3
while (q.size())
{
    // Remove the front state from queue
    int state = q.front();
    q.pop();

    // For the removed state, find failure function for
    // all those characters for which goto function is
    // not defined.
    for (int ch = 0; ch <= MAXC; ++ch)
    {
        // If goto function is defined for character 'ch'
        // and 'state'
        if (g[state][ch] != -1)
        {
            // Find failure state of removed state
            int failure = f[state];
```

```
        // Find the deepest node labeled by proper
        // suffix of string from root to current
        // state.
        while (g[failure][ch] == -1)
            failure = f[failure];

        failure = g[failure][ch];
        f[g[state][ch]] = failure;

        // Merge output values
        out[g[state][ch]] |= out[failure];

        // Insert the next level node (of Trie) in Queue
        q.push(g[state][ch]);
    }
}

return states;
}

// Returns the next state the machine will transition to using goto
// and failure functions.
// currentState - The current state of the machine. Must be between
//                0 and the number of states - 1, inclusive.
// nextInput - The next character that enters into the machine.
int findNextState(int currentState, char nextInput)
{
    int answer = currentState;
    int ch = nextInput - 'a';

    // If goto is not defined, use failure function
    while (g[answer][ch] == -1)
        answer = f[answer];

    return g[answer][ch];
}

// This function finds all occurrences of all array words
// in text.
void searchWords(string arr[], int k, string text)
{
    // Preprocess patterns.
    // Build machine with goto, failure and output functions
    buildMatchingMachine(arr, k);

    // Initialize current state
    int currentState = 0;
```

```
// Traverse the text through the built machine to find
// all occurrences of words in arr[]
for (int i = 0; i < text.size(); ++i)
{
    currentState = findNextState(currentState, text[i]);

    // If match not found, move to next state
    if (out[currentState] == 0)
        continue;

    // Match found, print all matching words of arr[]
    // using output function.
    for (int j = 0; j < k; ++j)
    {
        if (out[currentState] & (1 << j))
        {
            cout << "Word " << arr[j] << " appears from "
                 << i - arr[j].size() + 1 << " to " << i << endl;
        }
    }
}

// Driver program to test above
int main()
{
    string arr[] = {"he", "she", "hers", "his"};
    string text = "ahishers";
    int k = sizeof(arr)/sizeof(arr[0]);

    searchWords(arr, k, text);

    return 0;
}
```

Output:

```
Word his appears from 1 to 3
Word he appears from 4 to 5
Word she appears from 3 to 5
Word hers appears from 4 to 7
```

**Source:**

<http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf>

This article is contributed by **Ayush Govil**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [Pawel Wolowiec](#)

## **Source**

<https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/>



## Chapter 36

# kasai's Algorithm for Construction of LCP array from Suffix Array

kasai's Algorithm for Construction of LCP array from Suffix Array - GeeksforGeeks

### Background

**Suffix Array :** A suffix array is a sorted array of all suffixes of a given string. Let the given string be “banana”.

|          |                   |          |
|----------|-------------------|----------|
| 0 banana |                   | 5 a      |
| 1 anana  | Sort the Suffixes | 3 ana    |
| 2 nana   | ----->            | 1 anana  |
| 3 ana    | alphabetically    | 0 banana |
| 4 na     |                   | 4 na     |
| 5 a      |                   | 2 nana   |

The suffix array for “banana” :

`suffix[] = {5, 3, 1, 0, 4, 2}`

We have discussed [Suffix Array and its  \$O\(n \log n\)\$  construction](#).

Once Suffix array is built, we can use it to efficiently search a pattern in a text. For example, we can use Binary Search to find a pattern (Complete code for the same is discussed [here](#))

### LCP Array

The Binary Search based solution discussed [here](#) takes  $O(m \cdot \log n)$  time where  $m$  is length of the pattern to be searched and  $n$  is length of the text. With the help of LCP array, we can search a pattern in  $O(m + \log n)$  time. For example, if our task is to search “ana” in “banana”,  $m = 3$ ,  $n = 5$ .

**LCP Array is an array of size  $n$  (like Suffix Array). A value  $lcp[i]$  indicates length of the longest common prefix of the suffixes indexed by  $suffix[i]$  and  $suffix[i+1]$ .  $suffix[n-1]$  is not defined as there is no suffix after it.**

```
txt[0..n-1] = "banana"
suffix[]    = {5, 3, 1, 0, 4, 2}
lcp[]       = {1, 3, 0, 0, 2, 0}
```

Suffixes represented by suffix array in order are:  
{`"a"`, `"ana"`, `"anana"`, `"banana"`, `"na"`, `"nana"`}

```
lcp[0] = Longest Common Prefix of "a" and "ana"      = 1
lcp[1] = Longest Common Prefix of "ana" and "anana" = 3
lcp[2] = Longest Common Prefix of "anana" and "banana" = 0
lcp[3] = Longest Common Prefix of "banana" and "na" = 0
lcp[4] = Longest Common Prefix of "na" and "nana" = 2
lcp[5] = Longest Common Prefix of "nana" and None = 0
```

### How to construct LCP array?

LCP array construction is done two ways:

- 1) Compute the LCP array as a byproduct to the suffix array (Manber & Myers Algorithm)
- 2) Use an already constructed suffix array in order to compute the LCP values. (Kasai Algorithm).

There exist algorithms that can construct Suffix Array in  $O(n)$  time and therefore we can always construct LCP array in  $O(n)$  time. But in the below implementation, a  $O(n \log n)$  algorithm is discussed.

### kasai's Algorithm

In this article kasai's Algorithm is discussed. The algorithm constructs LCP array from suffix array and input text in  $O(n)$  time. The idea is based on below fact:

Let lcp of suffix beginning at  $txt[i]$  be  $k$ . If  $k$  is greater than 0, then lcp for suffix beginning at  $txt[i+1]$  will be at-least  $k-1$ . The reason is, relative order of characters remain same. If we delete the first character from both suffixes, we know that at least  $k$  characters will match. For example for substring `"ana"`, lcp is 3, so for string `"na"` lcp will be at-least 2. Refer [this](#) for proof.

Below is C++ implementation of Kasai's algorithm.

```
// C++ program for building LCP array for given text
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
```

```
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
        (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according to first
    // 2 characters. Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n]; // This array is needed to get the index in suffixes[]
    // from original index. This mapping is needed to get
    // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)
```

```
{
    // If first rank and next ranks are same as that of previous
    // suffix in array, assign the same new rank to this suffix
    if (suffixes[i].rank[0] == prev_rank &&
        suffixes[i].rank[1] == suffixes[i-1].rank[1])
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = rank;
    }
    else // Otherwise increment rank and assign
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = ++rank;
    }
    ind[suffixes[i].index] = i;
}

// Assign next rank to every suffix
for (int i = 0; i < n; i++)
{
    int nextindex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextindex < n)?
        suffixes[ind[nextindex]].rank[0] : -1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
vector<int> suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{
    int n = suffixArr.size();

    // To store LCP array
    vector<int> lcp(n, 0);

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the
```

```
// invSuff[5] would store 0. This is used to get next
// suffix string from suffix array.
vector<int> invSuff(n, 0);

// Fill values in invSuff[]
for (int i=0; i < n; i++)
    invSuff[suffixArr[i]] = i;

// Initialize length of previous LCP
int k = 0;

// Process all suffixes one by one starting from
// first suffix in txt[]
for (int i=0; i<n; i++)
{
    /* If the current suffix is at n-1, then we don't
       have next substring to consider. So lcp is not
       defined for this substring, we put zero. */
    if (invSuff[i] == n-1)
    {
        k = 0;
        continue;
    }

    /* j contains index of the next substring to
       be considered to compare with the present
       substring, i.e., next string in suffix array */
    int j = suffixArr[invSuff[i]+1];

    // Directly start matching from k'th index as
    // at-least k-1 characters will match
    while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
        k++;

    lcp[invSuff[i]] = k; // lcp for the present suffix.

    // Deleting the starting character from the string.
    if (k>0)
        k--;
}

// return the constructed lcp array
return lcp;
}

// Utility function to print an array
void printArr(vector<int>arr, int n)
{
```

```

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program
int main()
{
    string str = "banana";

    vector<int>suffixArr = buildSuffixArray(str, str.length());
    int n = suffixArr.size();

    cout << "Suffix Array : \n";
    printArr(suffixArr, n);

    vector<int>lcp = kasai(str, suffixArr);

    cout << "\nLCP Array : \n";
    printArr(lcp, n);
    return 0;
}

```

Output:

Suffix Array :  
5 3 1 0 4 2

LCP Array :  
1 3 0 0 2 0

**Illustration:**

txt[] = "banana", suffix[] = {5, 3, 1, 0, 4, 2}

Suffix array represents  
{"a", "ana", "anana", "banana", "na", "nana"}

Inverse Suffix Array would be  
invSuff[] = {3, 2, 5, 1, 4, 0}

LCP values are evaluated in below order

We first compute LCP of first suffix in text which is “**banana**“. We need next suffix in suffix array to compute LCP (Remember lcp[i] is defined as Longest Common Prefix of suffix[i] and suffix[i+1]). **To find the next suffix in suffixArr[], we use SuffInv[]**. The next suffix is “na”. Since there is no common prefix between “banana” and “na”, the value of LCP for “banana” is 0 and it is at index 3 in suffix array, so we fill **lcp[3]** as 0.

Next we compute LCP of second suffix which “**anana**”. Next suffix of “anana” in suffix array is “banana”. Since there is no common prefix, the value of LCP for “anana” is 0 and it is at index 2 in suffix array, so we fill **lcp[2]** as 0.

Next we compute LCP of third suffix which “**nana**”. Since there is no next suffix, the value of LCP for “nana” is not defined. We fill **lcp[5]** as 0.

Next suffix in text is “ana”. Next suffix of “**ana**” in suffix array is “anana”. Since there is a common prefix of length 3, the value of LCP for “ana” is 3. We fill **lcp[1]** as 3.

Now we lcp for next suffix in text which is “**na**”. This is where Kasai’s algorithm uses the trick that LCP value must be at least 2 because previous LCP value was 3. Since there is no character after “na”, final value of LCP is 2. We fill **lcp[4]** as 2.

Next suffix in text is “**a**”. LCP value must be at least 1 because previous value was 2. Since there is no character after “a”, final value of LCP is 1. We fill **lcp[0]** as 1.

We will soon be discussing implementation of search with the help of LCP array and how LCP array helps in reducing time complexity to  $O(m + \log n)$ .

#### References:

<http://web.stanford.edu/class/cs97si/suffix-array.pdf>

<http://www.mi.fu-berlin.de/wiki/pub/ABI/RnaSeqP4/suffix-array.pdf>

<http://codeforces.com/blog/entry/12796>

This article is contributed by **Prakhar Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<https://www.geeksforgeeks.org/%c2%ad%c2%adkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/>

## Chapter 37

# Search a Word in a 2D Grid of characters

Search a Word in a 2D Grid of characters - GeeksforGeeks

Given a 2D grid of characters and a word, find all occurrences of given word in grid. A word can be matched in all 8 directions at any point. Word is said 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 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
```

```
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
```



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

|   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | H | S | M | A | L | L | T | R | P | T | L | A |
| E | A | P | C | R | S | R | P | S | P | B | L | S |
| E | L | I | C | F | T | O | S | P | A | R | Q | H |
| N | I | H | D | E | T | S | E | R | I | U | V | C |
| N | B | C | D | W | U | S | J | J | I | Y | B | D |
| Y | M | A | E | S | Y | C | E | N | O | T | N | Y |
| P | I | E | T | G | N | L | N | G | T | D | S | J |
| P | S | C | U | D | U | E | G | C | A | A | G | G |
| O | T | G | G | C | B | W | U | W | J | E | J | S |
| I | Q | L | E | A | V | Q | K | Q | N | T | T | D |
| N | D | L | S | D | C | A | H | T | M | R | E | R |
| T | O | C | T | G | H | J | H | D | S | E | T | Y |
| M | G | M | I | J | R | T | Y | Y | U | I | O | P |

Source: Microsoft Interview Question

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 is C++ implementation of the same.

```
// C++ programs to search a word in a 2D grid
#include<bits/stdc++.h>
using namespace std;

// Rows and columns in given grid
#define R 3
#define C 14

// For searching in all 8 direction
int x[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
int y[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

// This function searches in all 8-direction from point
// (row, col) in grid[] []
```

```
bool search2D(char grid[R][C], int row, int col, string word)
{
    // If first character of word doesn't match with
    // given starting point in grid.
    if (grid[row][col] != word[0])
        return false;

    int len = word.length();

    // Search word in all 8 directions starting from (row,col)
    for (int dir = 0; dir < 8; dir++)
    {
        // Initialize starting point for current direction
        int k, rd = row + x[dir], cd = col + y[dir];

        // First character is already checked, match remaining
        // characters
        for (k = 1; k < len; k++)
        {
            // If out of bound break
            if (rd >= R || rd < 0 || cd >= C || cd < 0)
                break;

            // If not matched, break
            if (grid[rd][cd] != word[k])
                break;

            // Moving in particular direction
            rd += x[dir], cd += y[dir];
        }

        // If all character matched, then value of must
        // be equal to length of word
        if (k == len)
            return true;
    }
    return false;
}

// Searches given word in a given matrix in all 8 directions
void patternSearch(char grid[R][C], string word)
{
    // Consider every point as starting point and search
    // given word
    for (int row = 0; row < R; row++)
        for (int col = 0; col < C; col++)
            if (search2D(grid, row, col, word))
                cout << "pattern found at " << row << ", "
```

```
        << col << endl;
    }

    // Driver program
    int main()
    {
        char grid[R][C] = {"GEEKSFORGEEKS",
                           "GEEKSQUIZGEEK",
                           "IDEQAPRACTICE"
                           };

        patternSearch(grid, "GEEKS");
        cout << endl;
        patternSearch(grid, "EEE");
        return 0;
    }
```

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
```

**Exercise:** The above solution only print locations of word. Extend it to print the direction where word is present.

See [this](#) for solution of exercise.

This article is contributed by **Utkarsh Trivedi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/search-a-word-in-a-2d-grid-of-characters/>

## Chapter 38

# Z algorithm (Linear time pattern searching Algorithm)

Z algorithm (Linear time pattern searching Algorithm) - GeeksforGeeks

This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be  $n$  and of pattern be  $m$ , then total time taken is  $O(m + n)$  with linear space complexity. Now we can see that both time and space complexity is same as KMP algorithm but this algorithm is simpler to understand.

In this algorithm, we construct a Z array.

### What is Z Array?

For a string  $str[0..n-1]$ , Z array is of same length as string. An element  $Z[i]$  of Z array stores length of the longest substring starting from  $str[i]$  which is also a prefix of  $str[0..n-1]$ . The first entry of Z array is meaningless as complete string is always prefix of itself.

Example:

|          |   |   |   |   |   |   |   |   |   |   |    |    |
|----------|---|---|---|---|---|---|---|---|---|---|----|----|
| Index    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Text     | a | a | b | c | a | a | b | x | a | a | a  | z  |
| Z values | X | 1 | 0 | 0 | 3 | 1 | 0 | 0 | 2 | 2 | 1  | 0  |

More Examples:

`str = "aaaaaa"`

`Z[] = {x, 5, 4, 3, 2, 1}`

`str = "aabaacd"`

`Z[] = {x, 1, 0, 2, 1, 0, 0}`

`str = "abababab"`

`Z[] = {x, 0, 6, 0, 4, 0, 2, 0}`

### How is Z array helpful in Searching Pattern in Linear time?

The idea is to concatenate pattern and text, and create a string "P\$T" where P is pattern, \$ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.

Example:

Pattern P = "aab", Text T = "baabaa"

The concatenated string is = "aab\$baabaa"

Z array for above concatenated string is {x, 1, 0, 0, 0,  
3, 1, 0, 2, 1}.

Since length of pattern is 3, the value 3 in Z array indicates presence of pattern.

### How to construct Z array?

A Simple Solution is to run two nested loops, the outer loop goes to every index and the inner loop finds length of the longest prefix that matches substring starting at current index. The time complexity of this solution is  $O(n^2)$ .

We can construct Z array in linear time.

The idea is to maintain an interval [L, R] which is the interval with max R such that [L,R] is prefix substring (substring which is also prefix).

Steps for maintaining this interval are as follows -

- 1) If  $i > R$  then there is no prefix substring that starts before i and ends after i, so we reset L and R and compute new [L,R] by comparing  $\text{str}[0..]$  to  $\text{str}[i..]$  and get  $Z[i]$  ( $= R-L+1$ ).
- 2) If  $i \leq R$  then let  $K = i-L$ , now  $Z[i] \geq \min(Z[K], R-i+1)$  because  $\text{str}[i..]$  matches with  $\text{str}[K..]$  for at least  $R-i+1$  characters (they are in [L,R] interval which we know is a prefix substring).  
Now two sub cases arise -
  - a) If  $Z[K] < R-i+1$  then there is no prefix substring starting at  $\text{str}[i]$  (otherwise  $Z[K]$  would be larger) so  $Z[i] = Z[K]$  and interval [L,R] remains same.
  - b) If  $Z[K] \geq R-i+1$  then it is possible to extend the [L,R] interval thus we will set L as i and start matching from  $\text{str}[R]$  onwards and get new R then we will update interval [L,R] and calculate  $Z[i]$  ( $= R-L+1$ ).

For better understanding of above step by step procedure please check this animation - <http://www.utdallas.edu/~besp/demo/John2010/z-algorithm.htm>

The algorithm runs in linear time because we never compare character less than R and with matching we increase R by one so there are at most T comparisons. In mismatch case, mismatch happen only once for each i (because of which R stops), that's another at most T comparison making overall linear complexity.

Below is the implementation of Z algorithm for pattern searching.

C++

```
// A C++ program that implements Z algorithm for pattern searching
#include<iostream>
using namespace std;

void getZarr(string str, int Z[]);

// prints all occurrences of pattern in text using Z algo
void search(string text, string pattern)
{
    // Create concatenated string "P$T"
    string concat = pattern + "$" + text;
    int l = concat.length();

    // Construct Z array
    int Z[l];
    getZarr(concat, Z);

    // now looping through Z array for matching condition
    for (int i = 0; i < l; ++i)
    {
        // if Z[i] (matched region) is equal to pattern
        // length we got the pattern
        if (Z[i] == pattern.length())
            cout << "Pattern found at index "
                 << i - pattern.length() - 1 << endl;
    }
}

// Fills Z array for given string str[]
void getZarr(string str, int Z[])
{
    int n = str.length();
    int L, R, k;

    // [L,R] make a window which matches with prefix of s
    L = R = 0;
    for (int i = 1; i < n; ++i)
    {
        // if i>R nothing matches so we will calculate.
        // Z[i] using naive way.

```

```

    if (i > R)
    {
        L = R = i;

        // R-L = 0 in starting, so it will start
        // checking from 0'th index. For example,
        // for "ababab" and i = 1, the value of R
        // remains 0 and Z[i] becomes 0. For string
        // "aaaaaa" and i = 1, Z[i] and R become 5
        while (R < n && str[R-L] == str[R])
            R++;
        Z[i] = R-L;
        R--;
    }
    else
    {
        // k = i-L so k corresponds to number which
        // matches in [L,R] interval.
        k = i-L;

        // if Z[k] is less than remaining interval
        // then Z[i] will be equal to Z[k].
        // For example, str = "ababab", i = 3, R = 5
        // and L = 2
        if (Z[k] < R-i+1)
            Z[i] = Z[k];

        // For example str = "aaaaaa" and i = 2, R is 5,
        // L is 0
        else
        {
            // else start from R and check manually
            L = i;
            while (R < n && str[R-L] == str[R])
                R++;
            Z[i] = R-L;
            R--;
        }
    }
}

// Driver program
int main()
{
    string text = "GEEKS FOR GEEKS";
    string pattern = "GEEK";
    search(text, pattern);
}

```

```
    return 0;
}
```

#### Java

```
// A Java program that implements Z algorithm for pattern
// searching
class GFG {

    // prints all occurrences of pattern in text using
    // Z algo
    public static void search(String text, String pattern)
    {

        // Create concatenated string "P$T"
        String concat = pattern + "$" + text;

        int l = concat.length();

        int Z[] = new int[l];

        // Construct Z array
        getZarr(concat, Z);

        // now looping through Z array for matching condition
        for(int i = 0; i < l; ++i){

            // if Z[i] (matched region) is equal to pattern
            // length we got the pattern

            if(Z[i] == pattern.length()){
                System.out.println("Pattern found at index "
                                   + (i - pattern.length() - 1));
            }
        }
    }

    // Fills Z array for given string str[]
    private static void getZarr(String str, int[] Z) {

        int n = str.length();

        // [L,R] make a window which matches with
        // prefix of s
        int L = 0, R = 0;

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



```

// if i>R nothing matches so we will calculate.
// Z[i] using naive way.
if(i > R){

    L = R = i;

    // R-L = 0 in starting, so it will start
    // checking from 0'th index. For example,
    // for "ababab" and i = 1, the value of R
    // remains 0 and Z[i] becomes 0. For string
    // "aaaaaa" and i = 1, Z[i] and R become 5

    while(R < n && str.charAt(R - L) == str.charAt(R))
        R++;

    Z[i] = R - L;
    R--;

}
else{

    // k = i-L so k corresponds to number which
    // matches in [L,R] interval.
    int k = i - L;

    // if Z[k] is less than remaining interval
    // then Z[i] will be equal to Z[k].
    // For example, str = "ababab", i = 3, R = 5
    // and L = 2
    if(Z[k] < R - i + 1)
        Z[i] = Z[k];

    // For example str = "aaaaaa" and i = 2, R is 5,
    // L is 0
    else{

        // else start from R and check manually
        L = i;
        while(R < n && str.charAt(R - L) == str.charAt(R))
            R++;

        Z[i] = R - L;
        R--;

    }
}
}
}

```

```
// Driver program
public static void main(String[] args)
{
    String text = "GEEKS FOR GEEKS";
    String pattern = "GEEK";

    search(text, pattern);
}

// This code is contributed by PavanKoli.
```

Output:

```
Pattern found at index 0
Pattern found at index 10
```

This article is contributed by [Utkarsh Trivedi](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [pkoli](#)

**Source**

<https://www.geeksforgeeks.org/z-algorithm-linear-time-pattern-searching-algorithm/>

## Chapter 39

# Online algorithm for checking palindrome in a stream

Online algorithm for checking palindrome in a stream - GeeksforGeeks

Given a stream of characters (characters are received one by one), write a function that prints 'Yes' if a character makes the complete string palindrome, else prints 'No'.

Examples:

```
Input: str[] = "abcba"
Output: a Yes    // "a" is palindrome
        b No     // "ab" is not palindrome
        c No     // "abc" is not palindrome
        b No     // "abcb" is not palindrome
        a Yes    // "abcba" is palindrome
```

```
Input: str[] = "aabaacaabaa"
Output: a Yes    // "a" is palindrome
        a Yes    // "aa" is palindrome
        b No     // "aab" is not palindrome
        a No     // "aaba" is not palindrome
        a Yes    // "aabaa" is palindrome
        c No     // "aabaac" is not palindrome
        a No     // "aabaaca" is not palindrome
        a No     // "aabaacaa" is not palindrome
        b No     // "aabaacaab" is not palindrome
        a No     // "aabaacaaba" is not palindrome
        a Yes    // "aabaacaabaa" is palindrome
```

Let input string be `str[0..n-1]`. A **Simple Solution** is to do following for every character `str[i]` in input string. Check if substring `str[0...i]` is palindrome, then print yes, else print no.

A **Better Solution** is to use the idea of Rolling Hash used in [Rabin Karp algorithm](#). The idea is to keep track of reverse of first half and second half (we also use first half and reverse of second half) for every index. Below is complete algorithm.

- 1) The first character is always a palindrome, so print yes for first character.
- 2) Initialize reverse of first half as "a" and second half as "b".  
Let the hash value of first half reverse be 'first' and that of second half be 'second'.
- 3) Iterate through string starting from second character, do following for every character str[i], i.e., i varies from 1 to n-1.
  - a) If 'first' and 'second' are same, then character by character check the substring ending with current character and print "Yes" if palindrome.  
Note that if hash values match, then strings need not be same. For example, hash values of "ab" and "ba" are same, but strings are different. That is why we check complete string after hash.
  - b) Update 'first' and 'second' for next iteration.  
If 'i' is even, then add next character to the beginning of 'first' and end of second half and update hash values.  
If 'i' is odd, then keep 'first' as it is, remove leading character from second and append a next character at end.

Let us see all steps for example string "abcba"

Initial values of 'first' and 'second'

first = hash("a"), second = hash("b")

Start from second character, i.e.,

i = 1

- a) Compare 'first' and 'second', they don't match, so print no.
- b) Calculate hash values for next iteration, i.e., i = 2  
Since i is odd, 'first' is not changed and 'second' becomes hash("c")

i = 2

- a) Compare 'first' and 'second', they don't match, so print no.
- b) Calculate hash values for next iteration, i.e., i = 3  
Since i is even, 'first' becomes hash("ba") and 'second' becomes hash("cb")

i = 3

- a) Compare 'first' and 'second', they don't match, so print no.
- b) Calculate hash values for next iteration, i.e., i = 4  
Since i is odd, 'first' is not changed and 'second' becomes hash("ba")

i = 4

- a) 'first' and 'second' match, compare the whole strings, they match, so print yes
- b) We don't need to calculate next hash values as this is last index

The idea of using rolling hashes is, next hash value can be calculated from previous in  $O(1)$  time by just doing some constant number of arithmetic operations.

Below are the implementations of above approach.

C/C++

```
// C program for online algorithm for palindrome checking
#include<stdio.h>
#include<string.h>

// d is the number of characters in input alphabet
#define d 256

// q is a prime number used for evaluating Rabin Karp's Rolling hash
#define q 103

void checkPalindromes(char str[])
{
    // Length of input string
    int N = strlen(str);

    // A single character is always a palindrome
    printf("%c Yes\n", str[0]);

    // Return if string has only one character
    if (N == 1) return;

    // Initialize first half reverse and second half for
    // as first and second characters
    int first = str[0] % q;
    int second = str[1] % q;

    int h = 1, i, j;

    // Now check for palindromes from second character
    // onward
    for (i=1; i<N; i++)
    {
        // If the hash values of 'first' and 'second'
        // match, then only check individual characters
        if (first == second)
        {
            /* Check if str[0..i] is palindrome using
```

```
        simple character by character match */
    for (j = 0; j < i/2; j++)
    {
        if (str[j] != str[i-j])
            break;
    }
    (j == i/2)? printf("%c Yes\n", str[i]):
    printf("%c No\n", str[i]);
}
else printf("%c No\n", str[i]);

// Calculate hash values for next iteration.
// Don't calculate hash for next characters if
// this is the last character of string
if (i != N-1)
{
    // If i is even (next i is odd)
    if (i%2 == 0)
    {
        // Add next character after first half at beginning
        // of 'firststr'
        h = (h*d) % q;
        firststr = (firststr + h*str[i/2])%q;

        // Add next character after second half at the end
        // of second half.
        second = (second*d + str[i+1])%q;
    }
    else
    {
        // If next i is odd (next i is even) then we
        // need not to change firststr, we need to remove
        // first character of second and append a
        // character to it.
        second = (d*(second + q - str[(i+1)/2]*h)%q
                + str[i+1])%q;
    }
}
}

}

/* Driver program to test above function */
int main()
{
    char *txt = "aabaacaabaa";
    checkPalindromes(txt);
    getchar();
    return 0;
}
```

```
}
```

#### Java

```
// Java program for online algorithm for
// palindrome checking
public class GFG
{
    // d is the number of characters in
    // input alphabet
    static final int d = 256;

    // q is a prime number used for
    // evaluating Rabin Karp's Rolling hash
    static final int q = 103;

    static void checkPalindromes(String str)
    {
        // Length of input string
        int N = str.length();

        // A single character is always a palindrome
        System.out.println(str.charAt(0)+" Yes");

        // Return if string has only one character
        if (N == 1) return;

        // Initialize first half reverse and second
        // half for as first and second characters
        int first = str.charAt(0) % q;
        int second = str.charAt(1) % q;

        int h = 1, i, j;

        // Now check for palindromes from second
        // character onward
        for (i = 1; i < N; i++)
        {
            // If the hash values of 'first' and
            // 'second' match, then only check
            // individual characters
            if (first == second)
            {
                /* Check if str[0..i] is palindrome
                using simple character by character
                match */
                for (j = 0; j < i/2; j++)
                {
```

```
        if (str.charAt(j) != str.charAt(i
                                - j))
            break;
    }
    System.out.println((j == i/2) ?
        str.charAt(i) + " Yes": str.charAt(i)+
        " No");
}
else System.out.println(str.charAt(i)+ " No");

// Calculate hash values for next iteration.
// Don't calculate hash for next characters
// if this is the last character of string
if (i != N - 1)
{
    // If i is even (next i is odd)
    if (i % 2 == 0)
    {
        // Add next character after first
        // half at beginning of 'firststr'
        h = (h * d) % q;
        firststr = (firststr + h *str.charAt(i /
                                2)) % q;

        // Add next character after second
        // half at the end of second half.
        second = (second * d + str.charAt(i +
                                1)) % q;
    }
    else
    {
        // If next i is odd (next i is even)
        // then we need not to change firststr,
        // we need to remove first character
        // of second and append a character
        // to it.
        second = (d * (second + q - str.charAt(
                                (i + 1) / 2) * h) % q +
                                str.charAt(i + 1)) % q;
    }
}
}

}

/* Driver program to test above function */
public static void main(String args[])
{
    String txt = "aabaacaabaa";
```



```
        checkPalindromes(txt);
    }
}
// This code is contributed by Sumit Ghosh
```

## Python

```
# Python program Online algorithm for checking palindrome
# in a stream

# d is the number of characters in input alphabet
d = 256

# q is a prime number used for evaluating Rabin Karp's
# Rolling hash
q = 103

def checkPalindromes(string):

    # Length of input string
    N = len(string)

    # A single character is always a palindrome
    print string[0] + " Yes"

    # Return if string has only one character
    if N == 1:
        return

    # Initialize first half reverse and second half for
    # as firststr and second characters
    firststr = ord(string[0]) % q
    second = ord(string[1]) % q

    h = 1
    i = 0
    j = 0

    # Now check for palindromes from second character
    # onward
    for i in xrange(1,N):

        # If the hash values of 'firststr' and 'second'
        # match, then only check individual characters
        if firststr == second:

            # Check if str[0..i] is palindrome using
            # simple character by character match
```

```
    for j in xrange(0,i/2):
        if string[j] != string[i-j]:
            break
    j += 1
    if j == i/2:
        print string[i] + " Yes"
    else:
        print string[i] + " No"
else:
    print string[i] + " No"

# Calculate hash values for next iteration.
# Don't calculate hash for next characters if
# this is the last character of string
if i != N-1:

    # If i is even (next i is odd)
    if i % 2 == 0:

        # Add next character after first half at
        # beginning of 'firststr'
        h = (h*d) % q
        firststr = (firststr + h*ord(string[i/2]))%q

        # Add next character after second half at
        # the end of second half.
        second = (second*d + ord(string[i+1]))%q
    else:
        # If next i is odd (next i is even) then we
        # need not to change firststr, we need to remove
        # first character of second and append a
        # character to it.
        second = (d*(second + q - ord(string[(i+1)/2])*h)%q
                  + ord(string[i+1]))%q

# Driver program
txt = "aabaacaabaa"
checkPalindromes(txt)
# This code is contributed by Bhavya Jain
```

Output:

```
a Yes
a Yes
b No
a No
a Yes
```

c No  
a No  
a No  
b No  
a No  
a Yes

The worst case time complexity of the above solution remains  $O(n*n)$ , but in general, it works much better than simple approach as we avoid complete substring comparison most of the time by first comparing hash values. The worst case occurs for input strings with all same characters like “aaaaaa”.

This article is contributed by **Ajay**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/online-algorithm-for-checking-palindrome-in-a-stream/>

## Chapter 40

# Suffix Tree Application 6 – Longest Palindromic Substring

Suffix Tree Application 6 - Longest Palindromic Substring - GeeksforGeeks

Given a string, find the longest substring which is palindrome.

We have already discussed Naïve [ $O(n^3)$ ], quadratic [ $O(n^2)$ ] and linear [ $O(n)$ ] approaches in [Set 1](#), [Set 2](#) and [Manacher's Algorithm](#).

In this article, we will discuss another linear time approach based on suffix tree.

If given string is S, then approach is following:

- Reverse the string S (say reversed string is R)
- Get [Longest Common Substring](#) of S and R **given that LCS in S and R must be from same position in S**

Can you see why we say that **LCS in R and S must be from same position in S** ?

Let's look at following examples:

- For S = *xababayz* and R = *zyababax*, LCS and LPS both are *ababa* (SAME)
- For S = *abacdfgdcaba* and R = *abacdghdcaba*, LCS is *abacd* and LPS is *aba* (DIFFERENT)
- For S = *pqrqpabdcfgdcba* and R = *abdcgfdcbapqrqp*, LCS and LPS both are *pqrqp* (SAME)
- For S = *pqqpabdcfghfdcba* and R = *abdcfhgfdcbapqqp*, LCS is *abcdf* and LPS is *pqqp* (DIFFERENT)

We can see that LCS and LPS are not same always. When they are different ?

*When S has a reversed copy of a non-palindromic substring in it which is of same or longer length than LPS in S, then LCS and LPS will be different.*

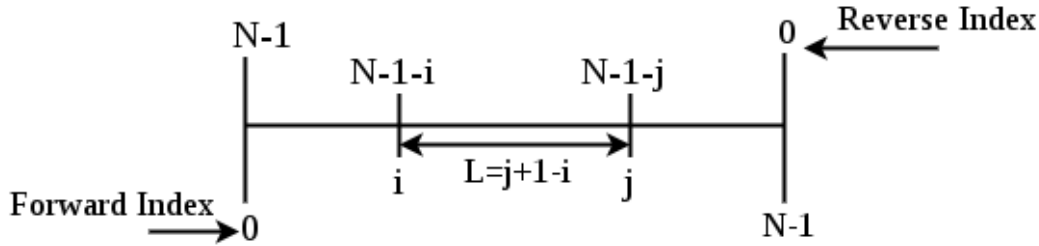
In 2<sup>nd</sup> example above (S = *abacdfgdcaba*), for substring *abacd*, there exists a reverse copy

*dcaba* in *S*, which is of longer length than LPS *aba* and so LPS and LCS are different here. Same is the scenario in 4<sup>th</sup> example.

To handle this scenario we say that LPS in *S* is same as LCS in *S* and *R* **given that LCS in *R* and *S* must be from same position in *S*.**

If we look at 2<sup>nd</sup> example again, substring *aba* in *R* comes from exactly same position in *S* as substring *aba* in *S* which is ZERO (0<sup>th</sup> index) and so this is LPS.

**The Position Constraint:**



We will refer string *S* index as forward index ( $S_i$ ) and string *R* index as reverse index ( $R_i$ ). Based on above figure, a character with index *i* (forward index) in a string *S* of length *N*, will be at index  $N-1-i$  (reverse index) in it's reversed string *R*.

If we take a substring of length *L* in string *S* with starting index *i* and ending index *j* ( $j = i+L-1$ ), then in it's reversed string *R*, the reversed substring of the same will start at index  $N-1-j$  and will end at index  $N-1-i$ .

If there is a common substring of length *L* at indices  $S_i$  (forward index) and  $R_i$  (reverse index) in *S* and *R*, then these will come from same position in *S* if  $R_i = (N - 1) - (S_i + L - 1)$  where *N* is string length.

So to find LPS of string *S*, we find longest common string of *S* and *R* where both substrings satisfy above constraint, i.e. if substring in *S* is at index  $S_i$ , then same substring should be in *R* at index  $(N - 1) - (S_i + L - 1)$ . If this is not the case, then this substring is not LPS candidate.

Naive [ $O(N \cdot M^2)$ ] and Dynamic Programming [ $O(N \cdot M)$ ] approaches to find LCS of two strings are already discussed [here](#) which can be extended to add position constraint to give LPS of a given string.

Now we will discuss suffix tree approach which is nothing but an extension to [Suffix Tree LCS approach](#) where we will add the position constraint.

While finding LCS of two strings *X* and *Y*, we just take deepest node marked as *XY* (i.e. the node which has suffixes from both strings as it's children).

While finding LPS of string *S*, we will again find LCS of *S* and *R* with a condition that the common substring should satisfy the position constraint (the common substring should come from same position in *S*). To verify position constraint, we need to know all forward and reverse indices on each internal node (i.e. the suffix indices of all leaf children below the internal nodes).

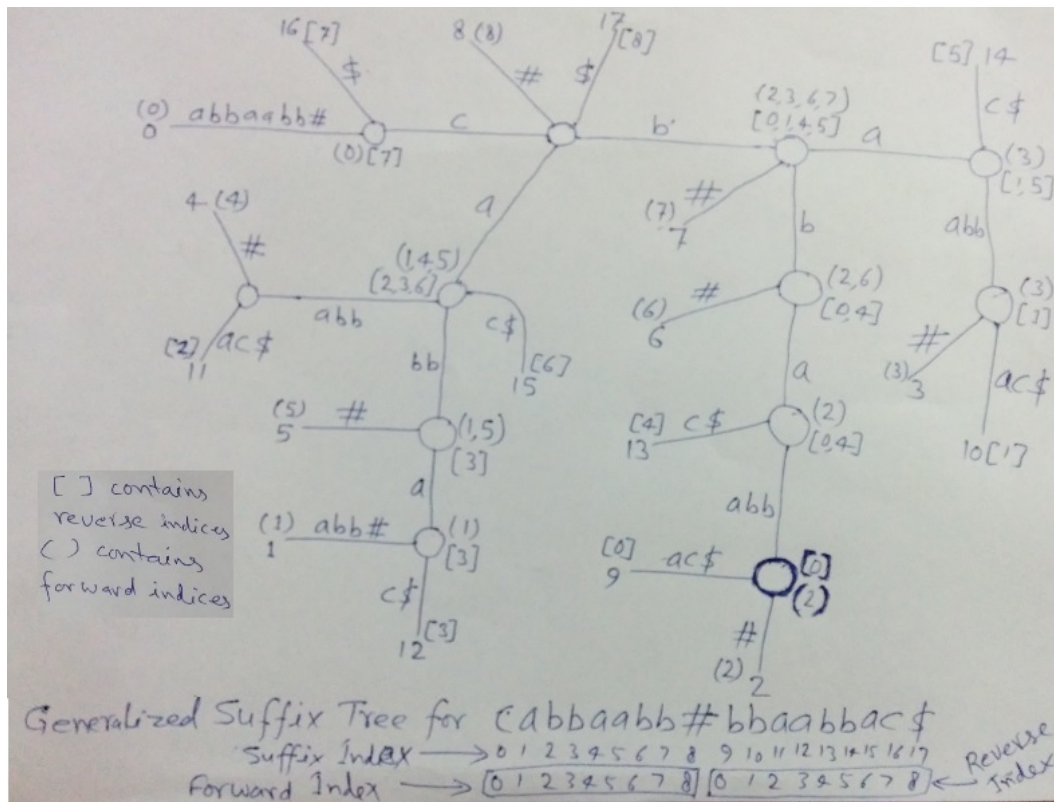
In [Generalized Suffix Tree](#) of  $S\#R\$, a substring on the path from root to an internal node is a common substring if the internal node has suffixes from both strings *S* and *R*. The index of the common substring in *S* and *R* can be found by looking at suffix index at respective leaf node.$

If string  $S\#$  is of length *N* then:

- If suffix index of a leaf is less than  $N$ , then that suffix belongs to  $S$  and same suffix index will become forward index of all ancestor nodes
- If suffix index of a leaf is greater than  $N$ , then that suffix belongs to  $R$  and reverse index for all ancestor nodes will be  $N - \text{suffix index}$

Let's take string  $S = \text{cabbaabb}$ . The figure below is **Generalized Suffix Tree** for  $\text{cabbaabb}\#\text{bbaabbac}\$$  where we have shown forward and reverse indices of all children suffixes on all internal nodes (except root).

Forward indices are in Parentheses ( ) and reverse indices are in square bracket [ ].



In above figure, all leaf nodes will have one forward or reverse index depending on which string ( $S$  or  $R$ ) they belong to. Then children's forward or reverse indices propagate to the parent.

Look at the figure to understand what would be the forward or reverse index on a leaf with a given suffix index. At the bottom of figure, it is shown that leaves with suffix indices from 0 to 8 will get same values (0 to 8) as their forward index in  $S$  and leaves with suffix indices 9 to 17 will get reverse index in  $R$  from 0 to 8.

For example, the highlighted internal node has two children with suffix indices 2 and 9. Leaf with suffix index 2 is from position 2 in  $S$  and so its forward index is 2 and shown in ( ). Leaf with suffix index 9 is from position 0 in  $R$  and so its reverse index is 0 and shown in [ ]. These indices propagate to parent and the parent has one leaf with suffix index 14 for which reverse index is 4. So on this parent node forward index is (2) and reverse index is [0,4].

And in same way, we should be able to understand the how forward and reverse indices are calculated on all nodes.

In above figure, all internal nodes have suffixes from both strings S and R, i.e. all of them represent a common substring on the path from root to themselves. Now we need to find deepest node satisfying position constraint. For this, we need to check if there is a forward index  $S_i$  on a node, then there must be a reverse index  $R_i$  with value  $(N - 2) - (S_i + L - 1)$  where N is length of string  $S\#$  and L is node depth (or substring length). If yes, then consider this node as a LPS candidate, else ignore it. In above figure, deepest node is highlighted which represents LPS as bbaabb.

We have not shown forward and reverse indices on root node in figure. Because root node itself doesn't represent any common substring (In code implementation also, forward and reverse indices will not be calculated on root node)

How to implement this approach to find LPS? Here are the things that we need:

- We need to know forward and reverse indices on each node.
- For a given forward index  $S_i$  on an internal node, we need know if reverse index  $R_i = (N - 2) - (S_i + L - 1)$  also present on same node.
- Keep track of deepest internal node satisfying above condition.

One way to do above is:

While DFS on suffix tree, we can store forward and reverse indices on each node in some way (storage will help to avoid repeated traversals on tree when we need to know forward and reverse indices on a node). Later on, we can do another DFS to look for nodes satisfying position constraint. For position constraint check, we need to search in list of indices.

What data structure is suitable here to do all these in quickest way ?

- If we store indices in array, it will require linear search which will make overall approach non-linear in time.
- If we store indices in tree (set in C++, TreeSet in Java), we may use binary search but still overall approach will be non-linear in time.
- If we store indices in hash function based set (unordered\_set in C++, HashSet in Java), it will provide a constant search on average and this will make overall approach linear in time. *A hash function based set may take more space depending on values being stored.*

We will use two unordered\_set (one for forward and other from reverse indices) in our implementation, added as a member variable in SuffixTreeNode structure.

```
// A C++ program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for given string S
// and it's reverse R, then we find
// longest palindromic substring of given string S
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <iostream>
#include <unordered_set>
#define MAX_CHAR 256
using namespace std;

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. Lets say there are two nodes 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;

    //To store indices of children suffixes in given string
    unordered_set<int> *forwardIndices;

    //To store indices of children suffixes in reversed string
    unordered_set<int> *reverseIndices;
};

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 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
int size1 = 0; //Size of 1st string
int reverseIndex; //Index of a suffix in reversed string
unordered_set<int>::iterator forwardIndex;

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;
    node->forwardIndices = new unordered_set<int>;
    node->reverseIndices = new unordered_set<int>;
    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 existng 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])
    {
        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in 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 && text[k] != '#'; k++)
printf("%c", text[k]);
if(k<=j)
printf("#");
}
```

```
//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(n != root)
            {
                //Add children's suffix indices in parent
                n->forwardIndices->insert(
                    n->children[i]->forwardIndices->begin(),
                    n->children[i]->forwardIndices->end());
                n->reverseIndices->insert(
                    n->children[i]->reverseIndices->begin(),
                    n->children[i]->reverseIndices->end());
            }
        }
    }
    if (leaf == 1)
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#')
            {
                n->end = (int*) malloc(sizeof(int));
            }
        }
    }
}
```

```
        *(n->end) = i;
    }
}
n->suffixIndex = size - labelHeight;

if(n->suffixIndex < size1) //Suffix of Given String
    n->forwardIndices->insert(n->suffixIndex);
else //Suffix of Reversed String
    n->reverseIndices->insert(n->suffixIndex - size1);

//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);
    }

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);

                if(*maxHeight < labelHeight
                    && n->forwardIndices->size() > 0 &&
                    n->reverseIndices->size() > 0)
                {
                    for (forwardIndex=n->forwardIndices->begin();
                        forwardIndex!=n->forwardIndices->end();
                        ++forwardIndex)
                    {
                        reverseIndex = (size1 - 2) -
                            (*forwardIndex + labelHeight - 1);
                        //If reverse suffix comes from
                        //SAME position in given string
                        //Keep track of deepest node
                        if(n->reverseIndices->find(reverseIndex) !=
                            n->reverseIndices->end())
                        {
                            *maxHeight = labelHeight;
                            *substringStartIndex = *(n->end) -
                                labelHeight + 1;
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

```

    }
}

void getLongestPalindromicSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No palindromic substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 9;
    printf("Longest Palindromic Substring in cabbaabb is: ");
    strcpy(text, "cabbaabb#bbaabbac$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 17;
    printf("Longest Palindromic Substring in forgeeksskeegfor is: ");
    strcpy(text, "forgeeksskeegfor#rofgeeksskeegrof$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abcde is: ");
    strcpy(text, "abcde#edcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 7;
    printf("Longest Palindromic Substring in abcdae is: ");
    strcpy(text, "abcdae#eadcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
}

```



```
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in abacd is: ");
strcpy(text, "abacd#dcaba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in abcdc is: ");
strcpy(text, "abcdc#cdcba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 13;
printf("Longest Palindromic Substring in abacdfgdcaba is: ");
strcpy(text, "abacdfgdcaba#abacdfgdcaba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 15;
printf("Longest Palindromic Substring in xyabacdfgdcaba is: ");
strcpy(text, "xyabacdfgdcaba#abacdfgdcabayx$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 9;
printf("Longest Palindromic Substring in xababayz is: ");
strcpy(text, "xababayz#zyababax$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in xabax is: ");
strcpy(text, "xabax#xabax$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}
```

Output:

```
Longest Palindromic Substring in cabbaabb is: bbaabb, of length: 6
Longest Palindromic Substring in forgeeksskeegfor is: geeksskeeg, of length: 10
Longest Palindromic Substring in abcde is: a, of length: 1
Longest Palindromic Substring in abcdae is: a, of length: 1
Longest Palindromic Substring in abacd is: aba, of length: 3
Longest Palindromic Substring in abcdc is: cdc, of length: 3
Longest Palindromic Substring in abacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xyabacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xababayz is: ababa, of length: 5
Longest Palindromic Substring in xabax is: xabax, of length: 5
```

#### Followup:

Detect ALL palindromes in a given string.

e.g. For string abcdccbefgf, all possible palindromes are a, b, c, d, e, f, g, dd, fgf, cddc, bcdccb.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Generalized Suffix Tree 1](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<https://www.geeksforgeeks.org/suffix-tree-application-6-longest-palindromic-substring/>

## Chapter 41

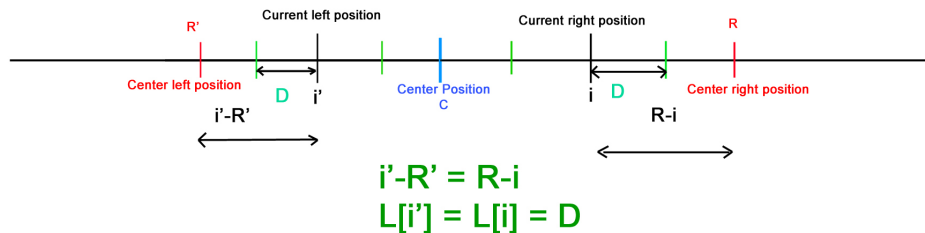
# Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 4

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 4 - Geeks-forGeeks

In Manacher's Algorithm [Part 1](#) and [Part 2](#), we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. In [Part 3](#), we implemented the same.

Here we will review the four cases again and try to see it differently and implement the same.

All four cases depends on LPS length value at currentLeftPosition ( $L[iMirror]$ ) and value of ( $centerRightPosition - currentRightPosition$ ), i.e.  $(R - i)$ . These two information are know before which helps us to reuse previous available information and avoid unnecessary character comparison.



If we look at all four cases, we will see that we 1<sup>st</sup> set minimum of  $L[iMirror]$  and  $R - i$  to  $L[i]$  and then we try to expand the palindrome in whichever case it can expand.

Above observation may look more intuitive, easier to understand and implement, given that one understands LPS length array, position, index, symmetry property etc.

## C/C++

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
int min(int a, int b)
{
    int res = a;
    if(b < a)
        res = b;
    return res;
}

void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;
    int diff = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
        //get currentLeftPosition iMirror for currentRightPosition i
        iMirror = 2*C-i;
        L[i] = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
            L[i] = min(L[iMirror], diff);

        //Attempt to expand palindrome centered at currentRightPosition i
        //Here for odd positions, we compare characters and
```

```

//if match then increment LPS Length by ONE
//If even position, we just increment LPS by ONE without
//any character comparison
while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
        ( ((i + L[i] + 1) % 2 == 0) ||
          (text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2] )))
{
    L[i]++;
}

if(L[i] > maxLPSLength) // Track maxLPSLength
{
    maxLPSLength = L[i];
    maxLPSCenterPosition = i;
}

//If palindrome centered at currentRightPosition i
//expand beyond centerRightPosition R,
//adjust centerPosition C based on expanded palindrome.
if (i + L[i] > R)
{
    C = i;
    R = i + L[i];
}
//Uncomment it to print LPS Length array
//printf("%d ", L[i]);
}
//printf("\n");
start = (maxLPSCenterPosition - maxLPSLength)/2;
end = start + maxLPSLength - 1;
printf("LPS of string is %s : ", text);
for(i=start; i<=end; i++)
    printf("%c", text[i]);
printf("\n");
}

int main(int argc, char *argv[])
{
    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
    findLongestPalindromicString();
}

```

```
strcpy(text, "abcbabcbabcba");
findLongestPalindromicString();

strcpy(text, "forgeeksskeegfor");
findLongestPalindromicString();

strcpy(text, "caba");
findLongestPalindromicString();

strcpy(text, "abacdfgdcaba");
findLongestPalindromicString();

strcpy(text, "abacdfgdcabba");
findLongestPalindromicString();

strcpy(text, "abacdedcaba");
findLongestPalindromicString();

return 0;
}
```

## Python

```
# Python program to implement Manacher's Algorithm

def findLongestPalindromicString(text):
    N = len(text)
    if N == 0:
        return
    N = 2*N+1    # Position count
    L = [0] * N
    L[0] = 0
    L[1] = 1
    C = 1    # centerPosition
    R = 2    # centerRightPosition
    i = 0    # currentRightPosition
    iMirror = 0    # currentLeftPosition
    maxLPSLength = 0
    maxLPSCenterPosition = 0
    start = -1
    end = -1
    diff = -1

    # Uncomment it to print LPS Length array
    # printf("%d %d ", L[0], L[1]);
    for i in xrange(2,N):

        # get currentLeftPosition iMirror for currentRightPosition i
```

```

iMirror = 2*C-i
L[i] = 0
diff = R - i
# If currentRightPosition i is within centerRightPosition R
if diff > 0:
    L[i] = min(L[iMirror], diff)

# Attempt to expand palindrome centered at currentRightPosition i
# Here for odd positions, we compare characters and
# if match then increment LPS Length by ONE
# If even position, we just increment LPS by ONE without
# any character comparison
try:
    while ((i + L[i]) < N and (i - L[i]) > 0) and \
          (((i + L[i] + 1) % 2 == 0) or \
           (text[(i + L[i] + 1) / 2] == text[(i - L[i] - 1) / 2]]):
        L[i]+=1
except Exception as e:
    pass

if L[i] > maxLPSLength:      # Track maxLPSLength
    maxLPSLength = L[i]
    maxLPSCenterPosition = i

# If palindrome centered at currentRightPosition i
# expand beyond centerRightPosition R,
# adjust centerPosition C based on expanded palindrome.
if i + L[i] > R:
    C = i
    R = i + L[i]

# Uncomment it to print LPS Length array
# printf("%d ", L[i]);
start = (maxLPSCenterPosition - maxLPSLength) / 2
end = start + maxLPSLength - 1
print "LPS of string is " + text + " : ",
print text[start:end+1],
print "\n",

# Driver program
text1 = "babcbabcbaccba"
findLongestPalindromicString(text1)

text2 = "abaaba"
findLongestPalindromicString(text2)

text3 = "abababa"
findLongestPalindromicString(text3)

```

```
text4 = "abcbabcbabcba"
findLongestPalindromicString(text4)

text5 = "forgeeksskeegfor"
findLongestPalindromicString(text5)

text6 = "caba"
findLongestPalindromicString(text6)

text7 = "abacdfgdcaba"
findLongestPalindromicString(text7)

text8 = "abacdfgdcabba"
findLongestPalindromicString(text8)

text9 = "abacdedcaba"
findLongestPalindromicString(text9)

# This code is contributed by BHAVYA JAIN
```

Output:

```
LPS of string is babcbabcbaccba : abcbabcb
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabcba : abcbabcbabcba
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdfgdcaba : aba
LPS of string is abacdfgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba
```

### Other Approaches

We have discussed two approaches here. One in [Part 3](#) and other in current article. In both approaches, we worked on given string. Here we had to handle even and odd positions differently while comparing characters for expansion (because even positions do not represent any character in string).

To avoid this different handling of even and odd positions, we need to make even positions also to represent some character (actually all even positions should represent SAME character because they MUST match while character comparison). One way to do this is to set some character at all even positions by modifying given string or create a new copy of given string. For example, if input string is “abcb”, new string should be “#a#b#c#b#” if we add # as unique character at even positions.

The two approaches discussed already can be modified a bit to work on modified string where different handling of even and odd positions will not be needed.



We may also add two DIFFERENT characters (not yet used anywhere in string at even and odd positions) at start and end of string as sentinels to avoid bound check. With these changes string “abcb” will look like “ $\wedge$ #a#b#c#b#\$” where  $\wedge$  and \$ are sentinels. This implementation may look cleaner with the cost of more memory.

We are not implementing these here as it's a simple change in given implementations.

Implementation of approach discussed in current article on a modified string can be found at [Longest Palindromic Substring Part II](#) and a [Java Translation](#) of the same by Princeton.

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-4/>

## Chapter 42

# Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 3

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 3 - Geeks-forGeeks

In Manacher's Algorithm [Part 1](#) and [Part 2](#), we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. Here we will implement the same.

We have seen that there are no new character comparison needed in case 1 and case 2. In case 3 and case 4, necessary new comparison are needed.

In following figure,

|              |  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|--------------|--|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| String S     |  | b |   | a |   | b |   | c |   | b |   | a  |    | b  |    | c  |    | b  |    | a  |    | c  |    | c  |    | b  |    | a  |    |    |
| LPS Length L |  | 0 | 1 | 0 | 3 | 0 | 1 | 0 | 7 | 0 | 1 | 0  | 9  | 0  | 1  | 0  | 5  | 0  | 1  | 0  | 1  | 0  | 1  | 2  | 1  | 0  | 1  | 0  | 1  | 0  |
| Position i   |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |

If at all we need a comparison, we will only compare actual characters, which are at “odd” positions like 1, 3, 5, 7, etc.

Even positions do not represent a character in string, so no comparison will be preformed for even positions.

If two characters at different odd positions match, then they will increase LPS length by 2.

There are many ways to implement this depending on how even and odd positions are handled. One way would be to create a new string 1<sup>st</sup> where we insert some unique character (say #, \$ etc) in all even positions and then run algorithm on that (to avoid different way of even and odd position handling). Other way could be to work on given string itself but here even and odd positions should be handled appropriately.

Here we will start with given string itself. When there is a need of expansion and character comparison required, we will expand in left and right positions one by one. When odd position is found, comparison will be done and LPS Length will be incremented by ONE.

When even position is found, no comparison done and LPS Length will be incremented by ONE (So overall, one odd and one even positions on both left and right side will increase LPS Length by TWO).

C/C++

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int expand = -1;
    int diff = -1;
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
        //get currentLeftPosition iMirror for currentRightPosition i
        iMirror = 2*C-i;
        //Reset expand - means no expansion required
        expand = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
        {
            if(L[iMirror] < diff) // Case 1
                L[i] = L[iMirror];
            else if(L[iMirror] == diff && i == N-1) // Case 2
                L[i] = L[iMirror];
            else if(L[iMirror] == diff && i < N-1) // Case 3
            {
```

```

        L[i] = L[iMirror];
        expand = 1; // expansion required
    }
    else if(L[iMirror] > diff) // Case 4
    {
        L[i] = diff;
        expand = 1; // expansion required
    }
}
else
{
    L[i] = 0;
    expand = 1; // expansion required
}

if (expand == 1)
{
    //Attempt to expand palindrome centered at currentRightPosition i
    //Here for odd positions, we compare characters and
    //if match then increment LPS Length by ONE
    //If even position, we just increment LPS by ONE without
    //any character comparison
    while (((i + L[i]) < N && (i - L[i]) > 0) &&
        ( ((i + L[i] + 1) % 2 == 0) ||
        (text[(i + L[i] + 1)/2] == text[(i-L[i]-1)/2] )))
    {
        L[i]++;
    }
}

if(L[i] > maxLPSLength) // Track maxLPSLength
{
    maxLPSLength = L[i];
    maxLPSCenterPosition = i;
}

// If palindrome centered at currentRightPosition i
// expand beyond centerRightPosition R,
// adjust centerPosition C based on expanded palindrome.
if (i + L[i] > R)
{
    C = i;
    R = i + L[i];
}
//Uncomment it to print LPS Length array
//printf("%d ", L[i]);
}
//printf("\n");

```

```
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    //printf("start: %d end: %d\n", start, end);
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}
```

```
int main(int argc, char *argv[])
{

    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
    findLongestPalindromicString();

    strcpy(text, "abcbabcbabcba");
    findLongestPalindromicString();

    strcpy(text, "forgeeksskeegfor");
    findLongestPalindromicString();

    strcpy(text, "caba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcaba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcabba");
    findLongestPalindromicString();

    strcpy(text, "abacdedcaba");
    findLongestPalindromicString();

    return 0;
}
```

## Python

```
# Python program to implement Manacher's Algorithm

def findLongestPalindromicString(text):
```

```

N = len(text)
if N == 0:
    return
N = 2*N+1    # Position count
L = [0] * N
L[0] = 0
L[1] = 1
C = 1        # centerPosition
R = 2        # centerRightPosition
i = 0        # currentRightPosition
iMirror = 0   # currentLeftPosition
maxLPSLength = 0
maxLPSCenterPosition = 0
start = -1
end = -1
diff = -1

# Uncomment it to print LPS Length array
# printf("%d %d ", L[0], L[1]);
for i in xrange(2,N):

    # get currentLeftPosition iMirror for currentRightPosition i
    iMirror = 2*C-i
    L[i] = 0
    diff = R - i
    # If currentRightPosition i is within centerRightPosition R
    if diff > 0:
        L[i] = min(L[iMirror], diff)

    # Attempt to expand palindrome centered at currentRightPosition i
    # Here for odd positions, we compare characters and
    # if match then increment LPS Length by ONE
    # If even position, we just increment LPS by ONE without
    # any character comparison
    try:
        while ((i+L[i]) < N and (i-L[i]) > 0) and \
            (((i+L[i]+1) % 2 == 0) or \
            (text[(i+L[i]+1)/2] == text[(i-L[i]-1)/2]]):
            L[i]+=1
    except Exception as e:
        pass

    if L[i] > maxLPSLength:        # Track maxLPSLength
        maxLPSLength = L[i]
        maxLPSCenterPosition = i

    # If palindrome centered at currentRightPosition i
    # expand beyond centerRightPosition R,

```

```
# adjust centerPosition C based on expanded palindrome.
if i + L[i] > R:
    C = i
    R = i + L[i]

# Uncomment it to print LPS Length array
# printf("%d ", L[i]);
start = (maxLPSCenterPosition - maxLPSLength) / 2
end = start + maxLPSLength - 1
print "LPS of string is " + text + " : ",
print text[start:end+1],
print "\n",

# Driver program
text1 = "babcbabcbaccba"
findLongestPalindromicString(text1)

text2 = "abaaba"
findLongestPalindromicString(text2)

text3 = "abababa"
findLongestPalindromicString(text3)

text4 = "abcbabcbabcba"
findLongestPalindromicString(text4)

text5 = "forgeeksskeegfor"
findLongestPalindromicString(text5)

text6 = "caba"
findLongestPalindromicString(text6)

text7 = "abacdfgdcaba"
findLongestPalindromicString(text7)

text8 = "abacdfgdcabba"
findLongestPalindromicString(text8)

text9 = "abacdedcaba"
findLongestPalindromicString(text9)

# This code is contributed by BHAVYA JAIN
```

Output:

```
LPS of string is babcbabcbaccba : abcbabcba
LPS of string is abaaba : abaaba
```

```
LPS of string is abababa : abababa
LPS of string is abcbabcbabcba : abcbabcbabcba
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdfgdcaba : aba
LPS of string is abacdfgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba
```

This is the implementation based on the four cases discussed in [Part 2](#). In [Part 4](#), we have discussed a different way to look at these four cases and few other approaches.

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-3-2/>



## Chapter 43

# Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 2

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 2 - Geeks-forGeeks

In [Manacher's Algorithm – Part 1](#), we gone through some of the basics and LPS length array.

Here we will see how to calculate LPS length array efficiently.

To calculate LPS array efficiently, we need to understand how LPS length for any position may relate to LPS length value of any previous already calculated position.

For string “abaaba”, we see following:

If we look around position 3:

- LPS length value at position 2 and position 4 are same
- LPS length value at position 1 and position 5 are same

We calculate LPS length values from left to right starting from position 0, so we can see if we already know LPS length values at positions 1, 2 and 3 already then we may not need to calculate LPS length at positions 4 and 5 because they are equal to LPS length values at corresponding positions on left side of position 3.

If we look around position 6:

- LPS length value at position 5 and position 7 are same
- LPS length value at position 4 and position 8 are same

..... and so on.

If we already know LPS length values at positions 1, 2, 3, 4, 5 and 6 already then we may not need to calculate LPS length at positions 7, 8, 9, 10 and 11 because they are equal to LPS length values at corresponding positions on left side of position 6.

For string “abababa”, we see following:

If we already know LPS length values at positions 1, 2, 3, 4, 5, 6 and 7 already then we may not need to calculate LPS length at positions 8, 9, 10, 11, 12 and 13 because they are equal to LPS length values at corresponding positions on left side of position 7.

Can you see why LPS length values are symmetric around positions 3, 6, 9 in string “abaaba”? That's because there is a palindromic substring around these positions. Same is the case in string “abababa” around position 7.

Is it always true that LPS length values around at palindromic center position are always symmetric (same)?

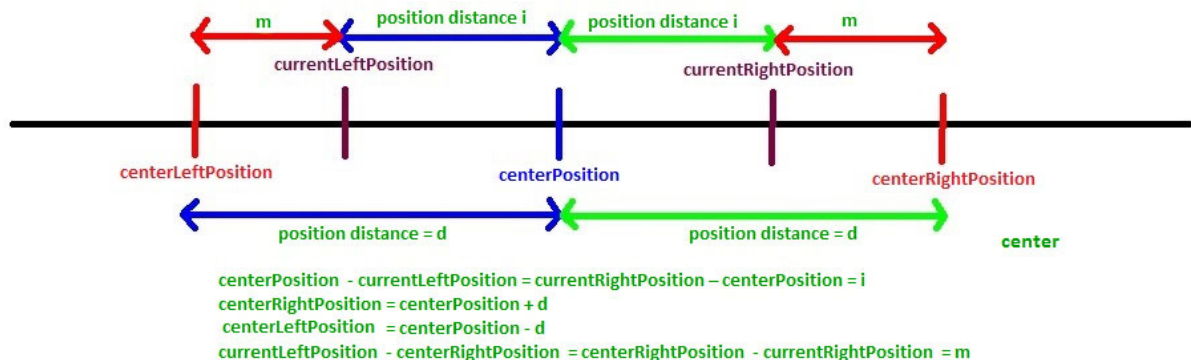
Answer is NO.

Look at positions 3 and 11 in string “abababa”. Both positions have LPS length 3. Immediate left and right positions are symmetric (with value 0), but not the next one. Positions 1 and 5 (around position 3) are not symmetric. Similarly, positions 9 and 13 (around position 11) are not symmetric.

At this point, we can see that if there is a palindrome in a string centered at some position, then LPS length values around the center position may or may not be symmetric depending on some situation. If we can identify the situation when left and right positions WILL BE SYMMETRIC around the center position, we NEED NOT calculate LPS length of the right position because it will be exactly same as LPS value of corresponding position on the left side which is already known. And this fact where we are avoiding LPS length computation at few positions makes Manacher's Algorithm linear.

In situations when left and right positions WILL NOT BE SYMMETRIC around the center position, we compare characters in left and right side to find palindrome, but here also algorithm tries to avoid certain no of comparisons. We will see all these scenarios soon.

Let's introduce few terms to proceed further:



- **centerPosition** – This is the position for which LPS length is calculated and let's say LPS length at centerPosition is d (i.e.  $L[\text{centerPosition}] = d$ )

- **centerRightPosition** – This is the position which is right to the centerPosition and d position away from centerPosition (i.e. **centerRightPosition** = **centerPosition** + d)
- **centerLeftPosition** – This is the position which is left to the centerPosition and d position away from centerPosition (i.e. **centerLeftPosition** = **centerPosition** – d)
- **currentRightPosition** – This is the position which is right of the centerPosition for which LPS length is not yet known and has to be calculated
- **currentLeftPosition** – This is the position on the left side of centerPosition which corresponds to the currentRightPosition  
 $\text{centerPosition} - \text{currentLeftPosition} = \text{currentRightPosition} - \text{centerPosition}$   
 $\text{currentLeftPosition} = 2 * \text{centerPosition} - \text{currentRightPosition}$
- **i-left palindrome** – The palindrome i positions left of centerPosition, i.e. at currentLeftPosition
- **i-right palindrome** – The palindrome i positions right of centerPosition, i.e. at currentRightPosition
- **center palindrome** – The palindrome at centerPosition

When we are at centerPosition for which LPS length is known, then we also know LPS length of all positions smaller than centerPosition. Let's say LPS length at centerPosition is d, i.e.

$$L[\text{centerPosition}] = d$$

It means that substring between positions “centerPosition-d” to “centerPosition+d” is a palindrom.

Now we proceed further to calculate LPS length of positions greater than centerPosition. Let's say we are at currentRightPosition ( > centerPosition) where we need to find LPS length.

For this we look at LPS length of currentLeftPosition which is already calculated.

If LPS length of currentLeftPosition is less than “centerRightPosition – currentRightPosition”, then LPS length of currentRightPosition will be equal to LPS length of currentLeftPosition. So

$L[\text{currentRightPosition}] = L[\text{currentLeftPosition}]$  if  $L[\text{currentLeftPosition}] < \text{centerRightPosition} - \text{currentRightPosition}$ . This is **Case 1**.

Let's consider below scenario for string “abababa”:

(click to see it clearly)

We have calculated LPS length up-to position 7 where  $L[7] = 7$ , if we consider position 7 as centerPosition, then centerLeftPosition will be 0 and centerRightPosition will be 14.

Now we need to calculate LPS length of other positions on the right of centerPosition.

For currentRightPosition = 8, currentLeftPosition is 6 and  $L[\text{currentLeftPosition}] = 0$

Also  $\text{centerRightPosition} - \text{currentRightPosition} = 14 - 8 = 6$

Case 1 applies here and so  $L[\text{currentRightPosition}] = L[8] = 0$

Case 1 applies to positions 10 and 12, so,

$L[10] = L[4] = 0$   
 $L[12] = L[2] = 0$

If we look at position 9, then:

$\text{currentRightPosition} = 9$

$\text{currentLeftPosition} = 2 * \text{centerPosition} - \text{currentRightPosition} = 2 * 7 - 9 = 5$

$\text{centerRightPosition} - \text{currentRightPosition} = 14 - 9 = 5$

Here  $L[\text{currentLeftPosition}] = \text{centerRightPosition} - \text{currentRightPosition}$ , so Case 1 doesn't apply here. Also note that  $\text{centerRightPosition}$  is the extreme end position of the string. That means center palindrome is suffix of input string. In that case,  $L[\text{currentRightPosition}] = L[\text{currentLeftPosition}]$ . This is **Case 2**.

Case 2 applies to positions 9, 11, 13 and 14, so:

$L[9] = L[5] = 5$

$L[11] = L[3] = 3$

$L[13] = L[1] = 1$

$L[14] = L[0] = 0$

What is really happening in Case 1 and Case 2? This is just utilizing the palindromic symmetric property and without any character match, it is finding LPS length of new positions.

When a bigger length palindrome contains a smaller length palindrome centered at left side of its own center, then based on symmetric property, there will be another same smaller palindrome centered on the right of bigger palindrome center. If left side smaller palindrome is not prefix of bigger palindrome, then **Case 1** applies and if it is a prefix AND bigger palindrome is suffix of the input string itself, then **Case 2** applies.

*The longest palindrome  $i$  places to the right of the current center (the  $i$ -right palindrome) is as long as the longest palindrome  $i$  places to the left of the current center (the  $i$ -left palindrome) if the  $i$ -left palindrome is completely contained in the longest palindrome around the current center (the center palindrome) and the  $i$ -left palindrome is not a prefix of the center palindrome (**Case 1**) or (i.e. when  $i$ -left palindrome is a prefix of center palindrome) if the center palindrome is a suffix of the entire string (**Case 2**).*

In Case 1 and Case 2,  $i$ -right palindrome can't expand more than corresponding  $i$ -left palindrome (can you visualize why it can't expand more?), and so LPS length of  $i$ -right palindrome is exactly same as LPS length of  $i$ -left palindrome.

Here both  $i$ -left and  $i$ -right palindromes are completely contained in center palindrome (i.e.  $L[\text{currentLeftPosition}] \leq \text{centerRightPosition} - \text{currentRightPosition}$ )

Now if  $i$ -left palindrome is not a prefix of center palindrome ( $L[\text{currentLeftPosition}] < \text{centerRightPosition} - \text{currentRightPosition}$ ), that means that  $i$ -left palindrome was not able to expand up-to position  $\text{centerLeftPosition}$ .

If we look at following with  $\text{centerPosition} = 11$ , then

(click to see it clearly)

$\text{centerLeftPosition}$  would be  $11 - 9 = 2$ , and  $\text{centerRightPosition}$  would be  $11 + 9 = 20$

If we take  $\text{currentRightPosition} = 15$ , its  $\text{currentLeftPosition}$  is 7. Case 1 applies here and

so  $L[15] = 3$ . i-left palindrome at position 7 is “bab” which is completely contained in center palindrome at position 11 (which is “dbabcbabd”). We can see that i-right palindrome (at position 15) can't expand more than i-left palindrome (at position 7).

If there was a possibility of expansion, i-left palindrome could have expanded itself more already. But there is no such possibility as i-left palindrome is prefix of center palindrome. So due to symmetry property, i-right palindrome will be exactly same as i-left palindrome and it can't expand more. This makes  $L[\text{currentRightPosition}] = L[\text{currentLeftPosition}]$  in Case 1.

Now if we consider  $\text{centerPosition} = 19$ , then  $\text{centerLeftPosition} = 12$  and  $\text{centerRightPosition} = 26$

If we take  $\text{currentRightPosition} = 23$ , it's  $\text{currentLeftPosition}$  is 15. Case 2 applies here and so  $L[23] = 3$ . i-left palindrome at position 15 is “bab” which is completely contained in center palindrome at position 19 (which is “babdbab”). In Case 2, where i-left palindrome is prefix of center palindrome, i-right palindrome can't expand more than length of i-left palindrome because center palindrome is suffix of input string so there are no more character left to compare and expand. This makes  $L[\text{currentRightPosition}] = L[\text{currentLeftPosition}]$  in Case 2.

**Case 1:**  $L[\text{currentRightPosition}] = L[\text{currentLeftPosition}]$  applies when:

- i-left palindrome is completely contained in center palindrome
- i-left palindrome is NOT a prefix of center palindrome

Both above conditions are satisfied when

$$L[\text{currentLeftPosition}] < \text{centerRightPosition} - \text{currentRightPosition}$$

**Case 2:**  $L[\text{currentRightPosition}] = L[\text{currentLeftPosition}]$  applies when:

- i-left palindrome is prefix of center palindrome (means completely contained also)
- center palindrome is suffix of input string

Above conditions are satisfied when

$$L[\text{currentLeftPosition}] = \text{centerRightPosition} - \text{currentRightPosition} \quad (\text{For } 1^{\text{st}} \text{ condition})$$

AND

$$\text{centerRightPosition} = 2 * N \text{ where } N \text{ is input string length } N \quad (\text{For } 2^{\text{nd}} \text{ condition}).$$

**Case 3:**  $L[\text{currentRightPosition}] > L[\text{currentLeftPosition}]$  applies when:

- i-left palindrome is prefix of center palindrome (and so i-left palindrome is completely contained in center palindrome)
- center palindrome is NOT suffix of input string

Above conditions are satisfied when

$$L[\text{currentLeftPosition}] = \text{centerRightPosition} - \text{currentRightPosition} \quad (\text{For } 1^{\text{st}} \text{ condition})$$

AND

$$\text{centerRightPosition} < 2 * N \text{ where } N \text{ is input string length } N \quad (\text{For } 2^{\text{nd}} \text{ condition}).$$

In this case, there is a possibility of i-right palindrome expansion and so length of i-right palindrome is at least as long as length of i-left palindrome.

**Case 4:**  $L[\text{currentRightPosition}] \geq \text{centerRightPosition} - \text{currentRightPosition}$  applies when:

- i-left palindrome is NOT completely contained in center palindrome

Above condition is satisfied when

$L[\text{currentLeftPosition}] > \text{centerRightPosition} - \text{currentRightPosition}$

In this case, length of i-right palindrome is at least as long (centerRightPosition – currentRightPosition) and there is a possibility of i-right palindrome expansion.

In following figure,

(click to see it clearly)

If we take center position 7, then Case 3 applies at currentRightPosition 11 because i-left palindrome at currentLeftPosition 3 is a prefix of center palindrome and i-right palindrome is not suffix of input string, so here  $L[11] = 9$ , which is greater than i-left palindrome length  $L[3] = 3$ . In the case, it is guaranteed that  $L[11]$  will be at least 3, and so in implementation, we 1<sup>st</sup> set  $L[11] = 3$  and then we try to expand it by comparing characters in left and right side starting from distance 4 (As up-to distance 3, it is already known that characters will match).

If we take center position 11, then Case 4 applies at currentRightPosition 15 because  $L[\text{currentLeftPosition}] = L[7] = 7 > \text{centerRightPosition} - \text{currentRightPosition} = 20 - 15 = 5$ . In the case, it is guaranteed that  $L[15]$  will be at least 5, and so in implementation, we 1<sup>st</sup> set  $L[15] = 5$  and then we try to expand it by comparing characters in left and right side starting from distance 5 (As up-to distance 5, it is already known that characters will match).

Now one point left to discuss is, when we work at one center position and compute LPS lengths for different rightPositions, how to know that what would be next center position. We change centerPosition to currentRightPosition if palindrome centered at currentRightPosition expands beyond centerRightPosition.

Here we have seen four different cases on how LPS length of a position will depend on a previous position's LPS length.

In [Part 3](#), we have discussed code implementation of it and also we have looked at these four cases in a different way and implement that too.

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-2/>

## Chapter 44

# Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 1

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 1 - Geeks-forGeeks

Given a string, find the longest substring which is palindrome.

- if the given string is “forgeeksskeegfor”, the output should be “geeksskeeg”
- if the given string is “abaaba”, the output should be “abaaba”
- if the given string is “abababa”, the output should be “abababa”
- if the given string is “abcbabcbabcba”, the output should be “abcbabcb”

We have already discussed Naïve [ $O(n^3)$ ] and quadratic [ $O(n^2)$ ] approaches at [Set 1](#) and [Set 2](#).

In this article, we will talk about [Manacher's algorithm](#) which finds Longest Palindromic Substring in linear time.

One way ([Set 2](#)) to find a palindrome is to start from the center of the string and compare characters in both directions one by one. If corresponding characters on both sides (left and right of the center) match, then they will make a palindrome.

Let's consider string “abababa”.

Here center of the string is 4th character (with index 3) b. If we match characters in left and right of the center, all characters match and so string “abababa” is a palindrome.

Here center position is not only the actual string character position but it could be the position between two characters also.

Consider string “abaaba” of even length. This string is palindrome around the position between 3rd and 4th characters a and a respectively.

To find Longest Palindromic Substring of a string of length  $N$ , one way is take each possible  $2*N + 1$  centers (the  $N$  character positions,  $N-1$  between two character positions and 2 positions at left and right ends), do the character match in both left and right directions at each  $2*N+ 1$  centers and keep track of LPS. This approach takes  $O(N^2)$  time and that's what we are doing in [Set 2](#).

Let's consider two strings "abababa" and "abaaba" as shown below:

In these two strings, left and right side of the center positions (position 7 in 1st string and position 6 in 2nd string) are symmetric. Why? Because the whole string is palindrome around the center position.

If we need to calculate Longest Palindromic Substring at each  $2*N+1$  positions from left to right, then palindrome's symmetric property could help to avoid some of the unnecessary computations (i.e. character comparison). If there is a palindrome of some length  $L$  centered at any position  $P$ , then we may not need to compare all characters in left and right side at position  $P+1$ . We already calculated LPS at positions before  $P$  and they can help to avoid some of the comparisons after position  $P$ .

This use of information from previous positions at a later point of time makes the Manacher's algorithm linear. In [Set 2](#), there is no reuse of previous information and so that is quadratic.

Manacher's algorithm is probably considered complex to understand, so here we will discuss it in as detailed way as we can. Some of its portions may require multiple reading to understand it properly.

Let's look at string "abababa". In 3rd figure above, 15 center positions are shown. We need to calculate length of longest palindromic string at each of these positions.

- At position 0, there is no LPS at all (no character on left side to compare), so length of LPS will be 0.
- At position 1, LPS is a, so length of LPS will be 1.
- At position 2, there is no LPS at all (left and right characters a and b don't match), so length of LPS will be 0.
- At position 3, LPS is aba, so length of LPS will be 3.
- At position 4, there is no LPS at all (left and right characters b and a don't match), so length of LPS will be 0.
- At position 5, LPS is ababa, so length of LPS will be 5.

..... and so on

We store all these palindromic lengths in an array, say  $L$ . Then string  $S$  and LPS Length  $L$  look like below:



Similarly, LPS Length L of string “abaaba” will look like:

In LPS Array L:

- LPS length value at odd positions (the actual character positions) will be odd and greater than or equal to 1 (1 will come from the center character itself if nothing else matches in left and right side of it)
- LPS length value at even positions (the positions between two characters, extreme left and right positions) will be even and greater than or equal to 0 (0 will come when there is no match in left and right side)

**Position and index for the string are two different things here. For a given string S of length N, indexes will be from 0 to N-1 (total N indexes) and positions will be from 0 to 2\*N (total 2\*N+1 positions).**

LPS length value can be interpreted in two ways, one in terms of index and second in terms of position. LPS value d at position I ( $L[i] = d$ ) tells that:

- Substring from position  $i-d$  to  $i+d$  is a palindrome of length d (in terms of position)
- Substring from index  $(i-d)/2$  to  $[(i+d)/2 - 1]$  is a palindrome of length d (in terms of index)

e.g. in string “abaaba”,  $L[3] = 3$  means substring from position 0 ( $3-3$ ) to 6 ( $3+3$ ) is a palindrome which is “aba” of length 3, it also means that substring from index 0  $[(3-3)/2]$  to 2  $[(3+3)/2 - 1]$  is a palindrome which is “aba” of length 3.

Now the main task is to compute LPS array efficiently. Once this array is computed, LPS of string S will be centered at position with maximum LPS length value.

We will see it in [Part 2](#).

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/>

## Chapter 45

# Suffix Tree Application 5 – Longest Common Substring

Suffix Tree Application 5 - Longest Common Substring - GeeksforGeeks

Given two strings X and Y, find the [Longest Common Substring](#) of X and Y.

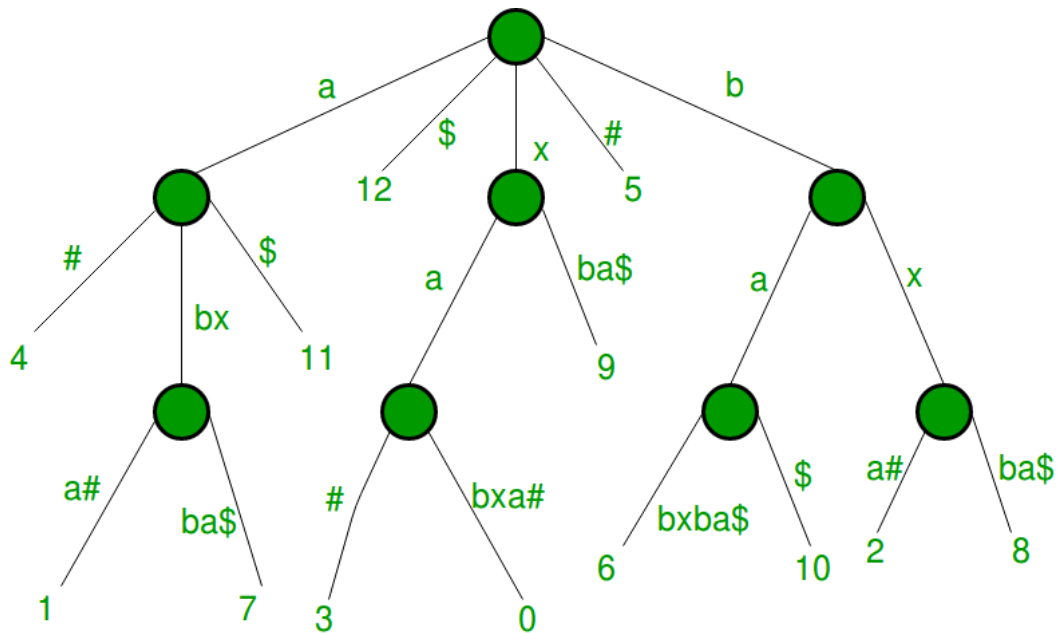
Naive [ $O(N \cdot M^2)$ ] and Dynamic Programming [ $O(N \cdot M)$ ] approaches are already discussed [here](#).

In this article, we will discuss a linear time approach to find LCS using suffix tree (The 5<sup>th</sup> Suffix Tree Application).

Here we will build generalized suffix tree for two strings X and Y as discussed already at: [Generalized Suffix Tree 1](#)

Lets take same example ( $X = \text{xabxa}$ , and  $Y = \text{babxba}$ ) we saw in [Generalized Suffix Tree 1](#).

We built following suffix tree for X and Y there:



This is generalized suffix tree for  $xabxa\#babxba\$$

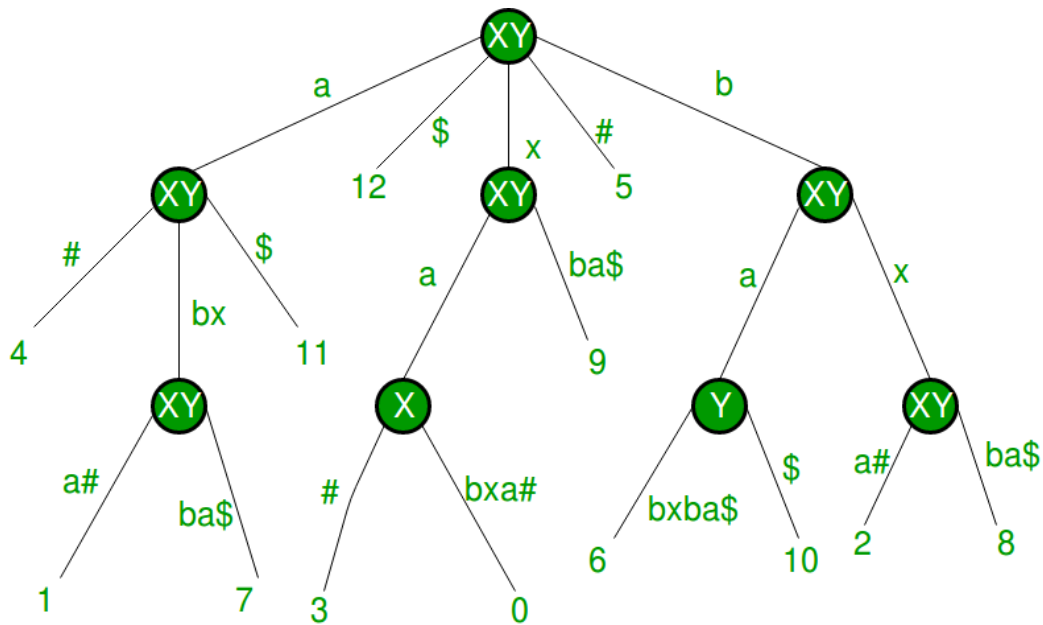
In above, leaves with suffix indices in  $[0,4]$  are suffixes of string  $xabxa$  and leaves with suffix indices in  $[6,11]$  are suffixes of string  $babxba$ . Why ??

Because in concatenated string  $xabxa\#babxba\$$ , index of string  $xabxa$  is 0 and it's length is 5, so indices of it's suffixes would be 0, 1, 2, 3 and 4. Similarly index of string  $babxba$  is 6 and it's length is 6, so indices of it's suffixes would be 6, 7, 8, 9, 10 and 11.

With this, we can see that in the generalized suffix tree figure above, there are some internal nodes having leaves below it from

- both strings X and Y (i.e. there is at least one leaf with suffix index in  $[0,4]$  and one leaf with suffix index in  $[6, 11]$ )
- string X only (i.e. all leaf nodes have suffix indices in  $[0,4]$ )
- string Y only (i.e. all leaf nodes have suffix indices in  $[6,11]$ )

Following figure shows the internal nodes marked as "XY", "X" or "Y" depending on which string the leaves belong to, that they have below themselves.



What these “XY”, “X” or “Y” marking mean ?

Path label from root to an internal node gives a substring of X or Y or both.

For node marked as XY, substring from root to that node belongs to both strings X and Y.

For node marked as X, substring from root to that node belongs to string X only.

For node marked as Y, substring from root to that node belongs to string Y only.

By looking at above figure, can you see how to get LCS of X and Y ?

By now, it should be clear that how to get common substring of X and Y at least.

If we traverse the path from root to nodes marked as XY, we will get common substring of X and Y.

Now we need to find the longest one among all those common substrings.

Can you think how to get LCS now ? Recall how did we get [Longest Repeated Substring](#) in a given string using suffix tree already.

The path label from root to the deepest node marked as XY will give the LCS of X and Y. The deepest node is highlighted in above figure and path label “abx” from root to that node is the LCS of X and Y.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for two strings
// And then we find longest common substring of the two input strings
#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. Lets say there are two nodes 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 its 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 its
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as 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
int size1 = 0; //Size of 1st 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 existng 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 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 && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//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)
{
    for(i= n->start; i<= *(n->end); i++)
    {
        if(text[i] == '#')
        {
            n->end = (int*) malloc(sizeof(int));
            *(n->end) = i;
        }
    }
    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 doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                ret = doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);

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

```
        n->suffixIndex = ret;
    else if((n->suffixIndex == -2 && ret == -3) ||
           (n->suffixIndex == -3 && ret == -2) ||
           n->suffixIndex == -4)
    {
        n->suffixIndex = -4; //Mark node as XY
        //Keep track of deepest node
        if(*maxHeight < labelHeight)
        {
            *maxHeight = labelHeight;
            *substringStartIndex = *(n->end) -
                labelHeight + 1;
        }
    }
}
}
}
else if(n->suffixIndex > -1 && n->suffixIndex < size1) //suffix of X
    return -2; //Mark node as X
else if(n->suffixIndex >= size1) //suffix of Y
    return -3; //Mark node as Y
return n->suffixIndex;
}

void getLongestCommonSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No common substring");
    else
        printf(", of length: %d", maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 7;
    printf("Longest Common Substring in xabxac and abcabxabcd is: ");
    strcpy(text, "xabxac#abcbxabcd$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
```

```
freeSuffixTreeByPostOrder(root);

size1 = 10;
printf("Longest Common Substring in xabxaabxa and babxba is: ");
strcpy(text, "xabxaabxa#babxba$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 14;
printf("Longest Common Substring in GeeksforGeeks and GeeksQuiz is: ");
strcpy(text, "GeeksforGeeks#GeeksQuiz$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 26;
printf("Longest Common Substring in OldSite:GeeksforGeeks.org");
printf(" and NewSite:GeeksQuiz.com is: ");
strcpy(text, "OldSite:GeeksforGeeks.org#NewSite:GeeksQuiz.com$");
buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Common Substring in abcde and fghie is: ");
strcpy(text, "abcde#fghie$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Common Substring in pqrst and uvwxyz is: ");
strcpy(text, "pqrst#uvwxyz$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}
```

Output:

```
Longest Common Substring in xabxac and abcabxabcd is: abxa, of length: 4
Longest Common Substring in xabxaabxa and babxba is: abx, of length: 3
Longest Common Substring in GeeksforGeeks and GeeksQuiz is: Geeks, of length: 5
```

Longest Common Substring in OldSite:GeeksforGeeks.org and  
NewSite:GeeksQuiz.com is: Site:Geeks, of length: 10  
Longest Common Substring in abcde and fghie is: e, of length: 1  
Longest Common Substring in pqrst and uvwxyz is: No common substring

If two strings are of size M and N, then Generalized Suffix Tree construction takes  $O(M+N)$  and LCS finding is a DFS on tree which is again  $O(M+N)$ .  
So overall complexity is linear in time and space.

**Followup:**

1. Given a pattern, check if it is substring of X or Y or both. If it is a substring, find all its occurrences along with which string (X or Y or both) it belongs to.
2. Extend the implementation to find LCS of more than two strings
3. Solve problem 1 for more than two strings
4. Given a string, find its [Longest Palindromic Substring](#)

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)
- [Generalized Suffix Tree 1](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<https://www.geeksforgeeks.org/suffix-tree-application-5-longest-common-substring-2/>

## Chapter 46

# Generalized Suffix Tree 1

Generalized Suffix Tree 1 - GeeksforGeeks

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for [substring check](#), [searching all patterns](#), [longest repeated substring](#) and [built suffix array](#) (All linear time operations).

There are lots of other problems where multiple strings are involved.  
e.g. pattern searching in a text file or dictionary, spell checker, phone book, [Autocomplete](#), [Longest common substring problem](#), [Longest palindromic substring](#) and [More](#).

For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.

A suffix tree made of a set of strings is known as [Generalized Suffix Tree](#).

We will discuss a simple way to build [Generalized Suffix Tree](#) here for **two strings only**.

Later, we will discuss another approach to build [Generalized Suffix Tree](#) for **two or more strings**.

Here we will use the [suffix tree implementation](#) for one string discussed already and modify that a bit to build [generalized suffix tree](#).

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y\$ where # and \$ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y\$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Lets say X = xabxa, and Y = babxba, then

X#Y\$ = xabxa#babxba\$

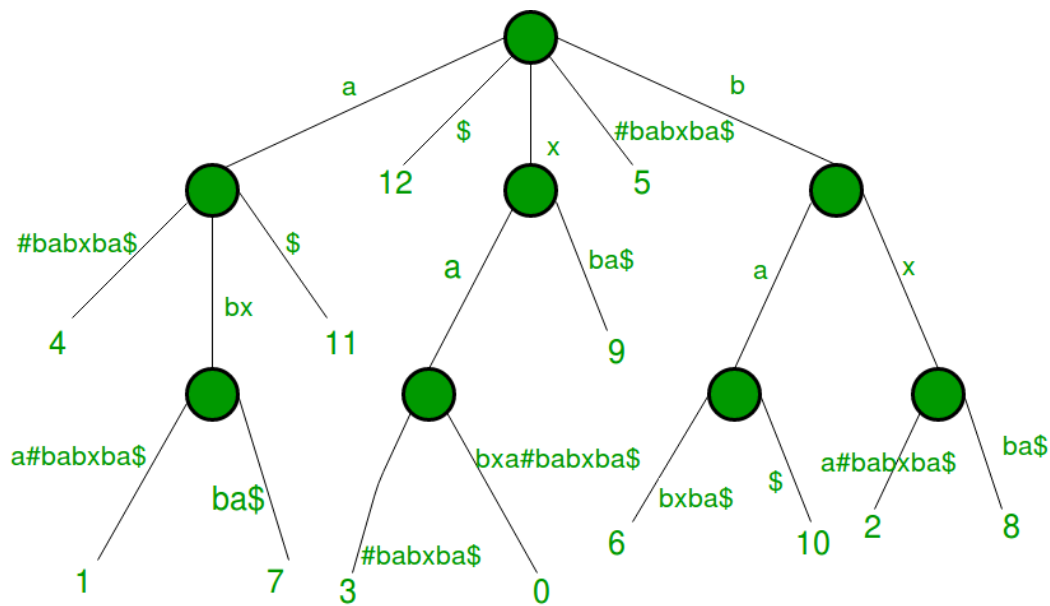
If we run the code implemented at [Ukkonen's Suffix Tree Construction – Part 6](#) for string xabxa#babxba\$, we get following output:

**Output:**

# babxba\$ [5]  
\$ [12]  
a [-1]  
#babxba\$ [4]  
\$ [11]  
bx [-1]  
a#babxba\$ [1]  
ba\$ [7]  
b [-1]  
a [-1]  
\$ [10]  
bxba\$ [6]  
x [-1]  
a#babxba\$ [2]  
ba\$ [8]  
x [-1]  
a [-1]  
#babxba\$ [3]  
bxa#babxba\$ [0]  
ba\$ [9]

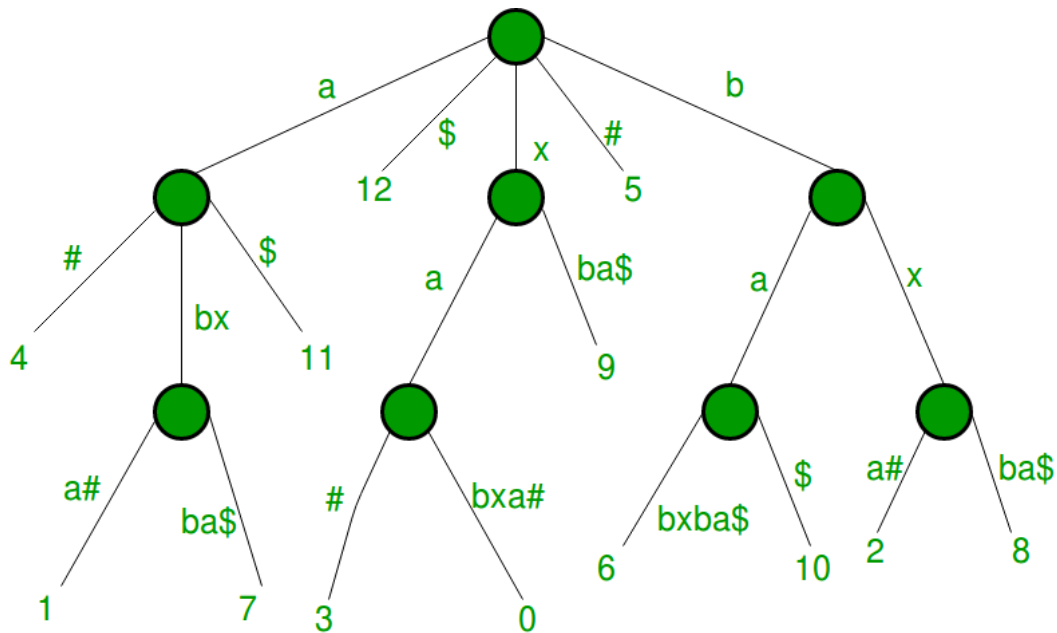
Pictorial View:





Suffix tree for string **xabxa#babxa\$**

We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. For example, for path labels `#babxba$`, `a#babxba$` and `bxa#babxba$`, we can remove `babxba$` (belongs to 2<sup>nd</sup> input string) and then new path labels will be `#`, `a#` and `bxa#` respectively. With this change, above diagram will look like below:



Below implementation is built on top of [original implementation](#). Here we are removing unwanted characters on path labels. If a path label has “#” character in it, then we are trimming all characters after the “#” in that path label.

**Note:** This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y\$

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then build generalized suffix tree
#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. Lets say there are two nodes 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 represeted as 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 existng 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 curent 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 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 && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//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
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            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)
{
    for(i= n->start; i<= *(n->end); i++)
    {
        if(text[i] == '#') //Trim unwanted characters
        {
            n->end = (int*) malloc(sizeof(int));
            *(n->end) = i;
        }
    }
    n->suffixIndex = size - labelHeight;
    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);

//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
// strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
strcpy(text, "xabxa#babxba$"); buildSuffixTree();
return 0;
}

```

Output: (You can see that below output corresponds to the 2<sup>nd</sup> Figure shown above)

```

# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]

```

If two strings are of size M and N, this implementation will take  $O(M+N)$  time and space.

If input strings are not concatenated already, then it will take  $2(M+N)$  space in total,  $M+N$  space to store the generalized suffix tree and another  $M+N$  space to store concatenated string.

**Followup:**

Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.

We will discuss another approach to build generalized suffix tree soon where we will need only one unique terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<https://www.geeksforgeeks.org/generalized-suffix-tree-1/>

## Chapter 47

# Suffix Tree Application 4 – Build Linear Time Suffix Array

Suffix Tree Application 4 - Build Linear Time Suffix Array - GeeksforGeeks

Given a string, build it's [Suffix Array](#)

We have already discussed following two ways of building suffix array:

- [Naive  \$O\(n^2 \text{Log} n\)\$  algorithm](#)
- [Enhanced  \$O\(n \text{Log} n\)\$  algorithm](#)

Please go through these to have the basic understanding.

Here we will see how to build suffix array in linear time using suffix tree.

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

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

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

Lets consider string `abcabxabcd`.

It's suffix array would be:

0 6 3 1 7 4 2 8 9 5

Lets look at following figure:



```
#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. Lets say there are two nodes 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 its 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 its
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as 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 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);
}

void doTraversal(Node *n, int suffixArray[], int *idx)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], suffixArray, idx);
            }
        }
    }
    //If it is Leaf node other than "$" label
    else if(n->suffixIndex > -1 && n->suffixIndex < size)
    {
        suffixArray[(*idx)++] = n->suffixIndex;
    }
}

void buildSuffixArray(int suffixArray[])
{

```

```
int i = 0;
for(i=0; i< size; i++)
    suffixArray[i] = -1;
int idx = 0;
doTraversal(root, suffixArray, &idx);
printf("Suffix Array for String ");
for(i=0; i<size; i++)
    printf("%c", text[i]);
printf(" is: ");
for(i=0; i<size; i++)
    printf("%d ", suffixArray[i]);
printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "banana$");
    buildSuffixTree();
    size--;
    int *suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
```

```
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    size--;
    suffixArray = (int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "abcabxabcd$");
    buildSuffixTree();
    size--;
    suffixArray = (int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "CCAAACCCGATTA$");
    buildSuffixTree();
    size--;
    suffixArray = (int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    return 0;
}
```

Output:

```
Suffix Array for String banana is: 5 3 1 0 4 2
Suffix Array for String GEEKSFORGEEKS is: 9 1 10 2 5 8 0 11 3 6 7 12 4
Suffix Array for String AAAAAAAAAA is: 9 8 7 6 5 4 3 2 1 0
Suffix Array for String ABCDEFG is: 0 1 2 3 4 5 6
Suffix Array for String ABABABA is: 6 4 2 0 5 3 1
Suffix Array for String abcabxabcd is: 0 6 3 1 7 4 2 8 9 5
Suffix Array for String CCAAACCCGATTA is: 12 2 3 4 9 1 0 5 6 7 8 11 10
```

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 of tree take  $O(N)$  to build suffix array.

So overall, it's linear in time and space.

Can you see why traversal is  $O(N)$  ?? Because a suffix tree of string of length  $N$  will have at most  $N-1$  internal nodes and  $N$  leaves. Traversal of these nodes can be done in  $O(N)$ .

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/suffix-tree-application-4-build-linear-time-suffix-array/>

## Chapter 48

# Suffix Tree Application 3 – Longest Repeated Substring

Suffix Tree Application 3 - Longest Repeated Substring - GeeksforGeeks

Given a text string, find [Longest Repeated Substring](#) in the text. If there are more than one Longest Repeated Substrings, get any one of them.

```
Longest Repeated Substring in GEEKSFORGEEEKS is: GEEKS
Longest Repeated Substring in AAAAAAAAAA is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabppq is: ab (pq is another LRS here)
```

This problem can be solved by different approaches with varying time and space complexities. Here we will discuss Suffix Tree approach (3<sup>rd</sup> Suffix Tree Application). Other approaches will be discussed soon.

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

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

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

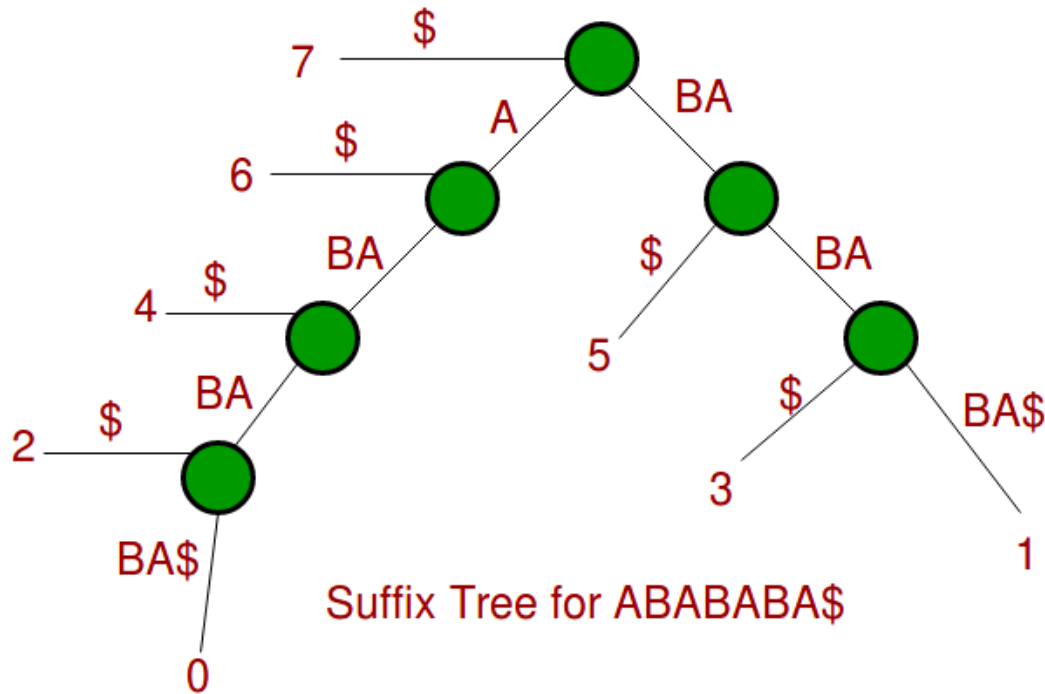
[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

Lets look at following figure:



This is suffix tree for string “ABABABA\$”.

In this string, following substrings are repeated:

A, B, AB, BA, ABA, BAB, ABAB, BABA, ABABA

And Longest Repeated Substring is ABABA.

In a suffix tree, one node can’t have more than one outgoing edge starting with same character, and so if there are repeated substring in the text, they will share on same path and that path in suffix tree will go through one or more internal node(s) down the tree (below the point where substring ends on that path).

In above figure, we can see that

- Path with Substring “A” has three internal nodes down the tree
- Path with Substring “AB” has two internal nodes down the tree
- Path with Substring “ABA” has two internal nodes down the tree
- Path with Substring “ABAB” has one internal node down the tree
- Path with Substring “ABABA” has one internal node down the tree
- Path with Substring “B” has two internal nodes down the tree
- Path with Substring “BA” has two internal nodes down the tree
- Path with Substring “BAB” has one internal node down the tree
- Path with Substring “BABA” has one internal node down the tree

All above substrings are repeated.

Substrings ABABAB, ABABABA, BABAB, BABABA have no internal node down the tree (after the point where substring end on the path), and so these are not repeated.

Can you see how to find longest repeated substring ??

We can see in figure that, longest repeated substring will end at the internal node which



is farthest from the root (i.e. deepest node in the tree), because length of substring is the path label length from root to that internal node.

So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then find Longest Repeated Substring
#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. Lets say there are two nodes 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 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 existng 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 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);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]), maxHeight,
```

```
        substringStartIndex);
    }
}
else if(n->suffixIndex > -1 &&
        (*maxHeight < labelHeight - edgeLength(n)))
{
    *maxHeight = labelHeight - edgeLength(n);
    *substringStartIndex = n->suffixIndex;
}
}

void getLongestRepeatedSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);
    // printf("maxHeight %d, substringStartIndex %d\n", maxHeight,
    //        substringStartIndex);
    printf("Longest Repeated Substring in %s is: ", text);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No repeated substring");
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}
```



```
    strcpy(text, "ABABABA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ATCGATCGA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "banana$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "abcpqrabppq$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "pqrppqabab$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Longest Repeated Substring in GEEKSFORGEEKS$ is: GEEKS
Longest Repeated Substring in AAAAAAAAAA$ is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG$ is: No repeated substring
Longest Repeated Substring in ABABABA$ is: ABABA
Longest Repeated Substring in ATCGATCGA$ is: ATCGA
Longest Repeated Substring in banana$ is: ana
Longest Repeated Substring in abcpqrabppq$ is: ab
Longest Repeated Substring in pqrppqabab$ is: ab
```

In case of multiple LRS (As we see in last two test cases), this implementation prints the LRS which comes 1<sup>st</sup> lexicographically.

Ukkonen's Suffix Tree Construction takes  $O(N)$  time and space to build suffix tree for a string of length  $N$  and after that finding deepest node will take  $O(N)$ .

So it is linear in time and space.

Followup questions:

1. Find all repeated substrings in given text
2. Find all unique substrings in given text
3. Find all repeated substrings of a given length
4. Find all unique substrings of a given length
5. In case of multiple LRS in text, find the one which occurs most number of times

All these problems can be solved in linear time with few changes in above implementation.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/suffix-tree-application-3-longest-repeated-substring/>

## Chapter 49

# Suffix Tree Application 2 – Searching All Patterns

Suffix Tree Application 2 - Searching All Patterns - GeeksforGeeks

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms ([KMP](#), [Rabin-Karp](#), [Naive Algorithm](#), [Finite Automata](#)) are already discussed, which can be used for this check.

Here we will discuss suffix tree based algorithm.

In the 1<sup>st</sup> Suffix Tree Application ([Substring Check](#)), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through [Substring Check](#) 1<sup>st</sup>.

In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

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

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

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

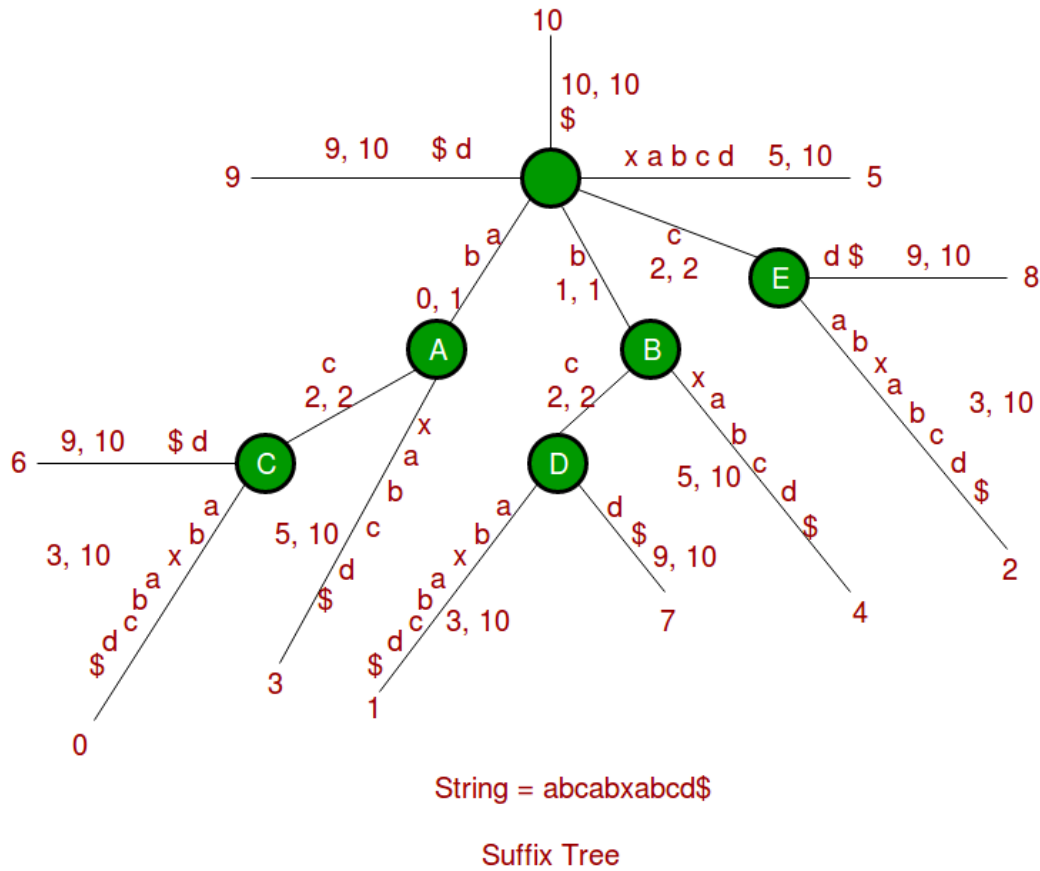
[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

Lets look at following figure:



This is suffix tree for String “abcbxabcdbcd\$”, showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.

Suffix Index of a path tells the index of a substring (starting from root) on that path.

Consider a path “bcd\$” in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd\$ are at index 7 in string.

Similarly path “bxabcdbcd\$” with suffix index 4 tells that substrings b, bx, bxa, bxab, bxabc, bxabcd, bxabcdbcd\$ are at index 4.

Similarly path “bcabxabcdbcd\$” with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxab, bcabxabc, bcabxabcdbcd, bcabxabcdbcd\$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring “b” is at indices 1, 4 and 7
- Substring “bc” is at indices 1 and 7

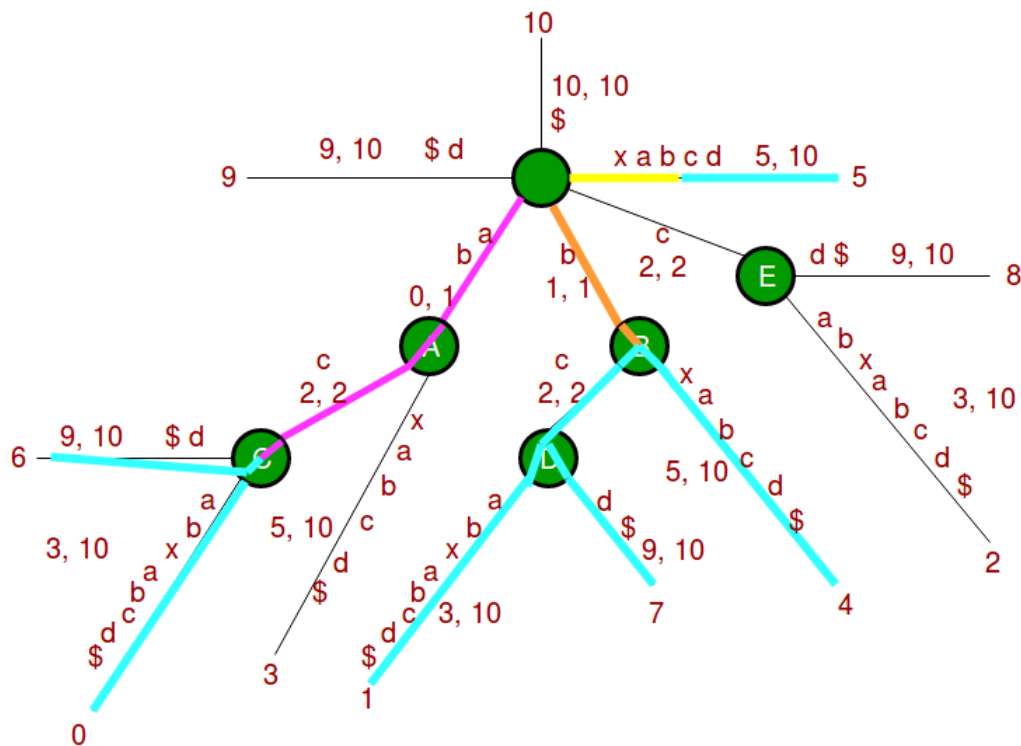
With above explanation, we should be able to see following:

- Substring “ab” is at indices 0, 3 and 6
- Substring “abc” is at indices 0 and 6

- Substring “c” is at indices 2 and 8
- Substring “xab” is at index 5
- Substring “d” is at index 9
- Substring “cd” is at index 8

Can you see how to find all the occurrences of a pattern in a string ?

- 1<sup>st</sup> of all, check if the given pattern really exists in string or not (As we did in [Substring Check](#)). For this, traverse the suffix tree against the pattern.
2. If you find pattern in suffix tree (don't fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string



Substring abc is found , subtree traversal shows that it is at indices 0 and 5

Substring xab is found, subtree traversal shows that it is at index 5

Substring **b** is found, subtree traversal shows that it is at indices 1, 4, and 7

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And find all locations of a pattern in string
#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. Lets say there are two nodes 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 its 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 its
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as 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 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; // no match
    }
    if(str[idx] == '\0')
        return 1; // match
    return 0; // more characters yet to match
}

int doTraversalToCountLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    if(n->suffixIndex > -1)
    {
        printf("\nFound at position: %d", n->suffixIndex);
        return 1;
    }
    int count = 0;
    int i = 0;
```

```
for (i = 0; i < MAX_CHAR; i++)
{
    if(n->children[i] != NULL)
    {
        count += doTraversalToCountLeaf(n->children[i]);
    }
}
return count;
}

int countLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    return doTraversalToCountLeaf(n);
}

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 == -1) //no match
            return -1;
        if(res == 1) //match
        {
            if(n->suffixIndex > -1)
                printf("\nsubstring count: 1 and position: %d",
                    n->suffixIndex);
            else
                printf("\nsubstring count: %d", countLeaf(n));
            return 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], travrse 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("\nPattern <%s> is a Substring\n", str);
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    printf("Text: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: A");
    checkForSubString("A");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AB");
    checkForSubString("AB");
    //Free the dynamically allocated memory
```

```
    freeSuffixTreeByPostOrder(root);  
  
    return 0;  
}
```

Output:

Text: GEEKSFORGEEKS, Pattern to search: GEEKS  
Found at position: 8  
Found at position: 0  
substring count: 2  
Pattern <GEEKS> is a Substring

Text: GEEKSFORGEEKS, Pattern to search: GEEK1  
Pattern <GEEK1> is NOT a Substring

Text: GEEKSFORGEEKS, Pattern to search: FOR  
substring count: 1 and position: 5  
Pattern <FOR> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AABA  
Found at position: 13  
Found at position: 9  
Found at position: 0  
substring count: 3  
Pattern <AABA> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AA  
Found at position: 16  
Found at position: 12  
Found at position: 13  
Found at position: 9  
Found at position: 0  
Found at position: 3  
Found at position: 6  
substring count: 7  
Pattern <AA> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AAE  
Pattern <AAE> is NOT a Substring

Text: AAAAAAAAAA, Pattern to search: AAAA  
Found at position: 5  
Found at position: 4  
Found at position: 3  
Found at position: 2  
Found at position: 1  
Found at position: 0  
substring count: 6  
Pattern <AAAA> is a Substring

Text: AAAAAAAAAA, Pattern to search: AA  
Found at position: 7  
Found at position: 6  
Found at position: 5  
Found at position: 4  
Found at position: 3  
Found at position: 2  
Found at position: 1  
Found at position: 0  
substring count: 8  
Pattern <AA> is a Substring

Text: AAAAAAAAAA, Pattern to search: A  
Found at position: 8  
Found at position: 7  
Found at position: 6  
Found at position: 5  
Found at position: 4  
Found at position: 3  
Found at position: 2  
Found at position: 1  
Found at position: 0  
substring count: 9  
Pattern <A> is a Substring

Text: AAAAAAAAAA, Pattern to search: AB  
Pattern <AB> 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  $M$  and then if there are  $Z$  occurrences of the pattern, it will take  $O(Z)$  to find indices of all those  $Z$  occurrences.

Overall pattern complexity is linear:  $O(M + Z)$ .

#### **A bit more detailed analysis**

How many internal nodes will there in a suffix tree of string of length  $N$  ??



Answer:  $N-1$  (Why ??)

There will be  $N$  suffixes in a string of length  $N$ .

Each suffix will have one leaf.

So a suffix tree of string of length  $N$  will have  $N$  leaves.

As each internal node has at least 2 children, an  $N$ -leaf suffix tree has at most  $N-1$  internal nodes.

If a pattern occurs  $Z$  times in string, means it will be part of  $Z$  suffixes, so there will be  $Z$  leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with  $Z$  leaves below that point will have  $Z-1$  internal nodes. A tree with  $Z$  leaves can be traversed in  $O(Z)$  time.

Overall pattern complexity is linear:  $O(M + Z)$ .

For a given pattern,  $Z$  (the number of occurrences) can be atmost  $N$ .

So worst case complexity can be:  $O(M + N)$  if  $Z$  is close/equal to  $N$  (A tree traversal with  $N$  nodes take  $O(N)$  time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/suffix-tree-application-2-searching-all-patterns/>

## Chapter 50

# Suffix Tree Application 1 – Substring Check

Suffix Tree Application 1 - Substring Check - GeeksforGeeks

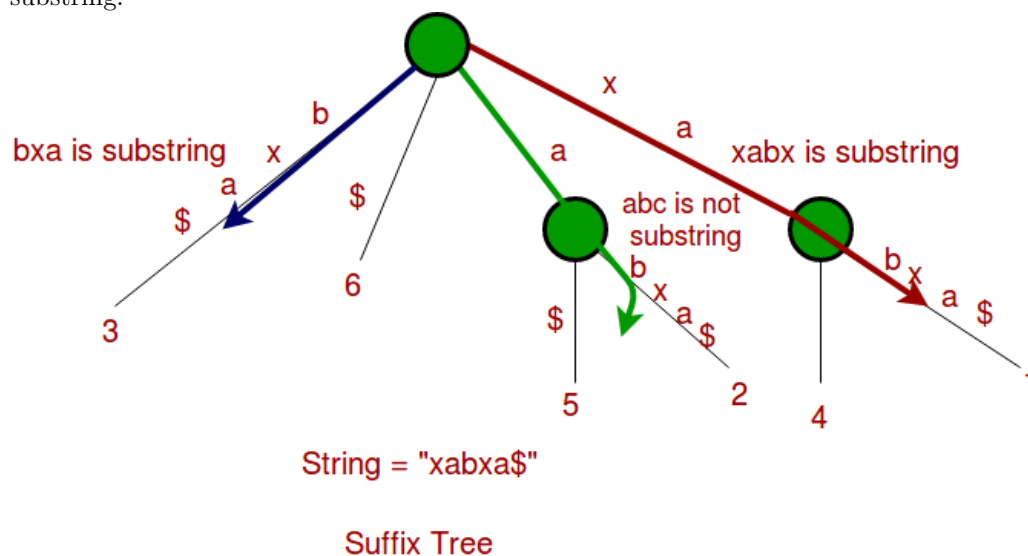
Given a text string and a pattern string, check if pattern exists in text or not.

Few pattern searching algorithms ([KMP](#), [Rabin-Karp](#), [Naive Algorithm](#), [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.



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

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

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

```
// 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. Lets say there are two nodes 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 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 existng 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 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; // no 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 slight modification in traversal algorithm discussed here, we can answer 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](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## **Source**

<https://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/>

## Chapter 51

# Ukkonen's Suffix Tree Construction – Part 6

Ukkonen's Suffix Tree Construction - Part 6 - GeeksforGeeks

This article is continuation of following five articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#), [Part 4](#) and [Part 5](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and activePoints along with an example string “abcabxabcd” where we went through all phases of building suffix tree.

Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of [Part 5](#) article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications.

There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have `SuffixTreeNode` structure to represent each node in tree. `SuffixTreeNode` structure will have following members:

- **children** – This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** – This will point to other node where current node should point via suffix link.
- **start, end** – These two will store the edge label details from parent node to current node. (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. Lets say there are two nodes A (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.

- **suffixIndex** – This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1 .

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? — Root is a special node, with no parent and so it's start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? — suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge? — Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? — If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ? — If A->children[c] is not NULL, there is a path, if NULL, no path.
- What is the character value on an edge at some given distance d from a node A ? — Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? — Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf ? — If leaf node is A on the path, then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

```
// A C program to implement 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. Lets say there are two nodes 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 represeted as 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) {
    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 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 in 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
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            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;
    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);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    // strcpy(text, "abc"); buildSuffixTree();
    // strcpy(text, "xabxac#"); buildSuffixTree();
    // strcpy(text, "xabxa"); buildSuffixTree();
    // strcpy(text, "xabxa$"); buildSuffixTree();
    strcpy(text, "abcabxabcd$"); buildSuffixTree();
    // strcpy(text, "geeksforgeeks$"); buildSuffixTree();
    // strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
    // strcpy(text, "AABAACAADAABAAABAA$"); buildSuffixTree();
    return 0;
}
```

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous [Part 5](#) article):

```
$ [10]
ab [-1]
c [-1]
abxabcd$ [0]
d$ [6]
xabcd$ [3]
b [-1]
c [-1]
abxabcd$ [1]
d$ [7]
xabcd$ [4]
c [-1]
abxabcd$ [2]
d$ [8]
d$ [9]
xabcd$ [5]
```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way:

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, [KMP](#) otherwise)

- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- [Linear Time Suffix Array Creation](#)

The above basic problems can be solved by DFS traversal on suffix tree.  
We will soon post articles on above problems and others like below:

- Build [Generalized suffix tree](#)
- Linear Time [Longest common substring problem](#)
- Linear Time [Longest palindromic substring](#)

And [More](#).

### Test you understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string “AABAA-CAADAABAAABAA\$” on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3. Following are the rules applied on five consecutive extensions in some Phase i ( $i > 5$ ), which ones are valid:
  - A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3
  - B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2
  - C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3
  - D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1
  - E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2
  - F) Rule 3, Rule 3, Rule 3, Rule 3, Rule 3
3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have it's suffix link set to another node (internal or root). Can a newly created node point to already existing internal node or not ? Can it happen that a new node created in extension j, may not get it's right suffix link in next extension j+1 and get the right one in later extensions like j+2, j+3 etc ?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

### References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

[Ukkonen's suffix tree algorithm in plain English](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## **Source**

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-6/>

## Chapter 52

# Ukkonen's Suffix Tree Construction – Part 5

Ukkonen's Suffix Tree Construction - Part 5 - GeeksforGeeks

This article is continuation of following four articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#) and [Part 4](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string “abcbxabcd” where we went through six phases of building suffix tree.

Here, we will go through rest of the phases (7 to 11) and build the tree completely.

\*\*\*\*\***Phase 7**\*\*\*\*\*

In phase 7, we read 7<sup>th</sup> character (a) from string S

- Set END to 7 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 6.



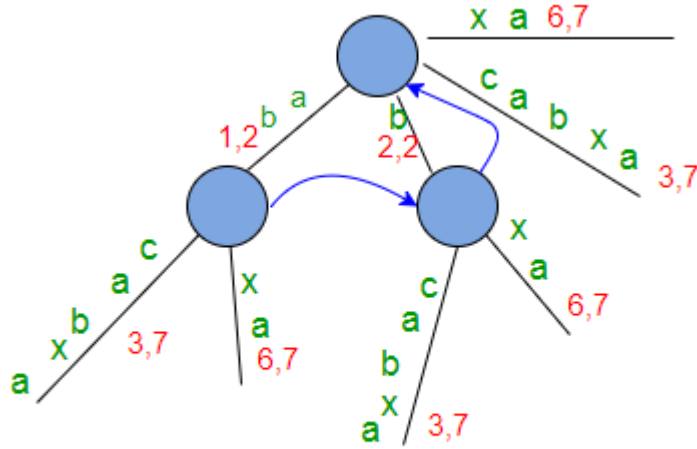


Figure 33 : Phase 7, Extension 6-Rule 1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is only 1 extension left to be performed, which is extensions 7 for suffix 'a')

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO [activePoint in previous phase was (root, x, 0)], set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**. Now activePoint becomes (root, 'a', 0).
- Check if there is an edge going out from activeNode (which is root in this phase 7) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root), here we increment activeLength from zero to 1 (**APCFER3**) and stop any further processing.
- At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 7, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).

Above Figure 33 is the resulting tree after phase 7.

\*\*\*\*\*Phase 8\*\*\*\*\*

In phase 8, we read 8<sup>th</sup> character (b) from string S

- Set END to 8 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 7 (Figure 34).

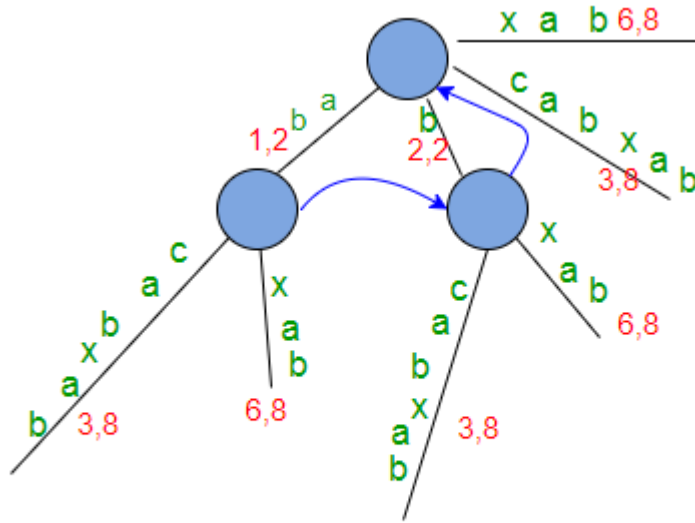


Figure 34 : Phase 8, Extension 6-Rule 1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are two extensions left to be performed, which are extensions 7 and 8 for suffixes 'ab' and 'b' respectively)

Run a loop remainingSuffixCount times (i.e. two times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 8) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 8, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 7 (remainingSuffixCount = 2)
- Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 8, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree explicitly, but they are in tree implicitly).

\*\*\*\*\***Phase 9**\*\*\*\*\*

In phase 9, we read 9<sup>th</sup> character (c) from string S

- Set END to 9 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 8.

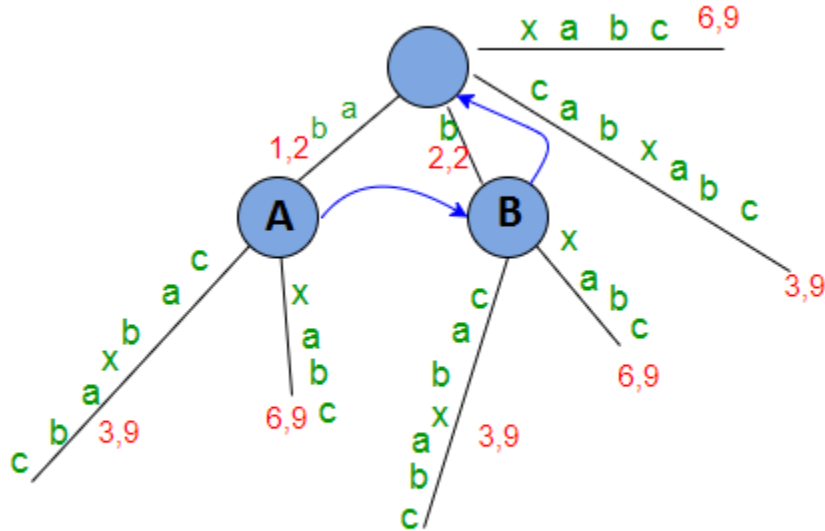


Figure 35 : Phase 9, Extension 6-Rule1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are three extensions left to be performed, which are extensions 7, 8 and 9 for suffixes 'abc', 'bc' and 'c' respectively)

Run a loop remainingSuffixCount times (i.e. three times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 9) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 9, walk down needed as activeLength(2) >= edgeLength(2). While walk down, activePoint changes to (Node A, c, 0) based on **APCFWD** (This is first time **APCFWD** is being applied in our example).
- Check if current character of string S (which is 'c') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 0 to 1 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (Node A, c, 1) and remainingSuffixCount remains set to 3 (no change in remainingSuffixCount)

At the end of phase 9, remainingSuffixCount is 3 (Three suffixes, 'abc', 'bc' and 'c', the last three, are not added explicitly in tree explicitly, but they are in tree implicitly).

\*\*\*\*\*Phase 10\*\*\*\*\*

In phase 10, we read 10<sup>th</sup> character (d) from string S

- Set END to 10 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 9.

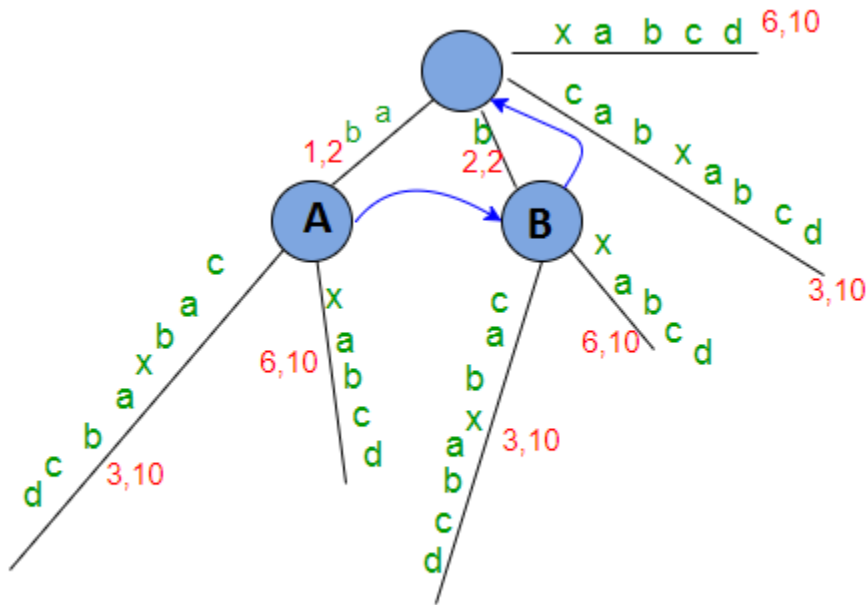


Figure 36 : Phase 10, Extension 6-Rule1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 4 here, i.e. there are four extensions left to be performed, which are extensions 7, 8, 9 and 10 for suffixes 'abcd', 'bcd', 'cd' and 'd' respectively)

Run a loop remainingSuffixCount times (i.e. four times) as below:

- Check if there is an edge going out from activeNode (Node A) for the activeEdge(c). If not, create a leaf edge. If present, walk down. In our example, edge 'c' is present going out of activeNode (Node A).
- Do a walk down (The trick 1 – skip/count) if necessary. In current Extension 7, no walk down needed as activeLength < edgeLength.
- Check if current character of string S (which is 'd') is already present after the activePoint. If not, rule 2 will apply. In our example, there is no path starting with 'd' going out of activePoint, so we create a leaf edge with label 'd'. Since activePoint ends in the middle of an edge, we will create a new internal node just after the activePoint (Rule 2)

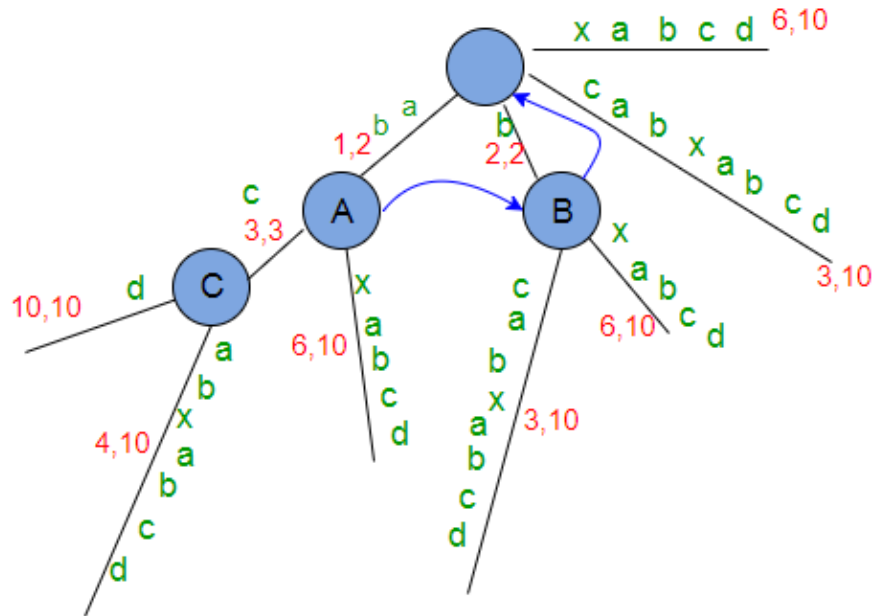


Figure 37 : Phase 10, Extension 7-Rule2 applied.  
New leaf edge and new internal node created

- Decrement the remainingSuffixCount by 1 (from 4 to 3) as suffix “abcd” added in tree.
- Now activePoint will change for next extension 8. Current activeNode is an internal node (Node A), so there must be a suffix link from there and we will follow that to get new activeNode and that’s going to be ‘Node B’. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (Node B, c, 1).
- Now in extension 8 (here we will add suffix ‘bcd’), while adding character ‘d’ after the current activePoint, exactly same logic will apply as previous extension 7. In previous extension 7, we added character ‘d’ at activePoint (Node A, c, 1) and in current extension 8, we are going to add same character ‘d’ at activePoint (Node B, c, 1). So logic will be same and here we a new leaf edge with label ‘d’ and a new internal node will be created. And the new internal node (C) of previous extension will point to the new node (D) of current extension via suffix link.

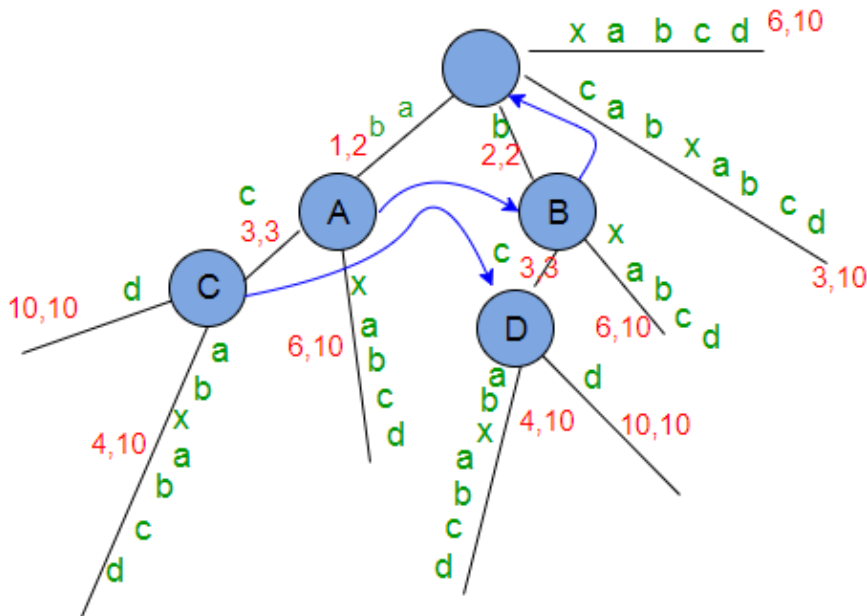


Figure 38 : Phase 10, Extension 8-Rule2 applied.  
 New leaf edge created and a new internal node created.  
 Also the internal node C created in previous extension 7, pointing  
 to the internal node D via suffix link

- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix “bcd” added in tree.
- Now activePoint will change for next extension 9. Current activeNode is an internal node (Node B), so there must be a suffix link from there and we will follow that to get new activeNode and that is ‘Root Node’. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (root, c, 1).
- Now in extension 9 (here we will add suffix ‘cd’), while adding character ‘d’ after the current activePoint, exactly same logic will apply as previous extensions 7 and 8. Note that internal node D created in previous extension 8, now points to internal node E (created in current extension) via suffix link.

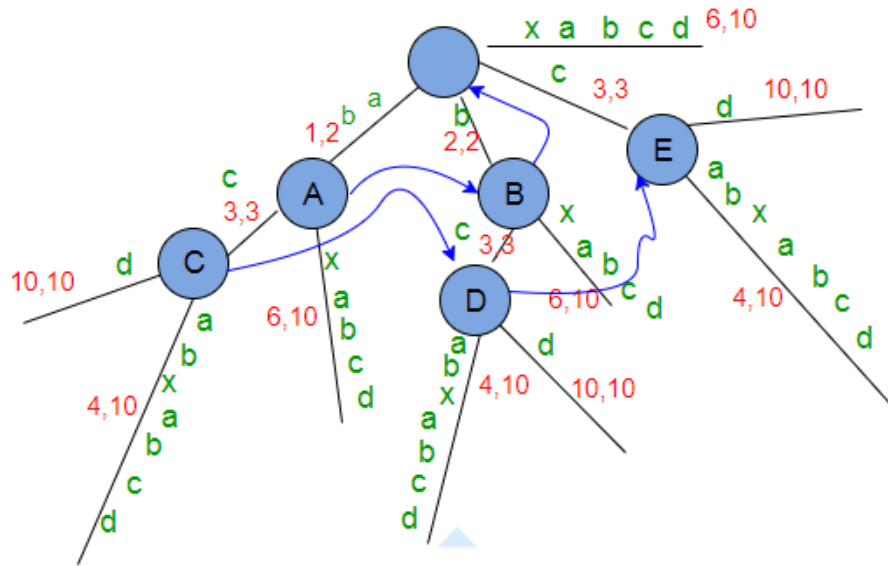


Figure 39 : Phase 10, Extension 9-Rule 2 applied.  
 New leaf edge created and a new internal node created.  
 Also the internal node D created in previous extension 8, pointing to the  
 internal node E via suffix link

- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix “cd” added in tree.
- Now activePoint will change for next extension 10. Current activeNode is root and activeLength is 1, based on **APCFER2C1**, activeNode will remain ‘root’, activeLength will be decremented by 1 (from 1 to ZERO) and activeEdge will be ‘d’. So new activePoint is (root, d, 0).
- Now in extension 10 (here we will add suffix ‘d’), while adding character ‘d’ after the current activePoint, there is no edge starting with d going out of activeNode root, so a new leaf edge with label d is created (Rule 2). Note that internal node E created in previous extension 9, now points to root node via suffix link (as no new internal node created in this extension).

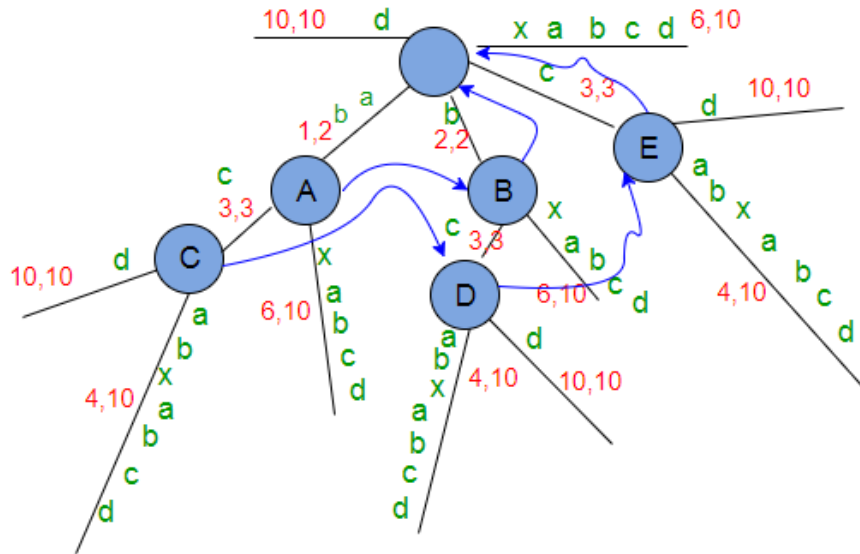


Figure 40 : Phase 10, Extension 10-Rule 2 applied. New leaf edge created. Also the internal node E created in previous extension 9, pointing to root node via suffix link

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix “d” added in tree. That means no more suffix is there to add and so the phase 10 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character d was not seen before in string S so far)
- activePoint for next phase 11 is (root, d, 0).

We see following facts in Phase 10:

- Internal Nodes connected via suffix links have exactly same tree below them, e.g. In above Figure 40, A and B have same tree below them, similarly C, D and E have same tree below them.
- Due to above fact, in any extension, when current activeNode is derived via suffix link from previous extension's activeNode, then exactly same extension logic apply in current extension as previous extension. (In Phase 10, same extension logic is applied in extensions 7, 8 and 9)
- If a new internal node gets created in extension j of any phase i, then this newly created internal node will get it's suffix link set by the end of next extension j+1 of same phase i. e.g. node C got created in extension 7 of phase 10 (Figure 37) and it got it's suffix link set to node D in extension 8 of same phase 10 (Figure 38). Similarly node D got created in extension 8 of phase 10 (Figure 38) and it got its suffix link set to node E in extension 9 of same phase 10 (Figure 39). Similarly node E got created in extension 9 of phase 10 (Figure 39) and it got its suffix link set to root in extension 10 of same phase 10 (Figure 40).



- Based on above fact, every internal node will have a suffix link to some other internal node or root. Root is not an internal node and it will not have suffix link.

\*\*\*\*\*Phase 11\*\*\*\*\*

In phase 11, we read 11<sup>th</sup> character (\$) from string S

- Set END to 11 (This will do extensions 1 to 10) – because we have 10 leaf edges so far by the end of previous phase 10.

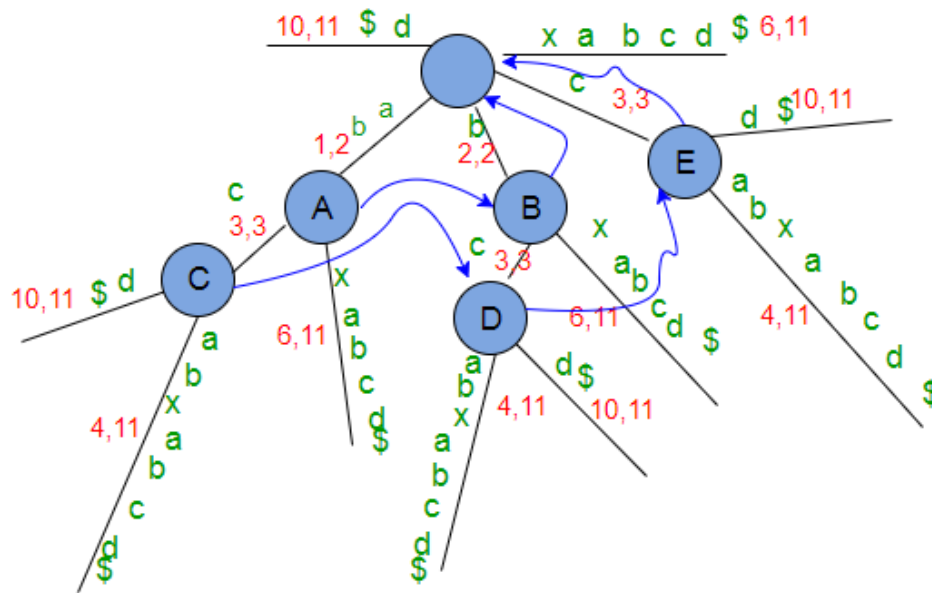


Figure 41 : Phase 11, Extension 10- Rule 1 applied

- Increment remainingSuffixCount by 1 (from 0 to 1), i.e. there is only one suffix '\$' to be added in tree.
- Since activeLength is ZERO, activeEdge will change to current character '\$' of string S being processed (**APCFALZ**).
- There is no edge going out from activeNode root, so a leaf edge with label '\$' will be created (Rule 2).

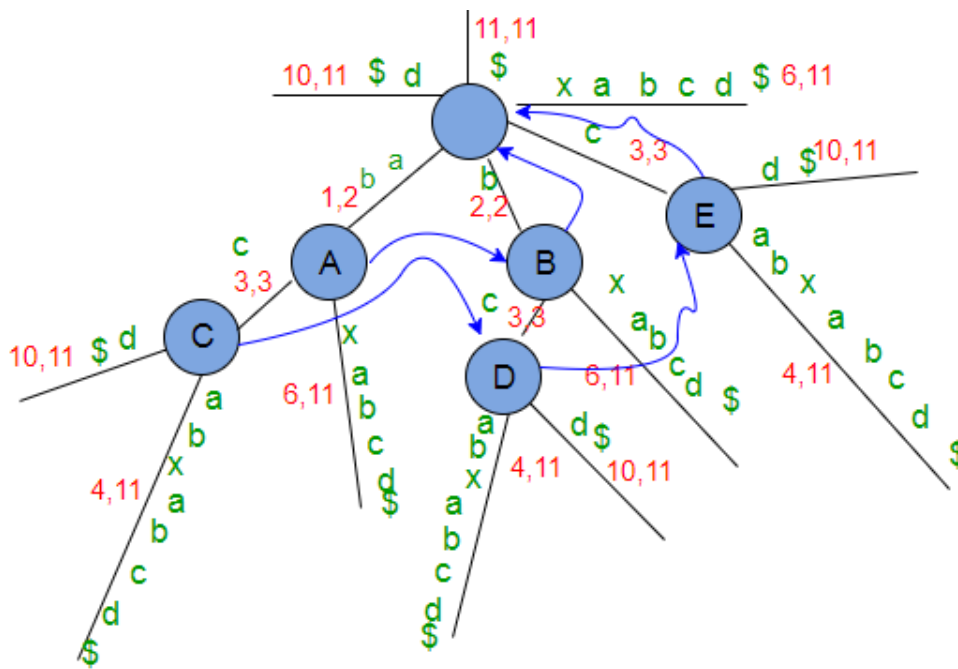
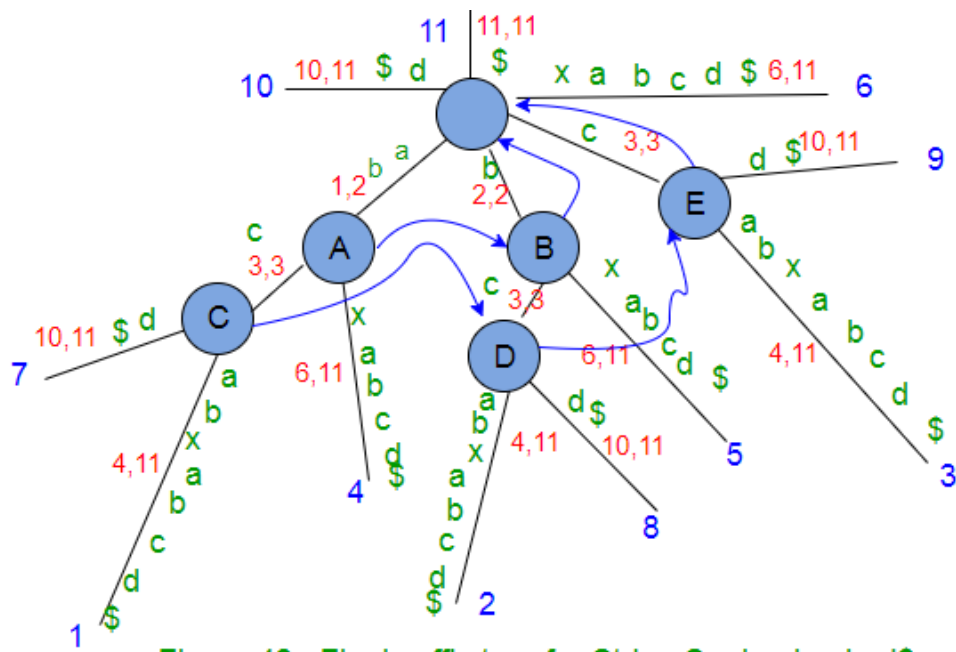


Figure 42 : Phase 11, Extension 11- Rule 2 applied

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix “\$” added in tree. That means no more suffix is there to add and so the phase 11 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character \$ was not seen before in string S so far)

Now we have added all suffixes of string ‘abcabxabcd\$’ in suffix tree. There are 11 leaf ends in this tree and labels on the path from root to leaf end represents one suffix. Now the only one thing left is to assign a number (suffix index) to each leaf end and that number would be the suffix starting position in the string S. This can be done by a DFS traversal on tree. While DFS traversal, keep track of label length and when a leaf end is found, set the suffix index as “stringSize – labelSize + 1”. Indexed suffix tree will look like below:

Figure 43 : Final suffix tree for String  $S=abcbabxabc d\$$ 

In above Figure, suffix indices are shown as character position starting with 1 (It's not zero indexed). In code implementation, suffix index will be set as zero indexed, i.e. where we see suffix index  $j$  (1 to  $m$  for string of length  $m$ ) in above figure, in code implementation, it will be  $j-1$  (0 to  $m-1$ )

And we are done !!!!

### Data Structure to represent suffix tree

How to represent the suffix tree?? There are nodes, edges, labels and suffix links and indices.

Below are some of the operations/query we will be doing while building suffix tree and later on while using the suffix tree in different applications/usages:

- What length of path label on some edge?
- What is the path label on some edge?
- Check if there is an outgoing edge for a given character from a node.
- What is the character value on an edge at some given distance from a node?
- Where an internal node is pointing via suffix link?
- What is the suffix index on a path from root to leaf?
- Check if a given string present in suffix tree (as substring, suffix or prefix)?

We may think of different data structures which can fulfil these requirements.

In the next [Part 6](#), we will discuss the data structure we will use in our code implementation and the code as well.

### References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

[Ukkonen's suffix tree algorithm in plain English](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-5/>

## Chapter 53

# Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction - Part 4 - GeeksforGeeks

This article is continuation of following three articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

Please go through [Part 1](#), [Part 2](#) and [Part 3](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string “abcabxabcd” where we went through four phases of building suffix tree.

Let's revisit those four phases we have seen already in [Part 3](#), in terms of trick 2, trick 3 and activePoint.

- activePoint is initialized to (root, NULL, 0), i.e. activeNode is root, activeEdge is NULL (for easy understanding, we are giving character value to activeEdge, but in code implementation, it will be index of the character) and activeLength is ZERO.
- The global variable END and remainingSuffixCount are initialized to ZERO

\*\*\*\*\***Phase 1**\*\*\*\*\*

In Phase 1, we read 1<sup>st</sup> character (a) from string S

- Set END to 1
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.

- Check if there is an edge going out from activeNode (which is root in this phase 1) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created (Rule 2).
- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, a, 0)

At the end of phase 1, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 20 in [Part 3](#) is the resulting tree after phase 1.

\*\*\*\*\*Phase 2\*\*\*\*\*

In Phase 2, we read 2<sup>nd</sup> character (b) from string S

Set END to 2 (This will do extension 1)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'b'). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 2) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, b, 0)

At the end of phase 2, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 22 in [Part 3](#) is the resulting tree after phase 2.

\*\*\*\*\*Phase 3\*\*\*\*\*

In Phase 3, we read 3<sup>rd</sup> character (c) from string S

Set END to 3 (This will do extensions 1 and 2)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'c'). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, c, 0)

At the end of phase 3, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 25 in [Part 3](#) is the resulting tree after phase 3.

## \*\*\*\*\*Phase 4\*\*\*\*\*

In Phase 4, we read 4<sup>th</sup> character (a) from string S

Set END to 4 (This will do extensions 1, 2 and 3)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down (The trick 1 – skip/count). In our example, edge 'a' is present going out of activeNode (i.e. root). No walk down needed as activeLength < edgeLength. We increment activeLength from zero to 1 (**APCFER3**) and stop any further processing (Rule 3).
- At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 4, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).

Figure 28 in Part 3 is the resulting tree after phase 4.

Revisiting completed for 1<sup>st</sup> four phases, we will continue building the tree and see how it goes.

## \*\*\*\*\*Phase 5\*\*\*\*\*

In phase 5, we read 5<sup>th</sup> character (b) from string S

Set END to 5 (This will do extensions 1, 2 and 3). See Figure 29 shown below.

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are 2 extension left to be performed, which are extensions 4 and 5. Extension 4 is supposed to add suffix "ab" and extension 5 is supposed to add suffix "b" in tree)

Run a loop remainingSuffixCount times (i.e. two times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 5, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 4 (remainingSuffixCount = 2)
- Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

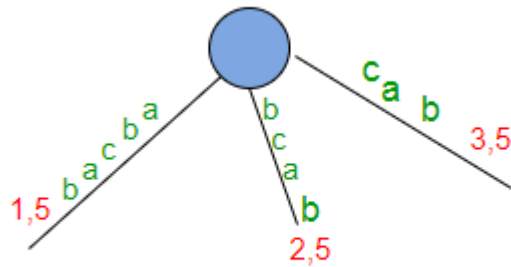


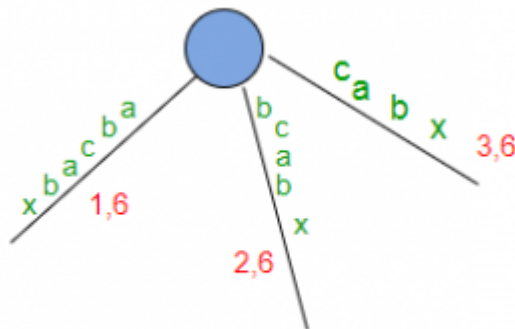
Figure 29 : Phase 5

At the end of phase 5, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree, but they are in tree implicitly).

\*\*\*\*\*Phase 6\*\*\*\*\*

In phase 6, we read 6<sup>th</sup> character (x) from string S

Set END to 6 (This will do extensions 1, 2 and 3)



Phase 6, Extension 3

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are 3 extension left to be performed, which are extensions 4, 5 and 6 for suffixes "abx", "bx" and "x" respectively)

Run a loop remainingSuffixCount times (i.e. three times) as below:

- While extension 4, the activePoint is (root, a, 2) which points to 'b' on edge starting with 'a'.
- In extension 4, current character 'x' from string S doesn't match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix "abx" added in tree.



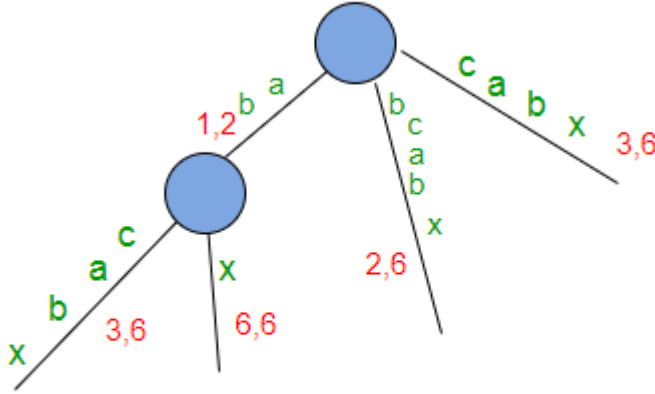


Figure 30 : Phase 6, Extension 4  
Rule 2 applied- Created a leaf edge and  
also a new internal node

Now activePoint will change after applying rule 2. Three other cases, (**APCFER3**, **APCFWD** and **APCFALZ**) where activePoint changes, are already discussed in [Part 3](#).

**activePoint change for extension rule 2 (APCFER2):**

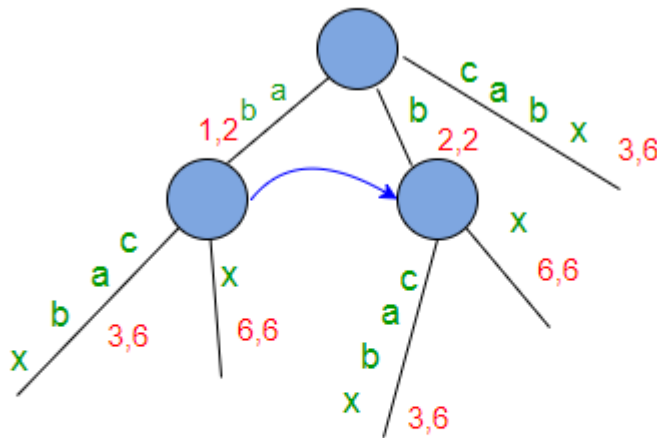
**Case 1 (APCFER2C1):** If activeNode is root and activeLength is greater than ZERO, then decrement the activeLength by 1 and activeEdge will be set “ $S[i - \text{remainingSuffixCount} + 1]$ ” where  $i$  is current phase number. Can you see why this change in activePoint? Look at current extension we just discussed above for phase 6 ( $i=6$ ) again where we added suffix “abx”. There activeLength is 2 and activeEdge is ‘a’. Now in next extension, we need to add suffix “bx” in the tree, i.e. path label in next extension should start with ‘b’. So ‘b’ (the 5<sup>th</sup> character in string  $S$ ) should be active edge for next extension and index of  $b$  will be “ $i - \text{remainingSuffixCount} + 1$ ” ( $6 - 2 + 1 = 5$ ). activeLength is decremented by 1 because activePoint gets closer to root by length 1 after every extension.

What will happen If activeNode is root and activeLength is ZERO? This case is already taken care by **APCFALZ**.

**Case 2 (APCFER2C2):** If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and activeEdge. Can you see why this change in activePoint? This is because: If two nodes are connected by a suffix link, then labels on all paths going down from those two nodes, starting with same character, will be exactly same and so for two corresponding similar point on those paths, activeEdge and activeLength will be same and the two nodes will be the activeNode. Look at Figure 18 in [Part 2](#). Let’s say in phase  $i$  and extension  $j$ , suffix ‘xAabcdedg’ was added in tree. At that point, let’s say activePoint was (Node-V, a, 7), i.e. point ‘g’. So for next extension  $j+1$ , we would add suffix ‘Aabcdefg’ and for that we need to traverse 2<sup>nd</sup> path shown in Figure 18. This can be done by following suffix link from current activeNode  $v$ . Suffix link takes us to the path to be traversed somewhere in between [Node

$s(v)$ ] below which the path is exactly same as how it was below the previous activeNode  $v$ . As said earlier, “activePoint gets closer to root by length 1 after every extension”, this reduction in length will happen above the node  $s(v)$  but below  $s(v)$ , no change at all. So when activeNode is not root in current extension, then for next extension, only activeNode changes (No change in activeEdge and activeLength).

- At this point in extension 4, current activePoint is (root, a, 2) and based on **APCFER2C1**, new activePoint for next extension 5 will be (root, b, 1)
- Next suffix to be added is ‘bx’ (with remainingSuffixCount 2).
- Current character ‘x’ from string S doesn’t match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.  
Suffix link is also created from previous internal node (of extension 4) to the new internal node created in current extension 5.
- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix “bx” added in tree.

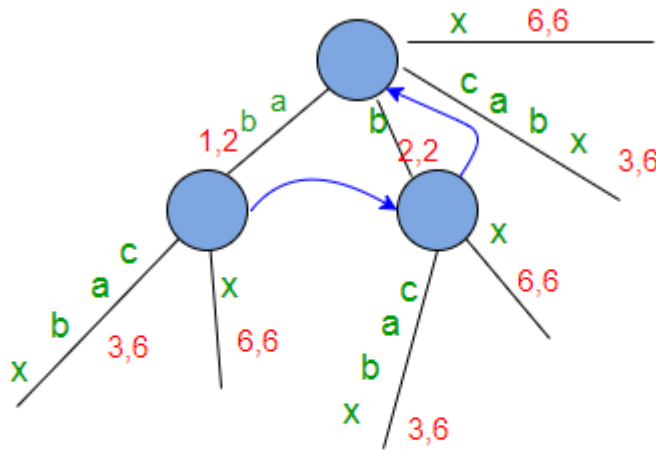


**Figure 31 : Phase 6, Extension 5-Rule2 applied  
Created a leaf edge, a new internal node and suffix link  
from previous internal node of extension 4 to the current  
newly internal node**

- At this point in extension 5, current activePoint is (root, b, 1) and based on **APCFER2C1** new activePoint for next extension 6 will be (root, x, 0)
- Next suffix to be added is ‘x’ (with remainingSuffixCount 1).
- In the next extension 6, character x will not match to any existing edge from root, so a new edge with label x will be created from root node. Also suffix link from previous

extension's internal node goes to root (as no new internal node created in current extension 6).

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "x" added in tree



**Figure 32 : Phase 6, Extension 6-Rule2 applied.**  
**Created a leaf edge and suffix link from previous internal node of extension 5 to root node(as no new internal node created in extension 6, so suffix link goes to roots)**

This completes the phase 6.

Note that phase 6 has completed all its 6 extensions (Why? Because the current character *c* was not seen in string so far, so rule 3, which stops further extensions never got chance to get applied in phase 6) and so the tree generated after phase 6 is a true suffix tree (i.e. not an implicit tree) for the characters 'abcabx' read so far and it has all suffixes explicitly in the tree.

While building the tree above, following facts were noticed:

- A newly created internal node in extension *i*, points to another internal node or root (if activeNode is root in extension *i*+1) by the end of extension *i*+1 via suffix link (Every internal node MUST have a suffix link pointing to another internal node or root)
- Suffix link provides short cut while searching path label end of next suffix
- With proper tracking of activePoints between extensions/phases, unnecessary walk-down from root can be avoided.

We will go through rest of the phases (7 to 11) in [Part 5](#) and build the tree completely and after that, we will see the code for the algorithm in [Part 6](#).

**References:**

<http://web.stanford.edu/~mjkay/gusfield.pdf>  
Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-4/>

## Chapter 54

# Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction - Part 3 - GeeksforGeeks

This article is continuation of following two articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

Please go through [Part 1](#) and [Part 2](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks.

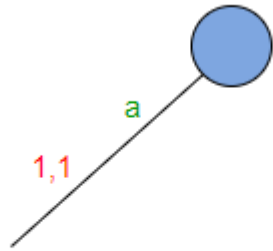
Here we will take string  $S = \text{"abcabxabcd"}$  as an example and go through all the things step by step and create the tree.

We will add  $\$$  (discussed in [Part 1](#) why we do this) so string  $S$  would be  $\text{"abcabxabcd\$"}$ .

While building suffix tree for string  $S$  of length  $m$ :

- There will be  $m$  phases 1 to  $m$  (one phase for each character)  
In our current example,  $m$  is 11, so there will be 11 phases.
- First phase will add first character 'a' in the tree, second phase will add second character 'b' in tree, third phase will add third character 'c' in tree, .....,  $m^{\text{th}}$  phase will add  $m^{\text{th}}$  character in tree (This makes Ukkonen's algorithm an online algorithm)
- Each phase  $i$  will go through at-most  $i$  extensions (from 1 to  $i$ ). If current character being added in tree is not seen so far, all  $i$  extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase  $i$  will complete early (as soon as Extension Rule 3 applies) without going through all  $i$  extensions
- There are three extension rules (1, 2 and 3) and each extension  $j$  (from 1 to  $i$ ) of any phase  $i$  will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge
- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)

- Rule 3 ends the current phase (when current character is found in current edge being traversed)
- Phase 1 will read first character from the string, will go through 1 extension.  
**(In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices – The Edge-label compression discussed in Part 2)**  
 Extension 1 will add suffix “a” in tree. We start from root and traverse path with label ‘a’. There is no path from root, going out with label ‘a’, so create a leaf edge (Rule 2).



**Figure 20 : Phase 1, extension 1-Rule2 applied. Created a leaf edge(1,1) Phase 1 completes here**

Phase 1 completes with the completion of extension 1 (As a phase  $i$  has at most  $i$  extensions)

For any string, Phase 1 will have only one extension and it will always follow Rule 2.

- Phase 2 will read second character, will go through at least 1 and at most 2 extensions. In our example, phase 2 will read second character ‘b’. Suffixes to be added are “ab” and “b”.

Extension 1 adds suffix “ab” in tree.

Path for label ‘a’ ends at leaf edge, so add ‘b’ at the end of this edge.

Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

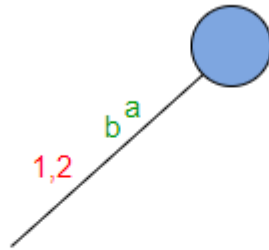


Figure 21 : Phase 2, Extension1-Rule1 applied extended the leaf edge from (1,1) to (1,2)

Extension 2 adds suffix “b” in tree. There is no path from root, going out with label ‘b’, so creates a leaf edge (Rule 2).

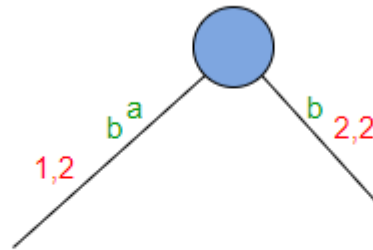


Figure 22 : Phase 2, Extension2-Rule2 applied  
Created a leaf edge(2,2)  
Phase 2 completes here

Phase 2 completes with the completion of extension 2.

Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions. In our example, phase 3 will read third character ‘c’. Suffixes to be added are “abc”, “bc” and “c”.

Extension 1 adds suffix “abc” in tree.

Path for label ‘ab’ ends at leaf edge, so add ‘c’ at the end of this edge.

Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

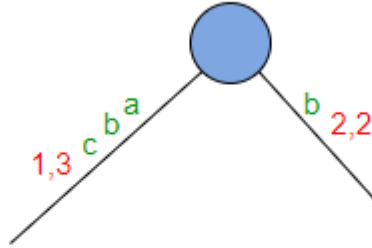


Figure 23 : Phase 3, Extension1-Rule1 applied  
Extended the leaf edge from (1,2) to (1,3)

Extension 2 adds suffix “bc” in tree.

Path for label ‘b’ ends at leaf edge, so add ‘c’ at the end of this edge.

Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

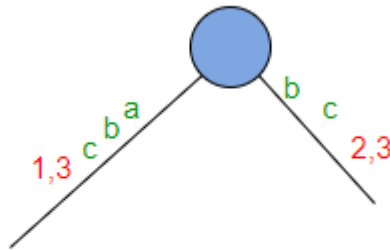


Figure 24 : Phase 3, Extension2-Rule1 applied  
Extended the leaf edge from (2,2) to (2,3)

Extension 3 adds suffix “c” in tree. There is no path from root, going out with label ‘c’, so creates a leaf edge (Rule 2).



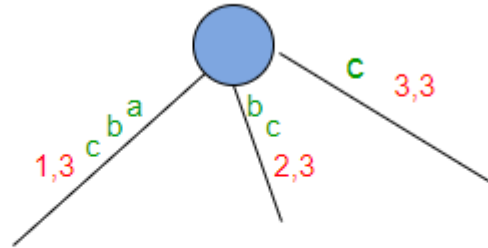


Figure 25 : Phase 3, Extension3-Rule2 applied  
Created a leaf edge(3,3)  
Phase 3 completes here

Phase 3 completes with the completion of extension 3.

Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions. In our example, phase 4 will read fourth character 'a'. Suffixes to be added are "abca", "bca", "ca" and "a".

Extension 1 adds suffix "abca" in tree.

Path for label 'abc' ends at leaf edge, so add 'a' at the end of this edge.

Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

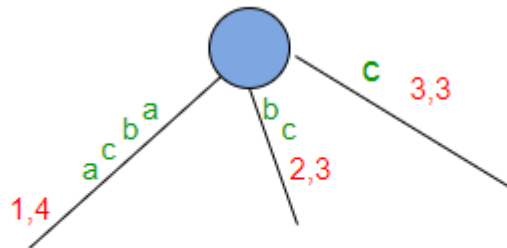


Figure 26 : Phase 4, Extension 1  
Rule 1 applied

Extension 2 adds suffix "bca" in tree.

Path for label 'bc' ends at leaf edge, so add 'a' at the end of this edge.

Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

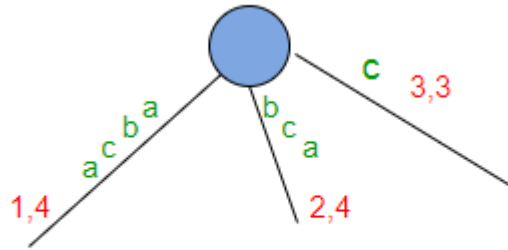


Figure 27 : Phase 4, Extension 2  
Rule 1 applied

Extension 3 adds suffix “ca” in tree.

Path for label ‘c’ ends at leaf edge, so add ‘a’ at the end of this edge.

Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

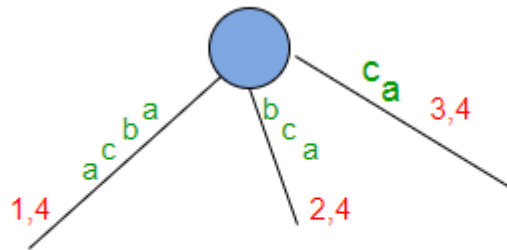


Figure 28 : Phase 4, Extension 3  
Rule 1 applied

Extension 4 adds suffix “a” in tree.

Path for label ‘a’ exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2). This is an example of implicit suffix tree. Here suffix “a” is not seen explicitly (because it doesn’t end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.

Phase 4 completes as soon as Rule 3 is applied while Extension 4.

Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

1. At the end of any phase  $i$ , there are at most  $i$  leaf edges (if  $i^{\text{th}}$  character is not seen so far, there will be  $i$  leaf edges, else there will be less than  $i$  leaf edges).  
e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).

2. After completing phase  $i$ , “end” indices of all leaf edges are  $i$ . How do we implement this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the “end” index? Answer is “NO”.

For this, we will maintain a global variable (say “END”) and we will just increment this global variable “END” and all leaf edge end indices will point to this global variable. So this way, if we have  $j$  leaf edges after phase  $i$ , then in phase  $i+1$ , first  $j$  extensions (1 to  $j$ ) will be done by just incrementing variable “END” by 1 (END will be  $i+1$  at the point).

Here we just implemented the trick 3 – **Once a leaf, always a leaf**. This trick processes all the  $j$  leaf edges (i.e. extension 1 to  $j$ ) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to  $j$ ). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.

3. In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are  $j$  leaf edges after phase  $i$ , then in phase  $i+1$ , first  $j$  extensions will follow Rule 1 and will be done in constant time using trick 3. There are  $i+1-j$  extensions yet to be performed. For these extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase  $i$  is completed.

If previous phase  $i$  went through all the  $i$  extensions (when  $i^{\text{th}}$  character is unique so far), then in next phase  $i+1$ , trick 3 will take care of first  $i$  suffixes (the  $i$  leaf edges) and then extension  $i+1$  will start from root node and it will insert just one character  $[(i+1)^{\text{th}}]$  suffix in tree by creating a leaf edge using Rule 2.

If previous phase  $i$  completes early (and this will happen if and only if rule 3 applies – when  $i^{\text{th}}$  character is already seen before), say at  $j^{\text{th}}$  extension (i.e. rule 3 is applied at  $j^{\text{th}}$  extension), then there are  $j-1$  leaf edges so far.

We will state few more facts (which may be a repeat, but we want to make sure it's clear to you at this point) here based on discussion so far:

- Phase 1 starts with Rule 2, all other phases start with Rule 1
- Any phase ends with either Rule 2 or Rule 3
- Any phase  $i$  may go through a series of  $j$  extensions ( $1 \leq j \leq i$ ). In these  $j$  extensions, first  $p$  ( $0 \leq p < i$ ) extensions will follow Rule 1, next  $q$  ( $0 \leq q \leq i-p$ ) extensions will follow Rule 2 and next  $r$  ( $0 \leq r \leq 1$ ) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3
- In a phase  $i$ ,  $p + q + r \leq i$
- At the end of any phase  $i$ , there will be  $p+q$  leaf edges and next phase  $i+1$  will go through Rule 1 for first  $p+q$  extensions

In the next phase  $i+1$ , trick 3 (Rule 1) will take care of first  $j-1$  suffixes (the  $j-1$  leaf edges), then extension  $j$  will start where we will add  $j^{\text{th}}$  suffix in tree. For this, we need to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse from root node and match tree edges against the  $j^{\text{th}}$  suffix being added character by

character? This will take time and overall algorithm will not be linear. `activePoint` comes to the rescue here.

In previous phase  $i$ , while  $j^{\text{th}}$  extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where  $i^{\text{th}}$  character being added was found in tree already and Rule 3 applied,  $j^{\text{th}}$  extension of phase  $i+1$  will start exactly from the same point and we start matching path against  $(i+1)^{\text{th}}$  character. `activePoint` helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1<sup>st</sup>  $p$  extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where `activePoint` tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, `activePoint` is set to right location already (with one exception case **APCFALZ** discussed below) and at the end of current extension, we reset `activePoint` as appropriate so that next extension (of same phase or next phase) where a traversal is required, `activePoint` points to the right place already.

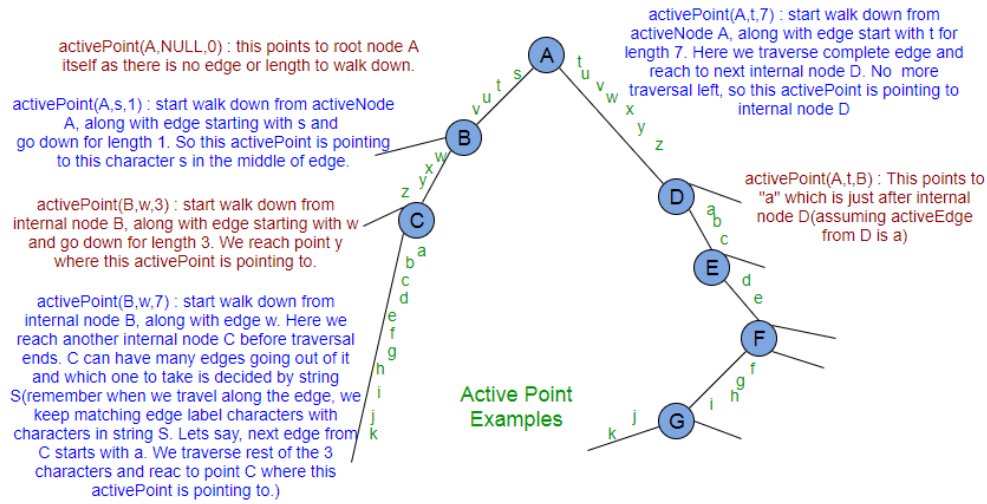
**activePoint:** This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, `activePoint` is set to root. Other extension will get `activePoint` set correctly by previous extension (with one exception case **APCFALZ** discussed below) and it is the responsibility of current extension to reset `activePoint` appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).

To accomplish this, we need a way to store `activePoint`. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.

**activeNode:** This could be root node or an internal node.

**activeEdge:** When we are on root node or internal node and we need to walk down, we need to know which edge to choose. `activeEdge` will store that information. In case, `activeNode` itself is the point from where traversal starts, then `activeEdge` will be set to next character being processed in next phase.

**activeLength:** This tells how many characters we need to walk down (on the path represented by `activeEdge`) from `activeNode` to reach the `activePoint` where traversal starts. In case, `activeNode` itself is the point from where traversal starts, then `activeLength` will be ZERO.



After phase  $i$ , if there are  $j$  leaf edges then in phase  $i+1$ , first  $j$  extensions will be done by trick 3. activePoint will be needed for the extensions from  $j+1$  to  $i+1$  and activePoint may or may not change between two extensions depending on the point where previous extension ends.

**activePoint change for extension rule 3 (APCFER3):** When rule 3 applies in any phase  $i$ , then before we move on to next phase  $i+1$ , we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string  $S$  is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLength is increased by 1 (because of matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase  $i+1$ .

**activePoint change for walk down (APCFWD):** activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is  $(A, s, 11)$  in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode  $A$ , other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLength as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:

$(A, s, 11) \rightarrow (B, w, 7) \rightarrow (C, a, 3)$

All above three activePoints refer to same point 'c'

Let's take another example.

If activePoint is  $(D, a, 11)$  at the start of an extension, then while walk down, below is the sequence of changes in activePoint:

$(D, a, 10) \rightarrow (E, d, 7) \rightarrow (F, f, 5) \rightarrow (G, j, 1)$

All above activePoints refer to same point 'k'.

If activePoints are  $(A, s, 3)$ ,  $(A, t, 5)$ ,  $(B, w, 1)$ ,  $(D, a, 2)$  etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.

The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

**activePoint change for Active Length ZERO (APCFALZ):** Let's consider an activePoint (A, s, 0) in the above activePoint example figure. And let's say current character being processed from string S is 'x' (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. 'x', because there is no walk down needed here (as activeLength is ZERO) and so next character we look for is current character being processed.

4. While code implementation, we will loop through all the characters of string S one by one. Each loop for  $i^{\text{th}}$  character will do processing for phase i. Loop will run one or more time depending on how many extensions are left to be performed (Please note that in a phase  $i+1$ , we don't really have to perform all  $i+1$  extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

We will continue our discussion in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

#### References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

[Ukkonen's suffix tree algorithm in plain English](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-3/>

## Chapter 55

# Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction - Part 2 - GeeksforGeeks

In [Ukkonen's Suffix Tree Construction – Part 1](#), we have seen high level Ukkonen's Algorithm. This 2<sup>nd</sup> part is continuation of [Part 1](#).

Please go through [Part 1](#), before looking at current article.

In Suffix Tree Construction of string  $S$  of length  $m$ , there are  $m$  phases and for a phase  $j$  ( $1 \leq j \leq m$ ), we add  $j^{\text{th}}$  character in tree built so far and this is done through  $j$  extensions. All extensions follow one of the three extension rules (discussed in [Part 1](#)).

To do  $j^{\text{th}}$  extension of phase  $i+1$  (adding character  $S[i+1]$ ), we first need to find end of the path from the root labelled  $S[j..i]$  in the current tree. One way is start from root and traverse the edges matching  $S[j..i]$  string. This will take  $O(m^3)$  time to build the suffix tree. Using few observations and implementation tricks, it can be done in  $O(m)$  which we will see now.

### Suffix links

For an internal node  $v$  with path-label  $xA$ , where  $x$  denotes a single character and  $A$  denotes a (possibly empty) substring, if there is another node  $s(v)$  with path-label  $A$ , then a pointer from  $v$  to  $s(v)$  is called a suffix link.

If  $A$  is empty string, suffix link from internal node will go to root node.

There will not be any suffix link from root node (As it's not considered as internal node).

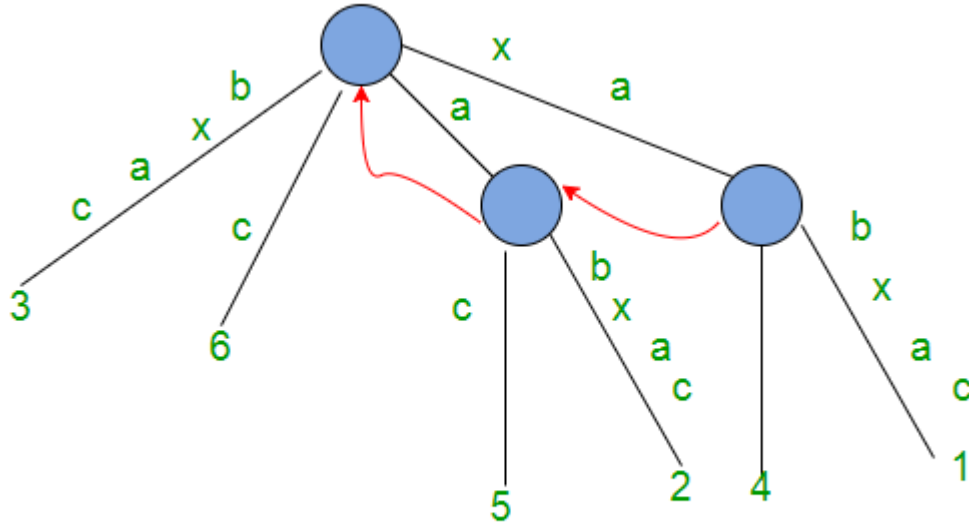


Figure 15 : Suffix links in red arrows

In extension  $j$  of some phase  $i$ , if a new internal node  $v$  with path-label  $xA$  is added, then in extension  $j+1$  in the same phase  $i$ :

- Either the path labelled  $A$  already ends at an internal node (or root node if  $A$  is empty)
- OR a new internal node at the end of string  $A$  will be created

In extension  $j+1$  of same phase  $i$ , we will create a suffix link from the internal node created in  $j^{\text{th}}$  extension to the node with path labelled  $A$ .

So in a given phase, any newly created internal node (with path-label  $xA$ ) will have a suffix link from it (pointing to another node with path-label  $A$ ) by the end of the next extension.

In any implicit suffix tree  $T_i$  after phase  $i$ , if internal node  $v$  has path-label  $xA$ , then there is a node  $s(v)$  in  $T_i$  with path-label  $A$  and node  $v$  will point to node  $s(v)$  using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

#### How suffix links are used to speed up the implementation?

In extension  $j$  of phase  $i+1$ , we need to find the end of the path from the root labelled  $S[j..i]$  in the current tree. One way is start from root and traverse the edges matching  $S[j..i]$  string. Suffix links provide a short cut to find end of the path.



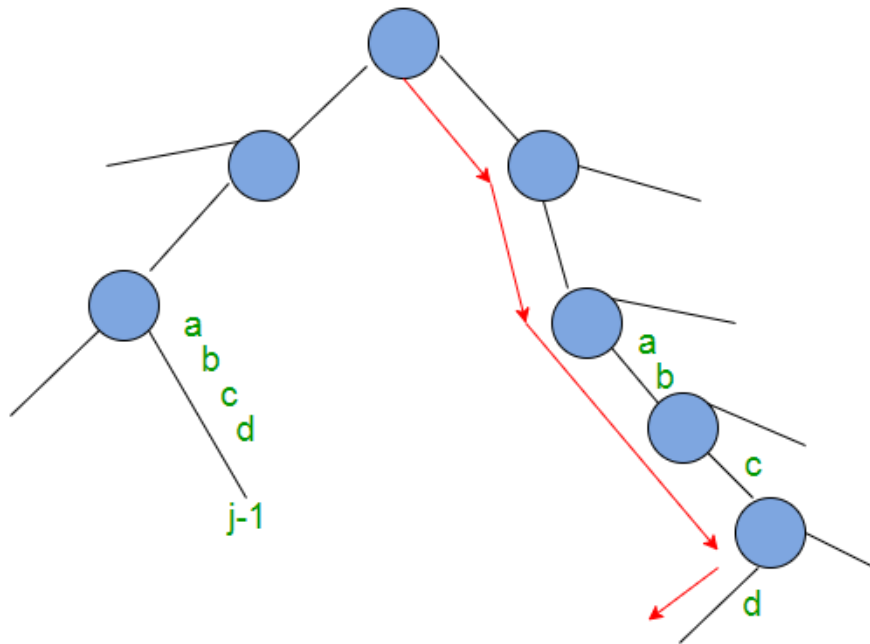


Figure 16 : Traversal from root to leaf in extension  $j$  of phase  $i+1$ , to find end of  $S[j....i]$ , when suffix link is not used.

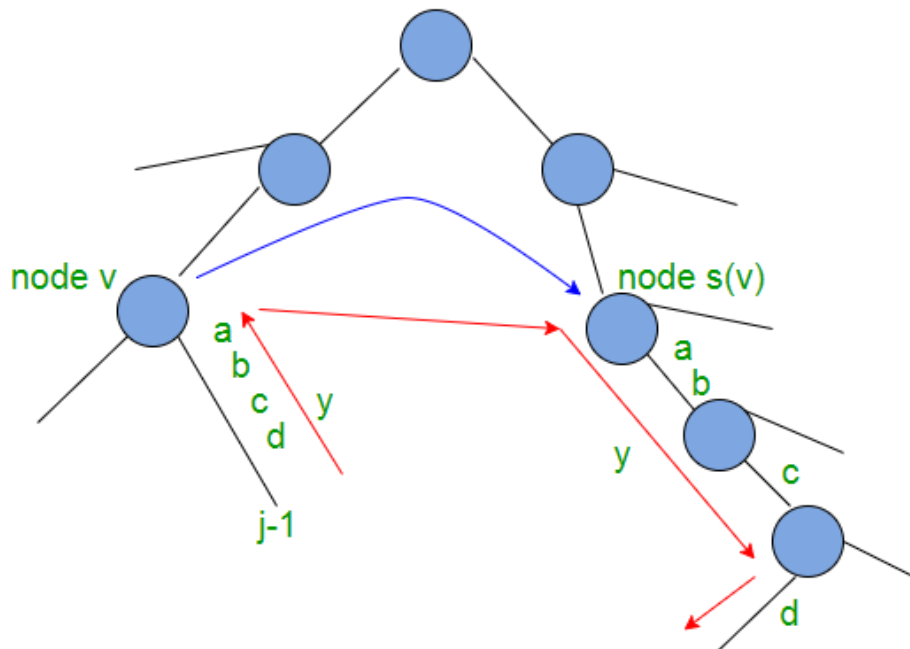


Figure 17 : Traversal from root to leaf in extension  $j$  of phase  $i+1$ , to find end of  $S[j....i]$ , when suffix link (blue arrow) is used.

So we can see that, to find end of path  $S[j..i]$ , we need not traverse from root. We can start from the end of path  $S[j-1..i]$ , walk up one edge to node  $v$  (i.e. go to parent node), follow the suffix link to  $s(v)$ , then walk down the path  $y$  (which is  $abcd$  here in Figure 17).

This shows the use of suffix link is an improvement over the process.

Note: In the next part 3, we will introduce `activePoint` which will help to avoid “walk up”. We can directly go to node  $s(v)$  from node  $v$ .

When there is a suffix link from node  $v$  to node  $s(v)$ , then if there is a path labelled with string  $y$  from node  $v$  to a leaf, then there must be a path labelled with string  $y$  from node  $s(v)$  to a leaf. In Figure 17, there is a path label “ $abcd$ ” from node  $v$  to a leaf, then there is a path will same label “ $abcd$ ” from node  $s(v)$  to a leaf.

This fact can be used to improve the walk from  $s(v)$  to leaf along the path  $y$ . This is called “skip/count” trick.

### Skip/Count Trick

When walking down from node  $s(v)$  to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.

If implementation is such a way that number of characters on any edge, character at a given position in string  $S$  should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

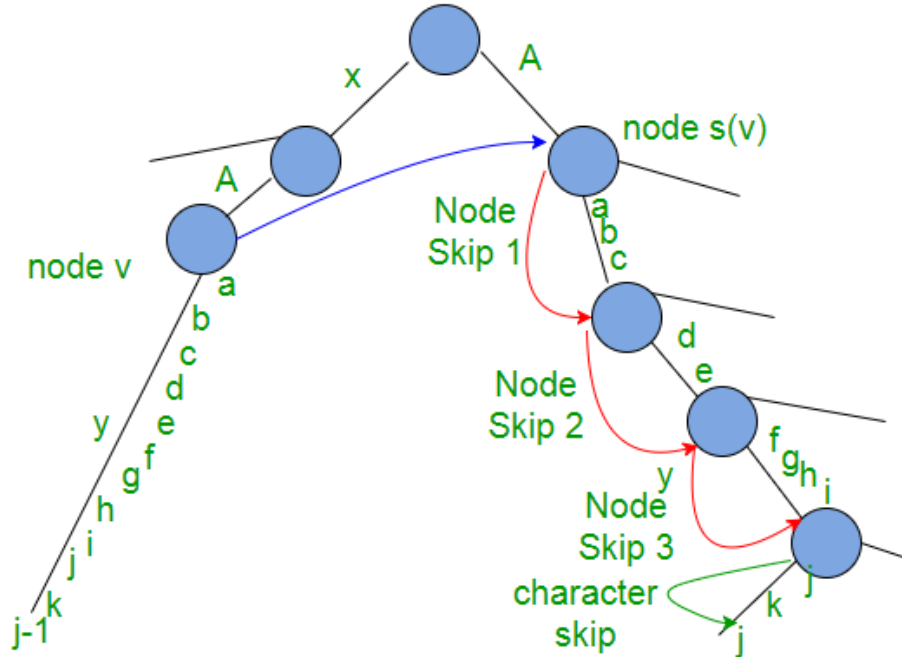


Figure 18 : skip/count trick : substring  $y$  from node  $v$  has length 11. Substring  $y$  from node  $s(v)$  is two characters down the last node, after 3 node skips

Using suffix link along with skip/count trick, suffix tree can be built in  $O(m^2)$  as there are

$m$  phases and each phase takes  $O(m)$ .

### Edge-label compression

So far, path labels are represented as characters in string. Such a suffix tree will take  $O(m^2)$  space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string  $S$ . With this, suffix tree needs  $O(m)$  space.

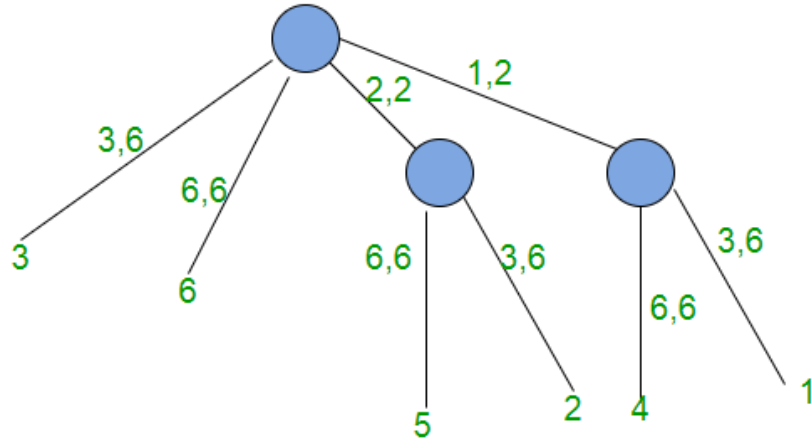


Figure 19 : Suffix tree for string xabxac with edge-label compression  
Figure 14 shows same suffix tree without edge-label compression

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick “skip/count” is seen already while walk down).

### Observation 1: Rule 3 is show stopper

In a phase  $i$ , there are  $i$  extensions (1 to  $i$ ) to be done.

When rule 3 applies in any extension  $j$  of phase  $i+1$  (i.e. path labelled  $S[j..i]$  continues with character  $S[i+1]$ ), then it will also apply in all further extensions of same phase (i.e. extensions  $j+1$  to  $i+1$  in phase  $i+1$ ). That’s because if path labelled  $S[j..i]$  continues with character  $S[i+1]$ , then path labelled  $S[j+1..i]$ ,  $S[j+2..i]$ ,  $S[j+3..i]$ , ...,  $S[i..i]$  will also continue with character  $S[i+1]$ .

Consider Figure 11, Figure12 and Figure 13 in [Part 1](#) where Rule 3 is applied.

In Figure 11, “xab” is added in tree and in Figure 12 (Phase 4), we add next character “x”. In this, 3 extensions are done (which adds 3 suffixes). Last suffix “x” is already present in tree.

In Figure 13, we add character “a” in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes “xa” and “a” are already present in tree. This shows that if suffix  $S[j..i]$  present in tree, then ALL the remaining suffixes  $S[j+1..i]$ ,  $S[j+2..i]$ ,  $S[j+3..i]$ , ...,  $S[i..i]$  will also be there in tree and no work needed to add those remaining suffixes.

So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node  $v$  gets created in extension  $j$  and rule 3 applies in next extension  $j+1$ , then we need to add suffix link from node  $v$  to current node (if we are on internal node) or root node. ActiveNode, which will be discussed in part 3, will help while setting suffix links.

**Trick 2**

Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

**Observation 2: Once a leaf, always a leaf**

Once a leaf is created and labelled  $j$  (for suffix starting at position  $j$  in string  $S$ ), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as  $j$ , extension rule 1 will always apply to extension  $j$  in all successive phases.

Consider Figure 9 to Figure 14 in [Part 1](#).

In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase  $i$ , there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase  $i$  ends.

Also rule 2 creates a new leaf always (and internal node sometimes).

If  $J_i$  represents the last extension in phase  $i$  when rule 1 or 2 was applied (i.e after  $i^{\text{th}}$  phase, there will be  $J_i$  leaves labelled 1, 2, 3, ...,  $J_i$ ), then  $J_i \leq J_{i+1}$

$J_i$  will be equal to  $J_{i+1}$  when there are no new leaf created in phase  $i+1$  (i.e rule 3 is applied in  $J_{i+1}$  extension)

In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here  $J_3 = 3$

In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here  $J_4 = 3 = J_3$

In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here  $J_5 = 3 = J_4$

$J_i$  will be less than  $J_{i+1}$  when few new leaves are created in phase  $i+1$ .

In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here  $J_6 = 6 > J_5$

So we can see that in phase  $i+1$ , only rule 1 will apply in extensions 1 to  $J_i$  (which really doesn't need much work, can be done in constant time and that's the trick 3), extension  $J_{i+1}$  onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.

Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase  $i$ , end =  $i$  for leaf edges, for phase  $i+1$ , end =  $i+1$  for leaf edges.

**Trick 3**

In any phase  $i$ , leaf edges may look like  $(p, i), (q, i), (r, i), \dots$  where  $p, q, r$  are starting position of different edges and  $i$  is end position of all. Then in phase  $i+1$ , these leaf edges will look like  $(p, i+1), (q, i+1), (r, i+1), \dots$ . This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index  $e$  and  $e$  will be equal to phase number. So now leaf edges will look like  $(p, e), (q, e), (r, e), \dots$ . In any phase, just increment  $e$  and extension on all leaf edges will be done. Figure 19 shows this.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.

Tree  $T_m$  could be implicit tree if a suffix is prefix of another. So we can add a \$ terminal symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index  $e$ ), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next [Part 3](#), we will take string  $S = \text{"abcbxabcd"}$  as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by ActivePoints. We will continue to discuss the algorithm in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

#### References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-2/>

## Chapter 56

# Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction - Part 1 - GeeksforGeeks

Suffix Tree is very useful in numerous string processing and computational biology problems. Many books and e-resources talk about it theoretically and in few places, code implementation is discussed. But still, I felt something is missing and it's not easy to implement code to construct suffix tree and it's usage in many applications. This is an attempt to bridge the gap between theory and complete working code implementation. Here we will discuss Ukkonen's Suffix Tree Construction Algorithm. We will discuss it in step by step detailed way and in multiple parts from theory to implementation. We will start with brute force way and try to understand different concepts, tricks involved in Ukkonen's algorithm and in the last part, code implementation will be discussed.

**Note:** You may find some portion of the algorithm difficult to understand while 1<sup>st</sup> or 2<sup>nd</sup> reading and it's perfectly fine. With few more attempts and thought, you should be able to understand such portions.

Book [Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology](#) by **Dan Gusfield** explains the concepts very well.

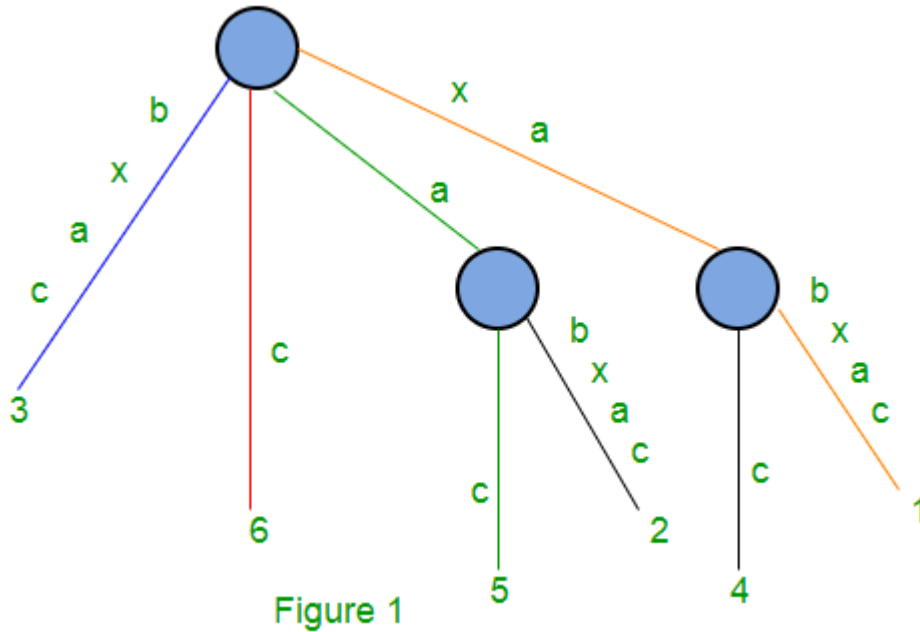
A suffix tree **T** for a m-character string **S** is a rooted directed tree with exactly m leaves numbered 1 to **m**. (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of **S**.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf **i** gives the suffix of **S** that starts at position **i**, i.e. **S**[**i**...**m**].

**Note:** Position starts with 1 (it's not zero indexed, but later, while code implementation, we will use zero indexed position)

For string  $S = \text{xabxac}$  with  $m = 6$ , suffix tree will look like following:



It has one root node and two internal nodes and 6 leaf nodes.

String Depth of red path is 1 and it represents suffix  $c$  starting at position 6

String Depth of blue path is 4 and it represents suffix  $\text{bxca}$  starting at position 3

String Depth of green path is 2 and it represents suffix  $\text{ac}$  starting at position 5

String Depth of orange path is 6 and it represents suffix  $\text{xabxac}$  starting at position 1

Edges with labels  $a$  (green) and  $xa$  (orange) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)

If one suffix of  $S$  matches a prefix of another suffix of  $S$  (when last character is not unique in string), then path for the first suffix would not end at a leaf.

For String  $S = \text{xabxa}$ , with  $m = 5$ , following is the suffix tree:

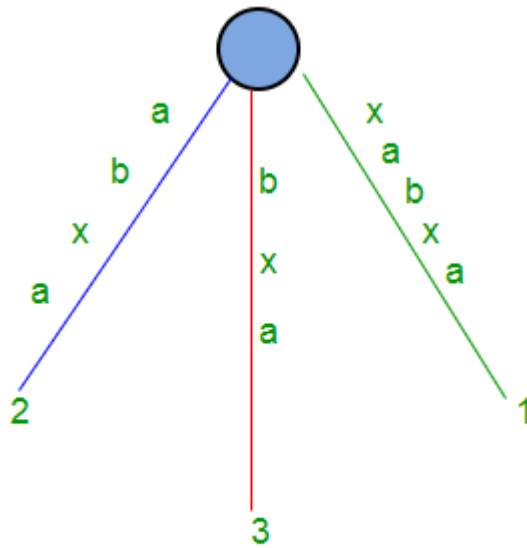


Figure 2

Here we will have 5 suffixes: xabxa, abxa, bxa, xa and a.

Path for suffixes 'xa' and 'a' do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes ('xa' and 'a') are not seen explicitly in tree.

To avoid this problem, we add a character which is not present in string already. We normally use \$, # etc as termination characters.

Following is the suffix tree for string  $S = \text{xabxa\$}$  with  $m = 6$  and now all 6 suffixes end at leaf.



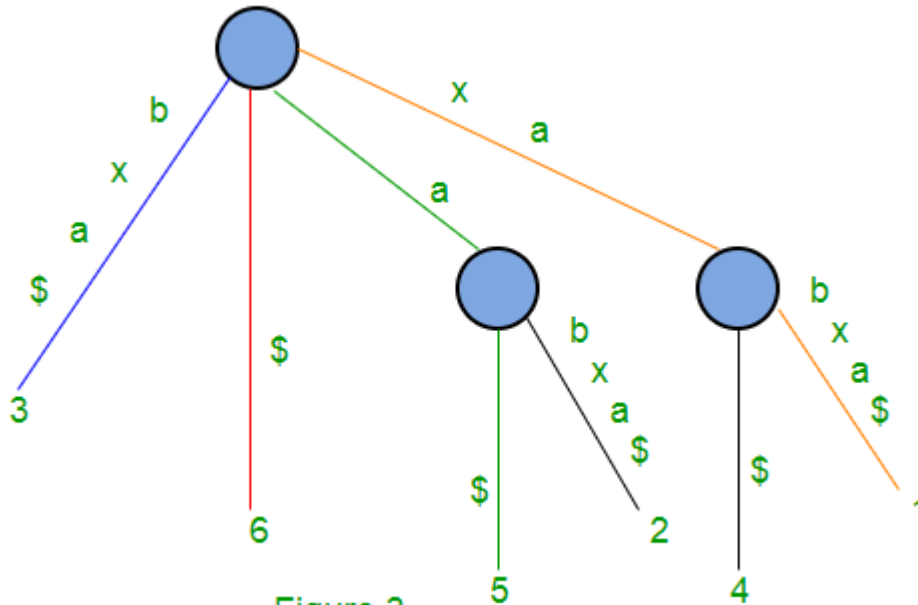


Figure 3

**A naive algorithm to build a suffix tree**

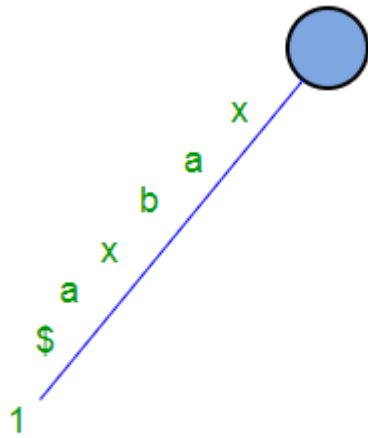
Given a string  $S$  of length  $m$ , enter a single edge for suffix  $S[1..m]\$$  (the entire string) into the tree, then successively enter suffix  $S[i..m]\$$  into the growing tree, for  $i$  increasing from 2 to  $m$ . Let  $N_i$  denote the intermediate tree that encodes all the suffixes from 1 to  $i$ .

So  $N_{i+1}$  is constructed from  $N_i$  as follows:

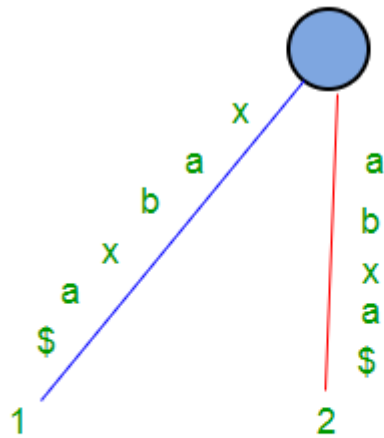
- Start at the root of  $N_i$
- Find the longest path from the root which matches a prefix of  $S[i+1..m]\$$
- Match ends either at the node (say  $w$ ) or in the middle of an edge [say  $(u, v)$ ].
- If it is in the middle of an edge  $(u, v)$ , break the edge  $(u, v)$  into two edges by inserting a new node  $w$  just after the last character on the edge that matched a character in  $S[i+1..m]$  and just before the first character on the edge that mismatched. The new edge  $(u, w)$  is labelled with the part of the  $(u, v)$  label that matched with  $S[i+1..m]$ , and the new edge  $(w, v)$  is labelled with the remaining part of the  $(u, v)$  label.
- Create a new edge  $(w, i+1)$  from  $w$  to a new leaf labelled  $i+1$  and it labels the new edge with the unmatched part of suffix  $S[i+1..m]$

This takes  $O(m^2)$  to build the suffix tree for the string  $S$  of length  $m$ .

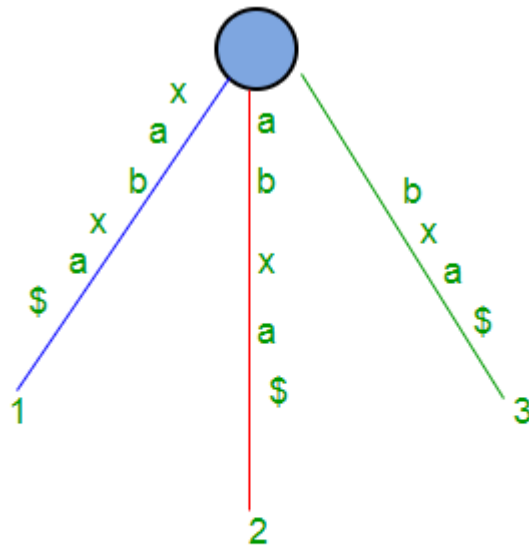
Following are few steps to build suffix tree based for string “xabxa\$” based on above algorithm:



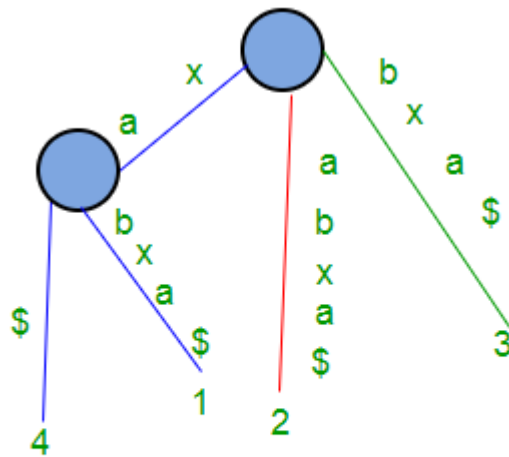
Tree with suffix N1, S[1....6]  
Figure 4



Tree with suffix N1, S[1....6] and N2, S[2...6]  
Figure 5



Tree with suffixes N1,N2 and N3  
Figure 6



Tree with suffix N1, N2, N3 and N4  
Figure 7

### Implicit suffix tree

While generating suffix tree using Ukkonen's algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with \$ (or # or any other termination character) label and no internal

node with only one edge going out of it.

To get implicit suffix tree from a suffix tree  $S\$$ ,

- Remove all terminal symbol  $\$$  from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.

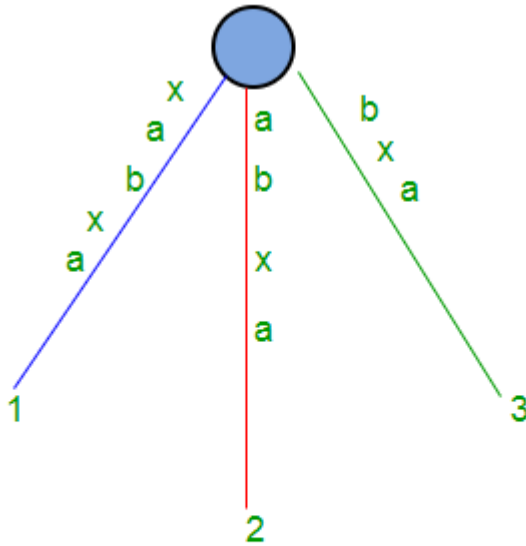


Figure 8 : Implicit suffix tree for storing xabxa  
Suffix tree shown in Figure 3

### High Level Description of Ukkonen's algorithm

Ukkonen's algorithm constructs an implicit suffix tree  $T_i$  for each prefix  $S[1..i]$  of  $S$  (of length  $m$ ).

It first builds  $T_1$  using 1<sup>st</sup> character, then  $T_2$  using 2<sup>nd</sup> character, then  $T_3$  using 3<sup>rd</sup> character, ...,  $T_m$  using  $m^{\text{th}}$  character.

Implicit suffix tree  $T_{i+1}$  is built on top of implicit suffix tree  $T_i$ .

The true suffix tree for  $S$  is built from  $T_m$  by adding  $\$$ .

At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.

Time taken is  $O(m)$ .

Ukkonen's algorithm is divided into  $m$  phases (one phase for each character in the string with length  $m$ )

In phase  $i+1$ , tree  $T_{i+1}$  is built from tree  $T_i$ .

Each phase  $i+1$  is further divided into  $i+1$  extensions, one for each of the  $i+1$  suffixes of  $S[1..i+1]$

In extension  $j$  of phase  $i+1$ , the algorithm first finds the end of the path from the root

labelled with substring  $S[j..i]$ .

It then extends the substring by adding the character  $S[i+1]$  to its end (if it is not there already).

In extension 1 of phase  $i+1$ , we put string  $S[1..i+1]$  in the tree. Here  $S[1..i]$  will already be present in tree due to previous phase  $i$ . We just need to add  $S[i+1]$ th character in tree (if not there already).

In extension 2 of phase  $i+1$ , we put string  $S[2..i+1]$  in the tree. Here  $S[2..i]$  will already be present in tree due to previous phase  $i$ . We just need to add  $S[i+1]$ th character in tree (if not there already).

In extension 3 of phase  $i+1$ , we put string  $S[3..i+1]$  in the tree. Here  $S[3..i]$  will already be present in tree due to previous phase  $i$ . We just need to add  $S[i+1]$ th character in tree (if not there already).

.

In extension  $i+1$  of phase  $i+1$ , we put string  $S[i+1..i+1]$  in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label  $S[i+1]$ .

### High Level Ukkonen's algorithm

Construct tree  $T_1$

For  $i$  from 1 to  $m-1$  do

begin {phase  $i+1$ }

For  $j$  from 1 to  $i+1$

begin {extension  $j$ }

Find the end of the path from the root labelled  $S[j..i]$  in the current tree.

Extend that path by adding character  $S[i+1]$  if it is not there already

end;

end;

Suffix extension is all about adding the next character into the suffix tree built so far.

In extension  $j$  of phase  $i+1$ , algorithm finds the end of  $S[j..i]$  (which is already in the tree due to previous phase  $i$ ) and then it extends  $S[j..i]$  to be sure the suffix  $S[j..i+1]$  is in the tree.

There are 3 extension rules:

**Rule 1:** If the path from the root labelled  $S[j..i]$  ends at leaf edge (i.e.  $S[i]$  is last character on leaf edge) then character  $S[i+1]$  is just added to the end of the label on that leaf edge.

**Rule 2:** If the path from the root labelled  $S[j..i]$  ends at non-leaf edge (i.e. there are more characters after  $S[i]$  on path) and next character is not  $s[i+1]$ , then a new leaf edge with label  $s[i+1]$  and number  $j$  is created starting from character  $S[i+1]$ .

A new internal node will also be created if  $s[1..i]$  ends inside (in-between) a non-leaf edge.

**Rule 3:** If the path from the root labelled  $S[j..i]$  ends at non-leaf edge (i.e. there are more characters after  $S[i]$  on path) and next character is  $s[i+1]$  (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string *xabxac* using Ukkonen's algorithm:

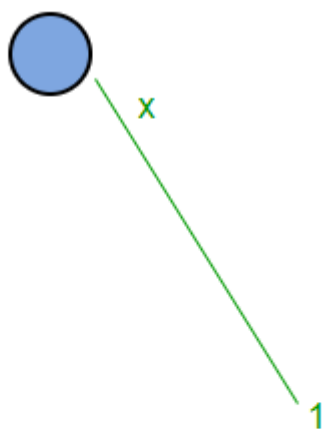


Figure 9 : T1 for S[1...1]  
Adding suffixes of x(x)  
Rule2-A new leaf node

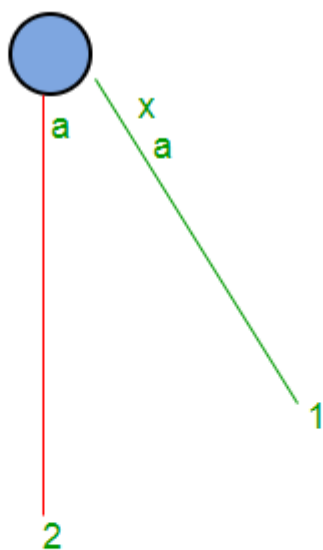


Figure 10 : T2 for S[1...2]  
Adding suffixes of xa(xa and a)  
Rule1-Extending path label in existing leaf edge  
Rule2-A new leaf node

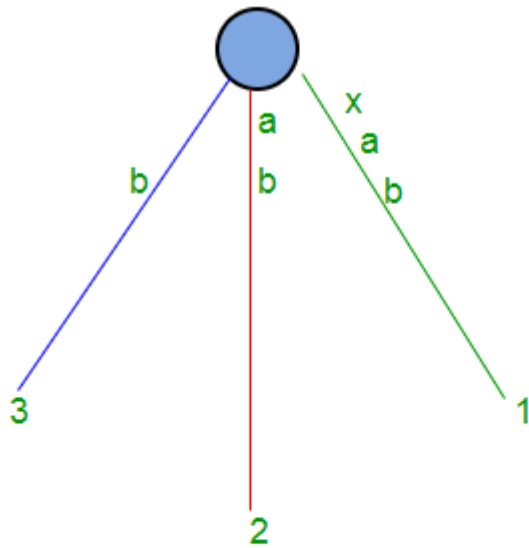


Figure 11 : T3 for S[1...3]

Adding suffixes of xab(xab,ab and b)

Rule1-Extending path label in existing leaf edge

Rule2-A new leaf node

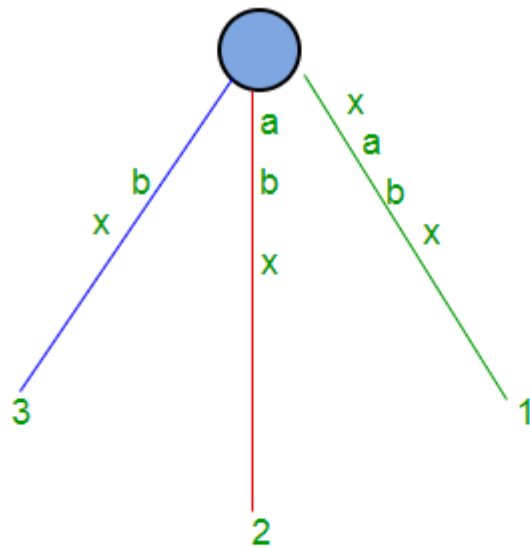


Figure 12 : T4 for S[1...4]

Adding suffixes of xabx(xabx,abx,bx and x)

Rule1-Extending path label in existing leaf edge

Rule3-Do nothing(path with label x already present)



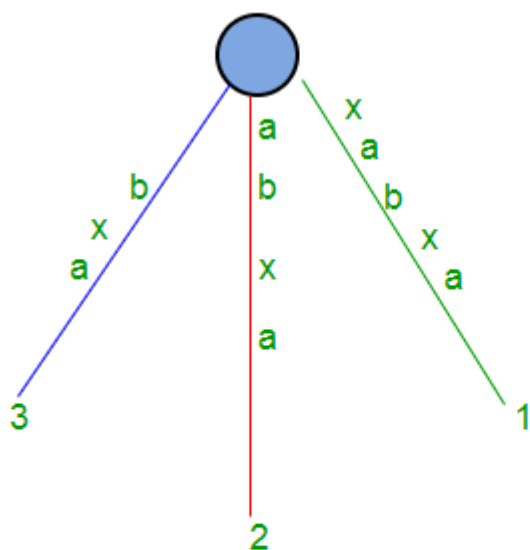


Figure 13 : T5 for S[1...5]

Adding suffixes of xabxa(xabxa,abxa,bxa,xa and x)

Rule1-Extending path label in existing leaf edge

Rule3-Do nothing(path with label xa and a already present)

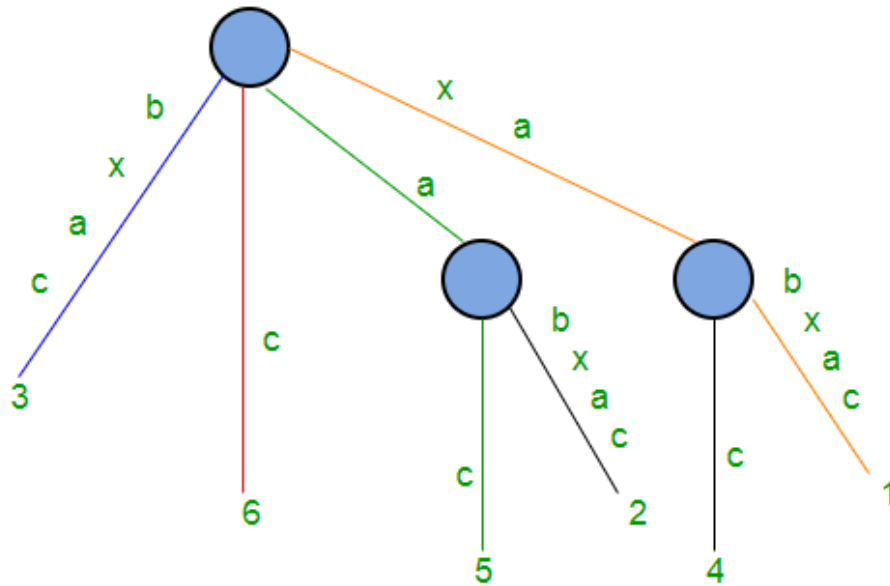


Figure 14 : T6 for S[1...6]

Adding suffixes of xabxac(xabxac, abxac, bxac, xac, ac, c)

Rule1-Extending path label in Existing leaf edge.

Rule2-Three new leaf edges and two new internal nodes

In next parts (Part 2, Part 3, Part 4 and Part 5), we will discuss suffix links, active points, few tricks and finally code implementations (Part 6).

#### References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/>

## Chapter 57

# Pattern Searching using a Trie of all Suffixes

Pattern Searching using a Trie of all Suffixes - GeeksforGeeks

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](#), we discussed that there are two ways efficiently solve the above problem.

1) Preprocess Pattern: [KMP Algorithm](#), [Rabin Karp Algorithm](#), [Finite Automata](#), [Boyer Moore Algorithm](#).

2) Preprocess Text: [Suffix Tree](#)

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 \(A compressed Trie of all suffixes of Text\)](#).

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](#) of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it's a [simple Trie](#) instead of compressed Trie.

As discussed in [Suffix Tree](#) 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.



```
#include<list>
#define MAX_CHAR 256
using namespace std;

// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTrieNode
{
private:
    SuffixTrieNode *children[MAX_CHAR];
    list<int> *indexes;
public:
    SuffixTrieNode() // Constructor
    {
        // Create an empty linked list for indexes of
        // suffixes starting from this node
        indexes = new list<int>;

        // Initialize all child pointers as NULL
        for (int i = 0; i < MAX_CHAR; i++)
            children[i] = NULL;
    }

    // A recursive function to insert a suffix of the txt
    // in subtree rooted with this node
    void insertSuffix(string suffix, int index);

    // A function to search a pattern in subtree rooted
    // with this node. The function returns pointer to a linked
    // list containing all indexes where pattern is present.
    // The returned indexes are indexes of last characters
    // of matched text.
    list<int>* search(string pat);
};

// A Trie of all suffixes
class SuffixTrie
{
private:
    SuffixTrieNode root;
public:
    // Constructor (Builds a trie of suffies of the given text)
    SuffixTrie(string txt)
    {
        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
        // insertSuffix() in SuffixTrieNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substr(i), i);
    }
};
```

```
    }

    // Function to searches a pattern in this suffix trie.
    void search(string pat);
};

// A recursive function to insert a suffix of the txt in
// subtree rooted with this node
void SuffixTrieNode::insertSuffix(string s, int index)
{
    // Store index in linked list
    indexes->push_back(index);

    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character
        char cIndex = s.at(0);

        // If there is no edge for this character, add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTrieNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1), index+1);
    }
}

// A recursive function to search a pattern in subtree rooted with
// this node
list<int>* SuffixTrieNode::search(string s)
{
    // If all characters of pattern have been processed,
    if (s.length() == 0)
        return indexes;

    // if there is an edge from the current node of suffix trie,
    // follow the edge.
    if (children[s.at(0)] != NULL)
        return (children[s.at(0)]->search(s.substr(1)));

    // If there is no edge, pattern doesn't exist in text
    else return NULL;
}

/* Prints all occurrences of pat in the Suffix Trie S (built for text)*/
void SuffixTrie::search(string pat)
{

```

```
// Let us call recursive search function for root of Trie.
// We get a list of all indexes (where pat is present in text) in
// variable 'result'
list<int> *result = root.search(pat);

// Check if the list of indexes is empty or not
if (result == NULL)
    cout << "Pattern not found" << endl;
else
{
    list<int>::iterator i;
    int patLen = pat.length();
    for (i = result->begin(); i != result->end(); ++i)
        cout << "Pattern found at position " << *i - patLen << endl;
}

// driver program to test above functions
int main()
{
    // Let us build a suffix trie for text "geeksforgeeks.org"
    string txt = "geeksforgeeks.org";
    SuffixTrie S(txt);

    cout << "Search for 'ee'" << endl;
    S.search("ee");

    cout << "\nSearch for 'geek'" << endl;
    S.search("geek");

    cout << "\nSearch for 'quiz'" << endl;
    S.search("quiz");

    cout << "\nSearch for 'forgeeks'" << endl;
    S.search("forgeeks");

    return 0;
}
```

## Java

```
import java.util.LinkedList;
import java.util.List;
class SuffixTrieNode {

    final static int MAX_CHAR = 256;

    SuffixTrieNode[] children = new SuffixTrieNode[MAX_CHAR];
```

```
List<Integer> indexes;

SuffixTrieNode() // Constructor
{
    // Create an empty linked list for indexes of
    // suffixes starting from this node
    indexes = new LinkedList<Integer>();

    // Initialize all child pointers as NULL
    for (int i = 0; i < MAX_CHAR; i++)
        children[i] = null;
}

// A recursive function to insert a suffix of
// the text in subtree rooted with this node
void insertSuffix(String s, int index) {

    // Store index in linked list
    indexes.add(index);

    // If string has more characters
    if (s.length() > 0) {

        // Find the first character
        char cIndex = s.charAt(0);

        // If there is no edge for this character,
        // add a new edge
        if (children[cIndex] == null)
            children[cIndex] = new SuffixTrieNode();

        // Recur for next suffix
        children[cIndex].insertSuffix(s.substring(1),
                                      index + 1);
    }
}

// A function to search a pattern in subtree rooted
// with this node. The function returns pointer to a
// linked list containing all indexes where pattern
// is present. The returned indexes are indexes of
// last characters of matched text.
List<Integer> search(String s) {

    // If all characters of pattern have been
    // processed,
    if (s.length() == 0)
        return indexes;
}
```



```
// if there is an edge from the current node of
// suffix tree, follow the edge.
if (children[s.charAt(0)] != null)
    return (children[s.charAt(0)].search(s.substring(1)));

// If there is no edge, pattern doesnt exist in
// text
else
    return null;
}
}

// A Trie of all suffixes
class Suffix_tree{

    SuffixTrieNode root = new SuffixTrieNode();

    // Constructor (Builds a trie of suffies of the
    // given text)
    Suffix_tree(String txt) {

        // Consider all suffixes of given string and
        // insert them into the Suffix Trie using
        // recursive function insertSuffix() in
        // SuffixTrieNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substring(i), i);
    }

    /* Prints all occurrences of pat in the Suffix Trie S
    (built for text) */
    void search_tree(String pat) {

        // Let us call recursive search function for
        // root of Trie.
        // We get a list of all indexes (where pat is
        // present in text) in variable 'result'
        List<Integer> result = root.search(pat);

        // Check if the list of indexes is empty or not
        if (result == null)
            System.out.println("Pattern not found");
        else {

            int patLen = pat.length();

            for (Integer i : result)
```

```
        System.out.println("Pattern found at position " +
                           (i - patLen));
    }
}

// driver program to test above functions
public static void main(String args[]) {

    // Let us build a suffix trie for text
    // "geeksforgeeks.org"
    String txt = "geeksforgeeks.org";
    Suffix_tree S = new Suffix_tree(txt);

    System.out.println("Search for 'ee'");
    S.search_tree("ee");

    System.out.println("\nSearch for 'geek'");
    S.search_tree("geek");

    System.out.println("\nSearch for 'quiz'");
    S.search_tree("quiz");

    System.out.println("\nSearch for 'forgeeks'");
    S.search_tree("forgeeks");
}
}
```

// This code is contributed by Sumit Ghosh

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.

This article is contributed by Ashish Anand. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Improved By :** [smodi2007](#)

## Source

<https://www.geeksforgeeks.org/pattern-searching-using-trie-suffixes/>

## Chapter 58

# Anagram Substring Search (Or Search for all permutations)

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

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[] = "BACDGABCD"  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
```

This problem is slightly different from 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.

A simple idea is to modify [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.
  - ....a) If the two count arrays are identical, we found an occurrence.
  - ....b) Increment count of current character of text in `countTW[]`
  - ....c) Decrement count of first character in previous window in `countTW[]`
- 3) The last window is not checked by above loop, so explicitly check it.

Following is the implementation of above algorithm.

C++

```
// C++ program to search all anagrams of a pattern in a text
#include<iostream>
#include<cstring>
#define MAX 256
using namespace std;

// This function returns true if contents of arr1[] and arr2[]
// are same, otherwise false.
bool compare(char arr1[], char arr2[])
{
    for (int i=0; i<MAX; i++)
        if (arr1[i] != arr2[i])
            return false;
    return true;
}

// This function search for all permutations of pat[] in txt[]
void search(char *pat, char *txt)
{
    int M = strlen(pat), N = strlen(txt);

    // countP[]: Store count of all characters of pattern
    // countTW[]: Store count of current window of text
    char countP[MAX] = {0}, countTW[MAX] = {0};
    for (int i = 0; i < M; i++)
    {
```

```
        (countP[pat[i]])++;
        (countTW[txt[i]])++;
    }

    // Traverse through remaining characters of pattern
    for (int i = M; i < N; i++)
    {
        // Compare counts of current window of text with
        // counts of pattern[]
        if (compare(countP, countTW))
            cout << "Found at Index " << (i - M) << endl;

        // Add current character to current window
        (countTW[txt[i]])++;

        // Remove the first character of previous window
        countTW[txt[i-M]]--;
    }

    // Check for the last window in text
    if (compare(countP, countTW))
        cout << "Found at Index " << (N - M) << endl;
}

/* Driver program to test above function */
int main()
{
    char txt[] = "BACDGABCD";
    char pat[] = "ABCD";
    search(pat, txt);
    return 0;
}
```

## Java

```
// Java program to search all anagrams
// of a pattern in a text
public class GFG
{
    static final int MAX = 256;

    // This function returns true if contents
    // of arr1[] and arr2[] are same, otherwise
    // false.
    static boolean compare(char arr1[], char arr2[])
    {
        for (int i = 0; i < MAX; i++)
            if (arr1[i] != arr2[i])
                return false;
        return true;
    }
}
```

```
        return false;
    return true;
}

// This function search for all permutations
// of pat[] in txt[]
static void search(String pat, String txt)
{
    int M = pat.length();
    int N = txt.length();

    // countP[]: Store count of all
    // characters of pattern
    // countTW[]: Store count of current
    // window of text
    char[] countP = new char[MAX];
    char[] countTW = new char[MAX];
    for (int i = 0; i < M; i++)
    {
        (countP[pat.charAt(i)])++;
        (countTW[txt.charAt(i)])++;
    }

    // Traverse through remaining characters
    // of pattern
    for (int i = M; i < N; i++)
    {
        // Compare counts of current window
        // of text with counts of pattern[]
        if (compare(countP, countTW))
            System.out.println("Found at Index " +
                               (i - M));

        // Add current character to current
        // window
        (countTW[txt.charAt(i)])++;

        // Remove the first character of previous
        // window
        countTW[txt.charAt(i-M)]--;
    }

    // Check for the last window in text
    if (compare(countP, countTW))
        System.out.println("Found at Index " +
                           (N - M));
}
```

```
/* Driver program to test above function */
public static void main(String args[])
{
    String txt = "BACDGABCD";
    String pat = "ABCD";
    search(pat, txt);
}
// This code is contributed by Sumit Ghosh
```

### Python3

```
# 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 = []

    for i in range(MAX):
        countP.append(0)

    countTW = []

    for i in range(MAX):
        countTW.append(0)

    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 = "BACDGABCD"
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
```

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/anagram-substring-search-search-permutations/>

## Chapter 59

# Suffix Array | Set 2 (nLogn Algorithm)

Suffix Array | Set 2 (nLogn Algorithm) - GeeksforGeeks

A **suffix array** is a sorted array of all suffixes of a given string. The definition is similar to [Suffix Tree](#) which is compressed trie of all suffixes of the given text.

Let the given string be "banana".

|          |                   |          |
|----------|-------------------|----------|
| 0 banana |                   | 5 a      |
| 1 anana  | Sort the Suffixes | 3 ana    |
| 2 nana   | ----->            | 1 anana  |
| 3 ana    | alphabetically    | 0 banana |
| 4 na     |                   | 4 na     |
| 5 a      |                   | 2 nana   |

The suffix array for "banana" is {5, 3, 1, 0, 4, 2}

We have discussed [Naive algorithm](#) for construction of suffix array. The Naive algorithm is to consider all suffixes, sort them using a  $O(n \log n)$  sorting algorithm and while sorting, maintain original indexes. Time complexity of the Naive algorithm is  $O(n^2 \log n)$  where  $n$  is the number of characters in the input string.

In this post, a  **$O(n \log n)$  algorithm** for suffix array construction is discussed. Let us first discuss a  $O(n * \log n * \log n)$  algorithm for simplicity. The idea is to use the fact that strings that are to be sorted are suffixes of a single string.

We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than  $2n$ . The important point is, if we have sorted suffixes according to first  $2^i$  characters, then we can sort suffixes according to first  $2^{i+1}$  characters in  $O(n \log n)$  time using a  $n \log n$  sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in  $O(1)$

time (we need to compare only two values, see the below example and code). The sort function is called  $O(\log n)$  times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes  $O(n \log n \log n)$ . See <http://www.stanford.edu/class/cs97si/suffix-array.pdf> for more details.

Let us build suffix array the example string “banana” using above algorithm.

**Sort according to first two characters** Assign a rank to all suffixes using ASCII value of first character. A simple way to assign rank is to do “str[i] - ‘a’” for  $i$ th suffix of strp[]

| Index | Suffix | Rank |
|-------|--------|------|
| 0     | banana | 1    |
| 1     | anana  | 0    |
| 2     | nana   | 13   |
| 3     | ana    | 0    |
| 4     | na     | 13   |
| 5     | a      | 0    |

For every character, we also store rank of next adjacent character, i.e., the rank of character at str[i + 1] (This is needed to sort the suffixes according to first 2 characters). If a character is last character, we store next rank as -1

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 0     | banana | 1    | 0         |
| 1     | anana  | 0    | 13        |
| 2     | nana   | 13   | 0         |
| 3     | ana    | 0    | 13        |
| 4     | na     | 13   | 0         |
| 5     | a      | 0    | -1        |

Sort all Suffixes according to rank and adjacent rank. Rank is considered as first digit or MSD, and adjacent rank is considered as second digit.

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5     | a      | 0    | -1        |
| 1     | anana  | 0    | 13        |
| 3     | ana    | 0    | 13        |
| 0     | banana | 1    | 0         |
| 2     | nana   | 13   | 0         |
| 4     | na     | 13   | 0         |

#### Sort according to first four character

Assign new ranks to all suffixes. To assign new ranks, we consider the sorted suffixes one by one. Assign 0 as new rank to first suffix. For assigning ranks to remaining suffixes, we

consider rank pair of suffix just before the current suffix. If previous rank pair of a suffix is same as previous rank of suffix just before it, then assign it same rank. Otherwise assign rank of previous suffix plus one.

| Index | Suffix | Rank |                                    |
|-------|--------|------|------------------------------------|
| 5     | a      | 0    | [Assign 0 to first]                |
| 1     | anana  | 1    | (0, 13) is different from previous |
| 3     | ana    | 1    | (0, 13) is same as previous        |
| 0     | banana | 2    | (1, 0) is different from previous  |
| 2     | nana   | 3    | (13, 0) is different from previous |
| 4     | na     | 3    | (13, 0) is same as previous        |

For every suffix `str[i]`, also store rank of next suffix at `str[i + 2]`. If there is no next suffix at `i + 2`, we store next rank as -1

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5     | a      | 0    | -1        |
| 1     | anana  | 1    | 1         |
| 3     | ana    | 1    | 0         |
| 0     | banana | 2    | 3         |
| 2     | nana   | 3    | 3         |
| 4     | na     | 3    | -1        |

Sort all Suffixes according to rank and next rank.

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5     | a      | 0    | -1        |
| 3     | ana    | 1    | 0         |
| 1     | anana  | 1    | 1         |
| 0     | banana | 2    | 3         |
| 4     | na     | 3    | -1        |
| 2     | nana   | 3    | 3         |

```
// C++ program for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
```

```

    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
        (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according to first
    // 2 characters. Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n]; // This array is needed to get the index in suffixes[]
                // from original index. This mapping is needed to get
                // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)

```

```

    {
        // If first rank and next ranks are same as that of previous
        // suffix in array, assign the same new rank to this suffix
        if (suffixes[i].rank[0] == prev_rank &&
            suffixes[i].rank[1] == suffixes[i-1].rank[1])
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = rank;
        }
        else // Otherwise increment rank and assign
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = ++rank;
        }
        ind[suffixes[i].index] = i;
    }

    // Assign next rank to every suffix
    for (int i = 0; i < n; i++)
    {
        int nextindex = suffixes[i].index + k/2;
        suffixes[i].rank[1] = (nextindex < n)?
            suffixes[ind[nextindex]].rank[0] : -1;
    }

    // Sort the suffixes according to first k characters
    sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
int *suffixArr = new int[n];
for (int i = 0; i < n; i++)
    suffixArr[i] = suffixes[i].index;

// Return the suffix array
return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()

```

```
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt, n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
Following is suffix array for banana
5 3 1 0 4 2
```

Note that the above algorithm uses standard sort function and therefore time complexity is  $O(n \log n \log n)$ . We can use [Radix Sort](#) here to reduce the time complexity to  $O(n \log n)$ .

Please note that suffix arrays can be constructed in  $O(n)$  time also. We will soon be discussing  $O(n)$  algorithms.

**References:**

<http://www.stanford.edu/class/cs97si/suffix-array.pdf>

<http://www.cbc.umd.edu/confcour/Fall2012/lec14b.pdf>

**Improved By :** [Akash Kumar](#) 31

**Source**

<https://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/>

## Chapter 60

# Suffix Array | Set 1 (Introduction)

Suffix Array | Set 1 (Introduction) - GeeksforGeeks

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

[Pattern Searching | Set 8 \(Suffix Tree Introduction\)](#)

**A suffix array is a sorted array of all suffixes of a given string.** The definition is similar to [Suffix Tree which is compressed trie of all suffixes of the given text](#). Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source [Wiki](#)).

A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.

Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: [Wiki](#))

### **Example:**

Let the given string be "banana".

|          |                   |          |
|----------|-------------------|----------|
| 0 banana |                   | 5 a      |
| 1 anana  | Sort the Suffixes | 3 ana    |
| 2 nana   | ----->            | 1 anana  |
| 3 ana    | alphabetically    | 0 banana |
| 4 na     |                   | 4 na     |
| 5 a      |                   | 2 nana   |

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}



**Naive method to build Suffix Array**

A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```
// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;
}
```

```
// Return the suffix array
return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt, n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
Following is suffix array for banana
5 3 1 0 4 2
```

The time complexity of above method to build suffix array is  $O(n^2 \log n)$  if we consider a  $O(n \log n)$  algorithm used for sorting. The sorting step itself takes  $O(n^2 \log n)$  time as every comparison is a comparison of two strings and the comparison takes  $O(n)$  time. There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

### ***Search a pattern using the built Suffix Array***

To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, [Binary Search](#) can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only report one of them.

```
// This code only contains search() and main. To make it a complete running
// above code or see https://ide.geeksforgeeks.org/oY70kD

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat); // get length of pattern, needed for strcmp()
```

```
// Do simple binary search for the pat in txt using the
// built suffix array
int l = 0, r = n-1; // Initilize left and right indexes
while (l <= r)
{
    // See if 'pat' is prefix of middle suffix in suffix array
    int mid = l + (r - l)/2;
    int res = strncmp(pat, txt+suffArr[mid], m);

    // If match found at the middle, print it and return
    if (res == 0)
    {
        cout << "Pattern found at index " << suffArr[mid];
        return;
    }

    // Move to left half if pattern is alphabtically less than
    // the mid suffix
    if (res < 0) r = mid - 1;

    // Otherwise move to right half
    else l = mid + 1;
}

// We reach here if return statement in loop is not executed
cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana"; // text
    char pat[] = "nan";    // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);

    // search pat in txt using the built suffix array
    search(pat, txt, suffArr, n);

    return 0;
}
```

Output:

Pattern found at index 2

The time complexity of the above search function is  $O(m \log n)$ . There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a  $O(m)$  suffix array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

### ***Applications of Suffix Array***

Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.

- 1) Pattern Searching
- 2) [Finding the longest repeated substring](#)
- 3) [Finding the longest common substring](#)
- 4) [Finding the longest palindrome in a string](#)

See [this](#) for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed [a  \$O\(n \log n\)\$  algorithm for Suffix Array construction here](#). We will soon be discussing more efficient suffix array algorithms.

### **References:**

<http://www.stanford.edu/class/cs97si/suffix-array.pdf>  
[http://en.wikipedia.org/wiki/Suffix\\_array](http://en.wikipedia.org/wiki/Suffix_array)

### **Source**

<https://www.geeksforgeeks.org/suffix-array-set-1-introduction/>

## Chapter 61

# String matching where one string contains wildcard characters

String matching where one string contains wildcard characters - GeeksforGeeks

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.

C++

```
// A C program to match wild card characters
#include <stdio.h>
#include <stdbool.h>

// The main function that checks if two given strings
// match. The first string may contain wildcard characters
bool match(char *first, char * second)
{
    // If we reach at the end of both strings, we are done
    if (*first == '\0' && *second == '\0')
        return true;
```

```
// Make sure that the characters after '*' are present
// in second string. This function assumes that the first
// string will not contain two consecutive '*'
if (*first == '*' && *(first+1) != '\0' && *second == '\0')
    return false;

// If the first string contains '?', or current characters
// of both strings match
if (*first == '?' || *first == *second)
    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 (*first == '*')
    return match(first+1, second) || match(first, second+1);
return false;
}

// A function to run test cases
void test(char *first, char *second)
{ match(first, second)? puts("Yes"): puts("No"); }

// Driver program to test above functions
int main()
{
    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
    return 0;
}
```

## Python

```
# 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 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

# This code is contributed by BHAVYA JAIN and ROHIT SIKKA
```

Output:

```
Yes
Yes
No
No
```

Yes

No

Yes

Yes

### Exercise

1) In the above solution, all non-wild characters of first string must be there in 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](#) 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'.

This article is compiled by [Vishal Chaudhary](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### Source

<https://www.geeksforgeeks.org/wildcard-character-matching/>



## Chapter 62

# Pattern Searching using Suffix Tree

Pattern Searching using Suffix Tree - GeeksforGeeks

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  $n > m$ .

### Preprocess Pattern or Preprocess Text?

We have discussed the following algorithms in the previous posts:

[KMP Algorithm](#)

[Rabin Karp Algorithm](#)

[Finite Automata based Algorithm](#)

[Boyer Moore Algorithm](#)

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is  $O(n)$  where  $n$  is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in  $O(m)$  time where  $m$  is length of the pattern.

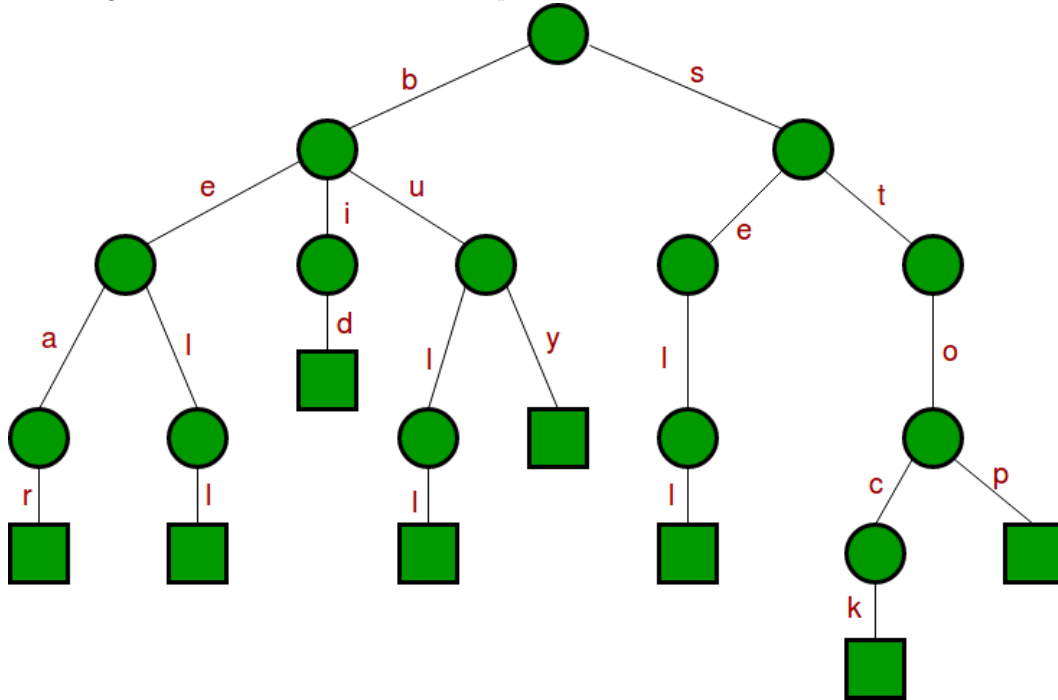
Imagine you have stored complete work of [William Shakespeare](#) and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.

Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

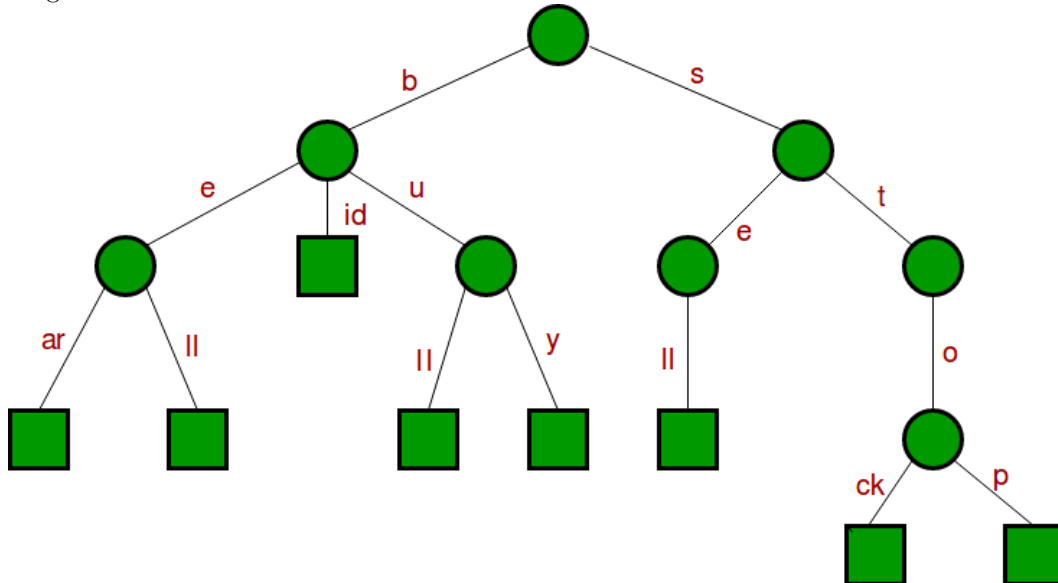
**A Suffix Tree for a given text is a compressed trie for all suffixes of the given text.** We have discussed [Standard Trie](#). Let us understand **Compressed Trie** with the following array of words.

```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input set of words.



Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.



### How to build a Suffix Tree for a given text?

As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract

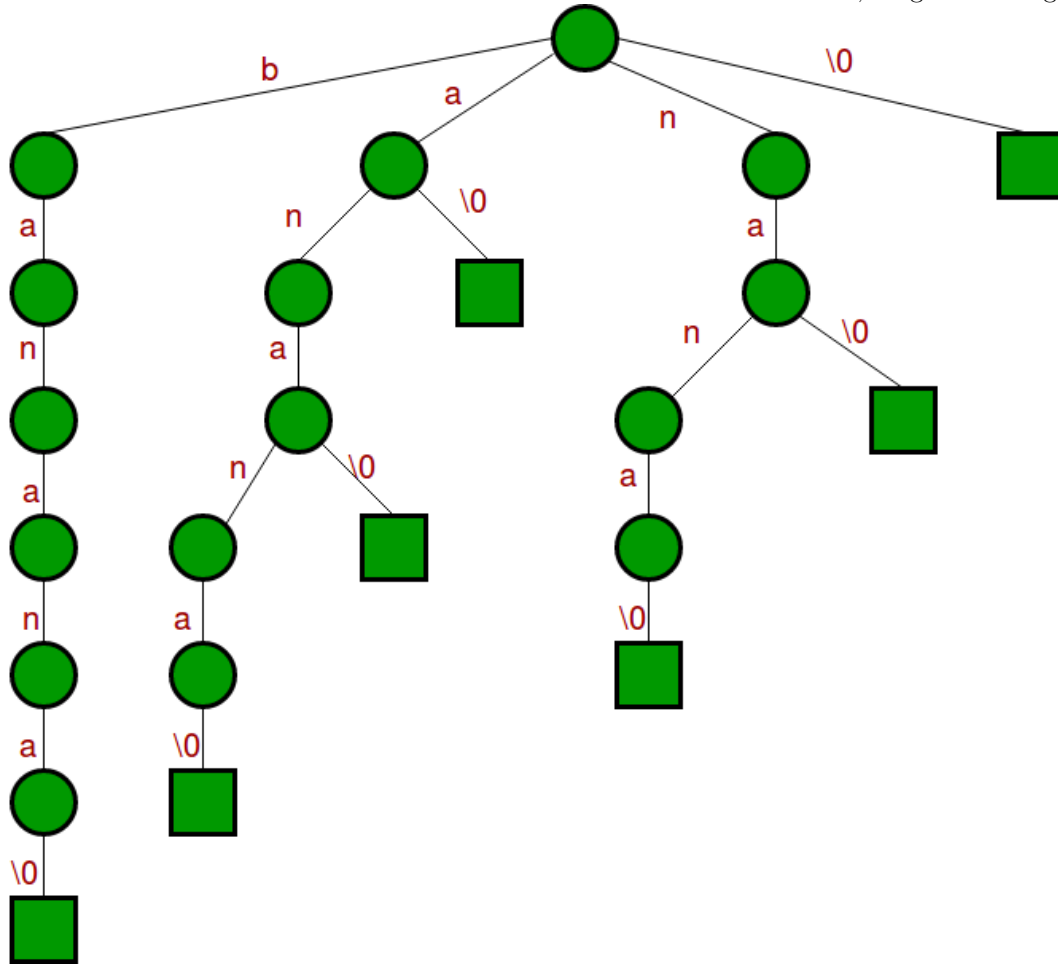
steps to build a suffix tree from given text.

- 1) Generate all suffixes of given text.
- 2) Consider all suffixes as individual words and build a compressed trie.

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

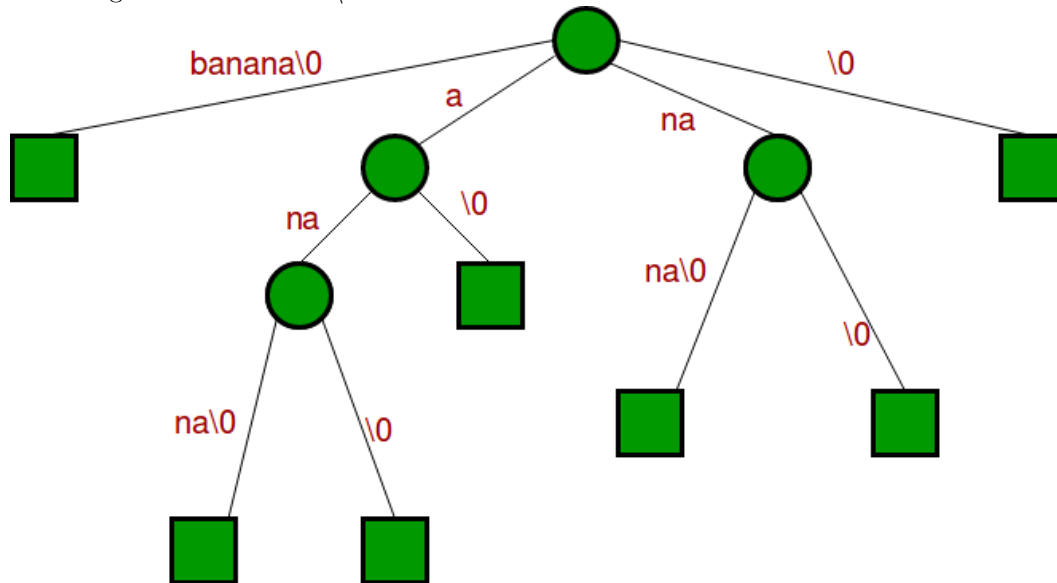
```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

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



If we join chains of single nodes, we get the following compressed trie, which is the Suffix

Tree for given text “banana\0”



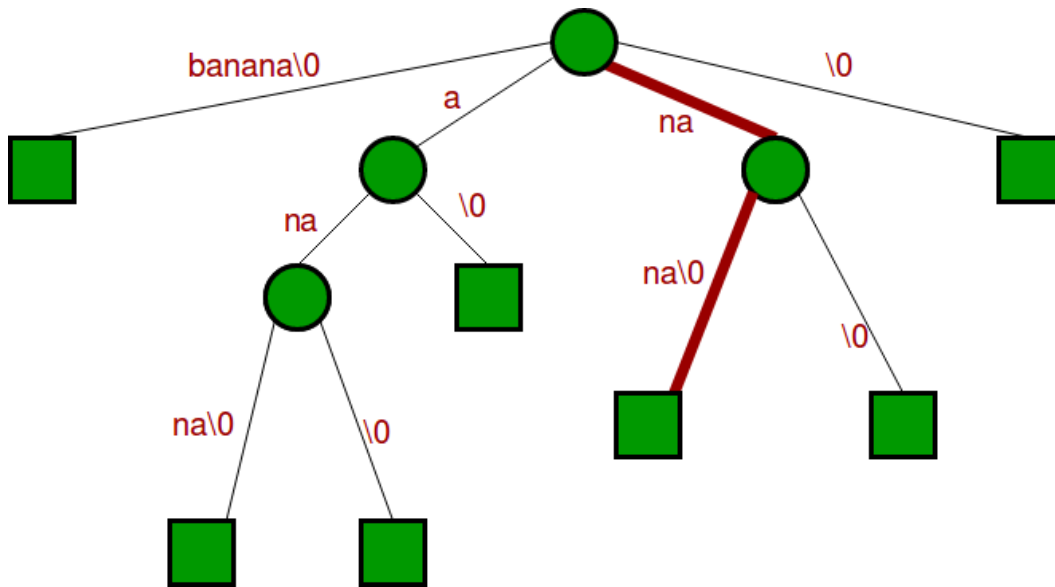
Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

#### How to search a pattern in the built suffix tree?

We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.

- 1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.
  - .....a) For the current character of pattern, if there is an edge from the current node of suffix tree, 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 “Pattern found”.

Let us consider the example pattern as “nan” to see the searching process. Following diagram shows the path followed for searching “nan” or “nana”.

**How does this work?**

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. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

**Applications of Suffix Tree**

Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.

- 1) [Pattern Searching](#)
- 2) [Finding the longest repeated substring](#)
- 3) [Finding the longest common substring](#)
- 4) [Finding the longest palindrome in a string](#)

There are many more applications. See [this](#) for more details.

Ukkonen's Suffix Tree Construction is discussed in following articles:

- [Ukkonen's Suffix Tree Construction – Part 1](#)
- [Ukkonen's Suffix Tree Construction – Part 2](#)
- [Ukkonen's Suffix Tree Construction – Part 3](#)
- [Ukkonen's Suffix Tree Construction – Part 4](#)
- [Ukkonen's Suffix Tree Construction – Part 5](#)
- [Ukkonen's Suffix Tree Construction – Part 6](#)

**Source**

<https://www.geeksforgeeks.org/pattern-searching-using-suffix-tree/>

## Chapter 63

# Boyer Moore Algorithm for Pattern Searching

Boyer Moore Algorithm for Pattern Searching - GeeksforGeeks

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results. A typical problem statement would be- 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  $n > m$ .

Examples:

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10
```

```
Input:  txt[] = "AABAACAADAABAABA"
        pat[] = "AABA"
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```

In this post, we will discuss Boyer Moore pattern searching algorithm. Like [KMP](#) and [Finite Automata](#) algorithms, Boyer Moore algorithm also preprocesses the pattern.

Boyer Moore is a combination of following two approaches.

- 1) Bad Character Heuristic
- 2) Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore

algorithm. If we take a look at the [Naive algorithm](#), it slides the pattern over the text one by one. [KMP algorithm](#) does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preprocesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. **So it uses best of the two heuristics at every step.**

Unlike the previous pattern searching algorithms, **Boyer Moore algorithm starts matching from the last character of the pattern.**

In this post, we will discuss bad character heuristic, and discuss Good Suffix heuristic in the next post.

### Bad Character Heuristic


The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of pattern is called the **Bad Character**. Upon mismatch we shift the pattern until –

- 1) The mismatch become a match
- 2) Pattern P move past the mismatch character.

#### Case 1 – Mismatch become match

We will lookup the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then we'll shift the pattern such that it get aligned to the mismatching character in text T.

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| G | C | A | A | T | G | C | C | T | A | T  | G  | T  | G  | A  | C  | C  |
| T | A | T | G | T | G |   |   |   |   |    |    |    |    |    |    |    |



|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| G | C | A | A | T | G | C | C | T | A | T  | G  | T  | G  | A  | C  | C  |
|   |   | T | A | T | G | T | G |   |   |    |    |    |    |    |    |    |

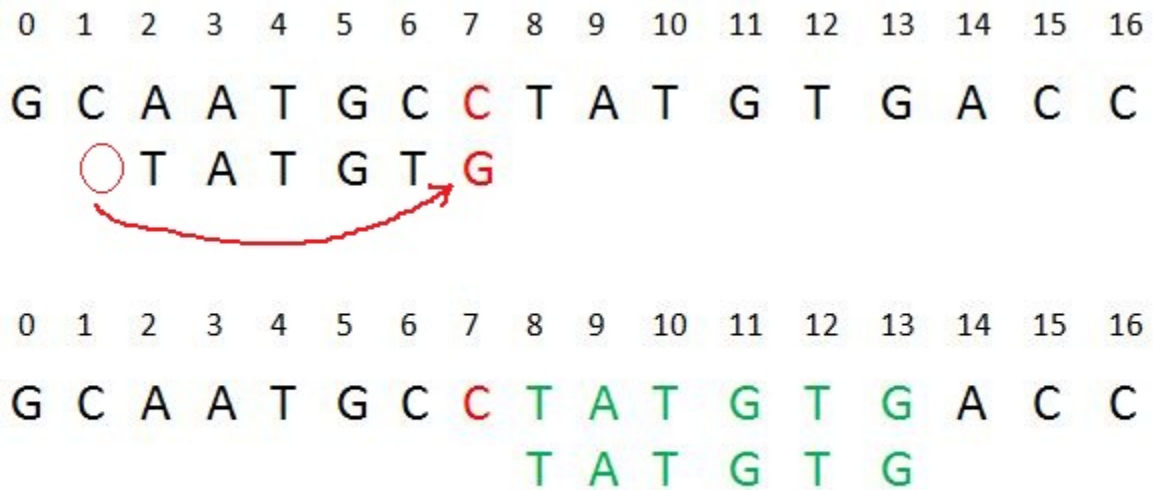
case 1

**Explanation:** In the above example, we got a mismatch at position 3. Here our mismatching character is “A”. Now we will search for last occurrence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.

#### Case 2 – Pattern move past the mismatch character

We'll lookup the position of last occurrence of mismatching character in pattern and if

character does not exist we will shift pattern past the mismatching character.



case2

**Explanation:** Here we have a mismatch at position 7. The mismatching character “C” does not exist in pattern before position 7 so we’ll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because, “C” do not exist in pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

In following implementation, we preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad

character heuristic takes  $O(n/m)$  time in the best case.

C

```
/* Program for Bad Character Heuristic of Boyer
   Moore String Matching Algorithm */
#include <limits.h>
#include <string.h>
#include <stdio.h>

#define NO_OF_CHARS 256

// A utility function to get maximum of two integers
int max (int a, int b) { return (a > b)? a: b; }
```



```
// The preprocessing function for Boyer Moore's
// bad character heuristic
void badCharHeuristic( char *str, int size,
                      int badchar[NO_OF_CHARS])
{
    int i;

    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;

    // Fill the actual value of last occurrence
    // of a character
    for (i = 0; i < size; i++)
        badchar[(int) str[i]] = i;
}

/* A pattern searching function that uses Bad
   Character Heuristic of Boyer Moore Algorithm */
void search( char *txt, char *pat)
{
    int m = strlen(pat);
    int n = strlen(txt);

    int badchar[NO_OF_CHARS];

    /* Fill the bad character array by calling
       the preprocessing function badCharHeuristic()
       for given pattern */
    badCharHeuristic(pat, m, badchar);

    int s = 0; // s is shift of the pattern with
               // respect to text
    while(s <= (n - m))
    {
        int j = m-1;

        /* Keep reducing index j of pattern while
           characters of pattern and text are
           matching at this shift s */
        while(j >= 0 && pat[j] == txt[s+j])
            j--;

        /* If the pattern is present at current
           shift, then index j will become -1 after
           the above loop */
        if (j < 0)
        {

```

```
printf("\n pattern occurs at shift = %d", s);

/* Shift the pattern so that the next
character in text aligns with the last
occurrence of it in pattern.
The condition s+m < n is necessary for
the case when pattern occurs at the end
of text */
s += (s+m < n)? m-badchar[txt[s+m]] : 1;

}

else
/* Shift the pattern so that the bad character
in text aligns with the last occurrence of
it in pattern. The max function is used to
make sure that we get a positive shift.
We may get a negative shift if the last
occurrence of bad character in pattern
is on the right side of the current
character. */
s += max(1, j - badchar[txt[s+j]]);
}

}

/* Driver program to test above funtion */
int main()
{
    char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
    return 0;
}
```

## Python

```
# Python3 Program for Bad Character Heuristic
# of Boyer Moore String Matching Algorithm

NO_OF_CHARS = 256

def badCharHeuristic(string, size):
    '''
    The preprocessing function for
    Boyer Moore's bad character heuristic
    '''

    # Initialize all occurrence as -1
```

```
badChar = [-1]*NO_OF_CHARS

# Fill the actual value of last occurrence
for i in range(size):
    badChar[ord(string[i])] = i;

# return initialized list
return badChar

def search(txt, pat):
    '''
    A pattern searching function that uses Bad Character
    Heuristic of Boyer Moore Algorithm
    '''
    m = len(pat)
    n = len(txt)

    # create the bad character list by calling
    # the preprocessing function badCharHeuristic()
    # for given pattern
    badChar = badCharHeuristic(pat, m)

    # s is shift of the pattern with respect to text
    s = 0
    while(s <= n-m):
        j = m-1

        # Keep reducing index j of pattern while
        # characters of pattern and text are matching
        # at this shift s
        while j>=0 and pat[j] == txt[s+j]:
            j -= 1

        # If the pattern is present at current shift,
        # then index j will become -1 after the above loop
        if j<0:
            print("Pattern occur at shift = {}".format(s))

            '''
            Shift the pattern so that the next character in text
            aligns with the last occurrence of it in pattern.
            The condition s+m < n is necessary for the case when
            pattern occurs at the end of text
            '''
            s += (m-badChar[ord(txt[s+m])] if s+m<n else 1)
        else:
            '''
            Shift the pattern so that the bad character in text
```

```
        aligns with the last occurrence of it in pattern. The
        max function is used to make sure that we get a positive
        shift. We may get a negative shift if the last occurrence
        of bad character in pattern is on the right side of the
        current character.
'''
s += max(1, j-badChar[ord(txt[s+j])])

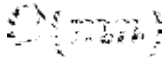
# Driver program to test above funtion
def main():
    txt = "ABAAABCD"
    pat = "ABC"
    search(txt, pat)

if __name__ == '__main__':
    main()

# This code is contributed by Atul Kumar
# (www.facebook.com/atul.kr.007)
```

Output:

```
pattern occurs at shift = 4
```

The Bad Character Heuristic may take  time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = "AAAAAAAAAAAAAAAAAAAA" and pat[] = "AAAAA".

[Boyer Moore Algorithm | Good Suffix heuristic](#)

This article is co-authored by [Atul Kumar](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

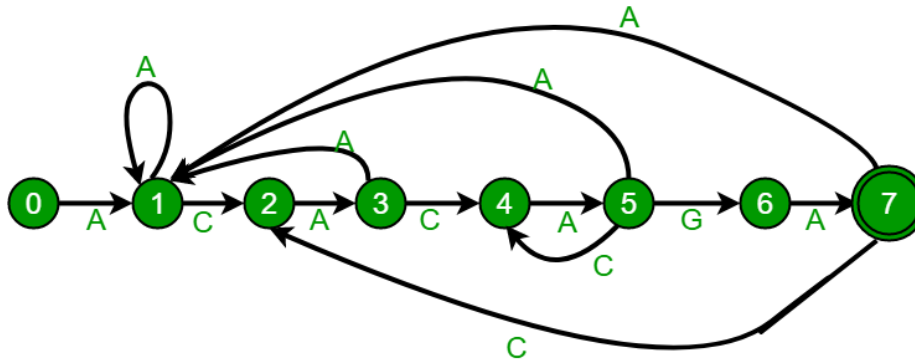
<https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/>

## Chapter 64

# Pattern Searching | Set 6 (Efficient Construction of Finite Automata)

Pattern Searching | Set 6 (Efficient Construction of Finite Automata) - GeeksforGeeks

In the [previous post](#), we discussed Finite Automata based pattern searching algorithm. The FA (Finite Automata) construction method discussed in previous post takes  $O(m^3 \cdot \text{NO\_OF\_CHARS})$  time. FA can be constructed in  $O(m \cdot \text{NO\_OF\_CHARS})$  time. In this post, we will discuss the  $O(m \cdot \text{NO\_OF\_CHARS})$  algorithm for FA construction. The idea is similar to lps (longest prefix suffix) array construction discussed in the [KMP algorithm](#). We use previously filled rows to fill a new row.



| state | character |   |   |   |
|-------|-----------|---|---|---|
|       | A         | C | G | T |
| 0     | 1         | 0 | 0 | 0 |
| 1     | 1         | 2 | 0 | 0 |
| 2     | 3         | 0 | 0 | 0 |
| 3     | 1         | 4 | 0 | 0 |
| 4     | 5         | 0 | 0 | 0 |
| 5     | 1         | 4 | 6 | 0 |
| 6     | 7         | 0 | 0 | 0 |
| 7     | 1         | 2 | 0 | 0 |

The above diagrams represent graphical and tabular representations of pattern ACACAGA.

#### Algorithm:

- 1) Fill the first row. All entries in first row are always 0 except the entry for pat[0] character. For pat[0] character, we always need to go to state 1.
- 2) Initialize lps as 0. lps for the first index is always 0.
- 3) Do following for rows at index i = 1 to M. (M is the length of the pattern)
  - .....a) Copy the entries from the row at index equal to lps.
  - .....b) Update the entry for pat[i] character to i+1.
  - .....c) Update lps “lps = TF[lps][pat[i]]” where TF is the 2D array which is being constructed.

#### Implementation

Following is C implementation for the above algorithm.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

/* This function builds the TF table which represents Finite Automata for a
   given pattern */
void computeTransFun(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int i, lps = 0, x;

    // Fill entries in first row
    for (x = 0; x < NO_OF_CHARS; x++)
        TF[0][x] = 0;
    TF[0][pat[0]] = 1;

    // Fill entries in other rows
    for (i = 1; i <= M; i++)
    {
        // Copy values from row at index lps
        for (x = 0; x < NO_OF_CHARS; x++)
            TF[i][x] = TF[lps][x];

        // Update the entry corresponding to this character
```

```
        TF[i][pat[i]] = i + 1;

        // Update lps for next row to be filled
        if (i < M)
            lps = TF[lps][pat[i]];
    }
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTransFun(pat, M, TF);

    // process text over FA.
    int i, j=0;
    for (i = 0; i < N; i++)
    {
        j = TF[j][txt[i]];
        if (j == M)
        {
            printf ("\n pattern found at index %d", i-M+1);
        }
    }
}

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEKS";
    search(pat, txt);
    getchar();
    return 0;
}
```

Output:

```
pattern found at index 0
pattern found at index 10
```

Time Complexity for FA construction is  $O(M \cdot \text{NO\_OF\_CHARS})$ . The code for search is

same as the [previous post](#) and time complexity for it is  $O(n)$ .

**Improved By :** [ParishrutPandey](#)

### Source

<https://www.geeksforgeeks.org/pattern-searching-set-5-efficient-construction-of-finite-automata/>



## Chapter 65

# Finite Automata algorithm for Pattern Searching

Finite Automata algorithm for Pattern Searching - GeeksforGeeks

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  $n > m$ .

Examples:

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
```

```
Output: Pattern found at index 10
```

```
Input:  txt[] = "AABAACAADAABAABA"
        pat[] = "AABA"
```

```
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

[Naive Algorithm](#)

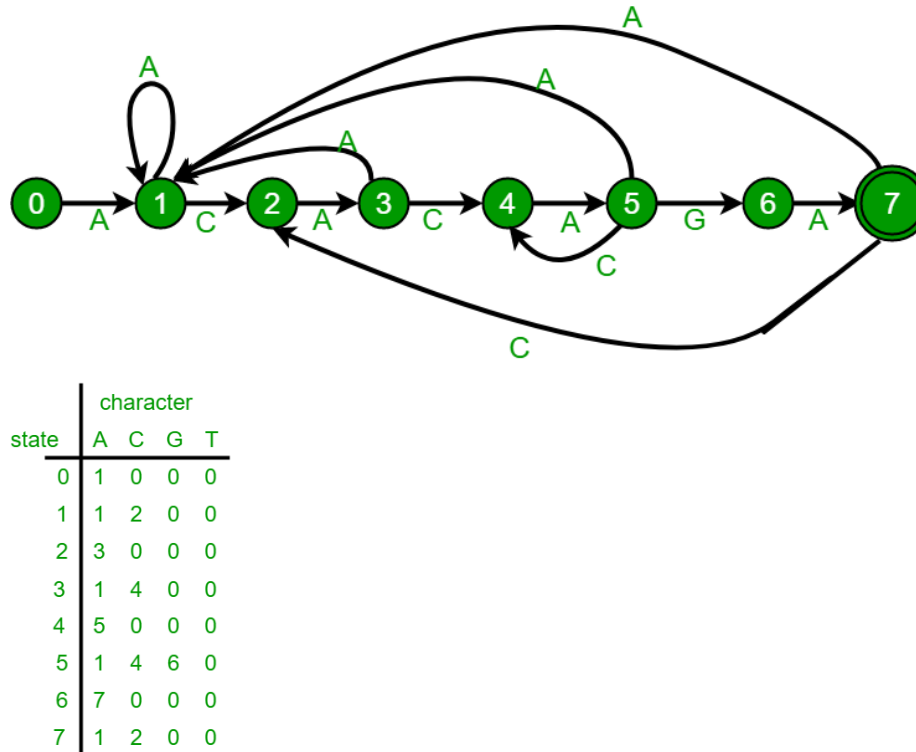
[KMP Algorithm](#)

[Rabin Karp Algorithm](#)

In this post, we will discuss Finite Automata (FA) based pattern searching algorithm. In FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite

Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and the first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to a new state. If we reach the final state, then the pattern is found in the text. The time complexity of the search process is  $O(n)$ .

Before we discuss FA construction, let us take a look at the following FA for pattern ACACAGA.



The above diagrams represent graphical and tabular representations of pattern ACACAGA.

Number of states in FA will be  $M+1$  where  $M$  is length of the pattern. The main thing to construct FA is to get the next state from the current state for every possible character. Given a character  $x$  and a state  $k$ , we can get the next state by considering the string “pat[0..k-1]x” which is basically concatenation of pattern characters pat[0], pat[1] ... pat[k-1] and the character  $x$ . The idea is to get length of the longest prefix of the given pattern such that the prefix is also suffix of “pat[0..k-1]x”. The value of length gives us the next state. For example, let us see how to get the next state from current state 5 and character ‘C’ in the above diagram. We need to consider the string, “pat[0..4]C” which is “ACACAC”. The length of the longest prefix of the pattern such that the prefix is suffix of “ACACAC” is 4 (“ACAC”). So the next state (from state 5) is 4 for character ‘C’.

In the following code, computeTF() constructs the FA. The time complexity of the computeTF() is  $O(m^3 \cdot \text{NO\_OF\_CHARS})$  where  $m$  is length of the pattern and NO\_OF\_CHARS is size of alphabet (total number of possible characters in pattern and text). The implementation tries all possible prefixes starting from the longest possible

that can be a suffix of “pat[0..k-1]x”. There are better implementations to construct FA in  $O(m \cdot \text{NO\_OF\_CHARS})$  (Hint: we can use something like [lps array construction in KMP algorithm](#)). We have covered the better implementation in our [next post on pattern searching](#).

C

```
// C program for Finite Automata Pattern searching
// Algorithm
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
    // If the character c is same as next character
    // in pattern, then simply increment state
    if (state < M && x == pat[state])
        return state+1;

    // ns stores the result which is next state
    int ns, i;

    // ns finally contains the longest prefix
    // which is also suffix in "pat[0..state-1]c"

    // Start from the largest possible value
    // and stop when you find a prefix which
    // is also suffix
    for (ns = state; ns > 0; ns--)
    {
        if (pat[ns-1] == x)
        {
            for (i = 0; i < ns-1; i++)
                if (pat[i] != pat[state-ns+1+i])
                    break;
            if (i == ns-1)
                return ns;
        }
    }

    return 0;
}

/* This function builds the TF table which represents4
   Finite Automata for a given pattern */
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int state, x;
```

```
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
            TF[state][x] = getNextState(pat, M, state, x);
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);

    // Process txt over FA.
    int i, state=0;
    for (i = 0; i < N; i++)
    {
        state = TF[state][txt[i]];
        if (state == M)
            printf ("\n Pattern found at index %d",
                    i-M+1);
    }
}

// Driver program to test above function
int main()
{
    char *txt = "AABAACAADAABAAABAA";
    char *pat = "AABA";
    search(pat, txt);
    return 0;
}
```

## Java

```
// Java program for Finite Automata Pattern
// searching Algorithm
class GFG {

    static int NO_OF_CHARS = 256;
    static int getNextState(char[] pat, int M,
                            int state, int x)
    {

        // If the character c is same as next
        // character in pattern, then simply
```

```
// increment state
if(state < M && x == pat[state])
    return state + 1;

// ns stores the result which is next state
int ns, i;

// ns finally contains the longest prefix
// which is also suffix in "pat[0..state-1]c"

// Start from the largest possible value
// and stop when you find a prefix which
// is also suffix
for (ns = state; ns > 0; ns--)
{
    if (pat[ns-1] == x)
    {
        for (i = 0; i < ns-1; i++)
            if (pat[i] != pat[state-ns+1+i])
                break;
        if (i == ns-1)
            return ns;
    }
}

return 0;
}

/* This function builds the TF table which
represents Finite Automata for a given pattern */
static void computeTF(char[] pat, int M, int TF[][])
{
    int state, x;
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
            TF[state][x] = getNextState(pat, M, state, x);
}

/* Prints all occurrences of pat in txt */
static void search(char[] pat, char[] txt)
{
    int M = pat.length;
    int N = txt.length;

    int[] [] TF = new int[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);
}
```

```
// Process txt over FA.
int i, state = 0;
for (i = 0; i < N; i++)
{
    state = TF[state][txt[i]];
    if (state == M)
        System.out.println("Pattern found "
                           + "at index " + (i-M+1));
}

// Driver code
public static void main(String[] args)
{
    char[] pat = "AABAACAADAABAAABAA".toCharArray();
    char[] txt = "AABA".toCharArray();
    search(txt,pat);
}

// This code is contributed by debjitdbb.
```

## Python

```
# Python program for Finite Automata
# Pattern searching Algorithm

NO_OF_CHARS = 256

def getNextState(pat, M, state, x):
    '''
    calculate the next state
    '''

    # If the character c is same as next character
    # in pattern, then simply increment state

    if state < M and x == ord(pat[state]):
        return state+1

    i=0
    # ns stores the result which is next state

    # ns finally contains the longest prefix
    # which is also suffix in "pat[0..state-1]c"

    # Start from the largest possible value and
    # stop when you find a prefix which is also suffix
```

```
for ns in range(state,0,-1):
    if ord(pat[ns-1]) == x:
        while(i<ns-1):
            if pat[i] != pat[state-ns+1+i]:
                break
            i+=1
        if i == ns-1:
            return ns
return 0

def computeTF(pat, M):
    '''
    This function builds the TF table which
    represents Finite Automata for a given pattern
    '''
    global NO_OF_CHARS

    TF = [[0 for i in range(NO_OF_CHARS)]\
           for _ in range(M+1)]

    for state in range(M+1):
        for x in range(NO_OF_CHARS):
            z = getNextState(pat, M, state, x)
            TF[state][x] = z

    return TF

def search(pat, txt):
    '''
    Prints all occurrences of pat in txt
    '''
    global NO_OF_CHARS
    M = len(pat)
    N = len(txt)
    TF = computeTF(pat, M)

    # Process txt over FA.
    state=0
    for i in range(N):
        state = TF[state][ord(txt[i])]
        if state == M:
            print("Pattern found at index: {}".\
                  format(i-M+1))

# Driver program to test above function
def main():
    txt = "AABAACAADAABAAABAA"
    pat = "AABA"
```

```
search(pat, txt)

if __name__ == '__main__':
    main()

# This code is contributed by Atul Kumar
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

**References:**

[Introduction to Algorithms](#) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

**Improved By :** [debjitdbb](#)

**Source**

<https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/>



## Chapter 66

# Optimized Naive Algorithm for Pattern Searching

Optimized Naive Algorithm for Pattern Searching - GeeksforGeeks

**Question:** We have discussed Naive String matching algorithm [here](#). Consider a situation where all characters of pattern are different. Can we modify [the original Naive String Matching algorithm](#) so that it works better for these types of patterns. If we can, then what are the changes to original algorithm?

**Solution:** In the [original Naive String matching algorithm](#), we always slide the pattern by 1. When all characters of pattern are different, we can slide the pattern by more than 1. Let us see how can we do this. When a mismatch occurs after j matches, we know that the first character of pattern will not match the j matched characters because all characters of pattern are different. So we can always slide the pattern by j without missing any valid shifts. Following is the modified code that is optimized for the special patterns.

C

```
/* C program for A modified Naive Pattern Searching
   algorithm that is optimized for the cases when all
   characters of pattern are different */
#include<stdio.h>
#include<string.h>

/* A modified Naive Pattern Searching algorithm that is optimized
   for the cases when all characters of pattern are different */
void search(char pat[], char txt[])
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i = 0;
```

```
while (i <= N - M)
{
    int j;

    /* For current index i, check for pattern match */
    for (j = 0; j < M; j++)
        if (txt[i+j] != pat[j])
            break;

    if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
    {
        printf("Pattern found at index %d \n", i);
        i = i + M;
    }
    else if (j == 0)
        i = i + 1;
    else
        i = i + j; // slide the pattern by j
}

/* Driver program to test above function */
int main()
{
    char txt[] = "ABCEABCDABCEABCD";
    char pat[] = "ABCD";
    search(pat, txt);
    return 0;
}
```

## Python

```
# Python program for A modified Naive Pattern Searching
# algorithm that is optimized for the cases when all
# characters of pattern are different
def search(pat, txt):
    M = len(pat)
    N = len(txt)
    i = 0

    while i <= N-M:
        # For current index i, check for pattern match
        for j in xrange(M):
            if txt[i+j] != pat[j]:
                break
            j += 1

        if j==M:      # if pat[0...M-1] = txt[i,i+1,...i+M-1]
```

```

        print "Pattern found at index " + str(i)
        i = i + M
    elif j==0:
        i = i + 1
    else:
        i = i+ j      # slide the pattern by j

# Driver program to test the above function
txt = "ABCEABCDABCEABCD"
pat = "ABCD"
search(pat, txt)

# This code is contributed by Bhavya Jain

```

## PHP

```

<?php
// PHP program for A modified Naive
// Pattern Searching algorithm that
// is optimized for the cases when all
// characters of pattern are different

/* A modified Naive Pettern Searching
algorithm that is optimized for the
cases when all characters of
pattern are different */
function search($pat, $txt)
{
    $M = strlen($pat);
    $N = strlen($txt);
    $i = 0;

    while ($i <= $N - $M)
    {
        $j;

        /* For current index i,
        check for pattern match */
        for ($j = 0; $j < $M; $j++)
            if ($txt[$i + $j] != $pat[$j])
                break;

        // if pat[0...M-1] =
        // txt[i, i+1, ...i+M-1]
        if ($j == $M)
        {
            echo("Pattern found at index $i"."\\n" );
            $i = $i + $M;
        }
    }
}

```

```
    }
    else if ($j == 0)
        $i = $i + 1;
    else

        // slide the pattern by j
        $i = $i + $j;
    }
}

// Driver Code
$txt = "ABCEABCDABCEABCD";
$pat = "ABCD";
search($pat, $txt);

// This code is contributed by nitin mittal.
?>
```

Output:

```
Pattern found at index 4
Pattern found at index 12
```

Improved By : [nitin mittal](#)

Source

<https://www.geeksforgeeks.org/optimized-naive-algorithm-for-pattern-searching/>

## Chapter 67

# Rabin-Karp Algorithm for Pattern Searching

Rabin-Karp Algorithm for Pattern Searching - GeeksforGeeks

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  $n > m$ .

**Examples:**

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
```

```
Output: Pattern found at index 10
```

```
Input:  txt[] = "AABAACAADAABAABA"
        pat[] = "AABA"
```

```
Output: Pattern found at index 0
```

```
        Pattern found at index 9
```

```
        Pattern found at index 12
```

The [Naive String Matching](#) algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.

Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

- 1) Pattern itself.
- 2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size  $m$  of text, we must have a hash function which has following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say  $hash(txt[s+1 .. s+m])$  must be efficiently computable from  $hash(txt[s .. s+m-1])$  and  $txt[s+m]$  i.e.,  $hash(txt[s+1 .. s+m]) = rehash(txt[s+m], hash(txt[s .. s+m-1]))$  and rehash must be  $O(1)$  operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

$$hash(txt[s+1 .. s+m]) = (d * (hash(txt[s .. s+m-1]) - txt[s] * h) + txt[s+m]) \bmod q$$

$hash(txt[s .. s+m-1])$  : Hash value at shift  $s$ .

$hash(txt[s+1 .. s+m])$  : Hash value at next shift (or shift  $s+1$ )

$d$ : Number of characters in the alphabet

$q$ : A prime number

$h$ :  $d^{m-1}$

C/C++

```
/* Following program is a C implementation of Rabin Karp
Algorithm given in the CLRS book */
```

```
#include<stdio.h>
```

```
#include<string.h>
```

```
// d is the number of characters in the input alphabet
```

```
#define d 256
```

```
/* pat -> pattern
```

```
txt -> text
```

```
q -> A prime number
```

```
*/
```

```
void search(char pat[], char txt[], int q)
```

```
{
```

```
    int M = strlen(pat);
```

```
    int N = strlen(txt);
```

```
    int i, j;
```

```
    int p = 0; // hash value for pattern
```

```
    int t = 0; // hash value for txt
```

```
    int h = 1;
```

```
    // The value of h would be "pow(d, M-1)%q"
```

```
    for (i = 0; i < M-1; i++)
```

```
        h = (h*d)%q;
```

```
// Calculate the hash value of pattern and first
// window of text
for (i = 0; i < M; i++)
{
    p = (d*p + pat[i])%q;
    t = (d*t + txt[i])%q;
}

// Slide the pattern over text one by one
for (i = 0; i <= N - M; i++)
{
    // Check the hash values of current window of text
    // and pattern. If the hash values match then only
    // check for characters one by one
    if ( p == t )
    {
        /* Check for characters one by one */
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }

        // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
        if (j == M)
            printf("Pattern found at index %d \n", i);
    }

    // Calculate hash value for next window of text: Remove
    // leading digit, add trailing digit
    if ( i < N-M )
    {
        t = (d*(t - txt[i]*h) + txt[i+M])%q;

        // We might get negative value of t, converting it
        // to positive
        if (t < 0)
            t = (t + q);
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "GEEKS FOR GEEKS";
```

```
char pat[] = "GEEK";
int q = 101; // A prime number
search(pat, txt, q);
return 0;
}
```

#### Java

```
// Following program is a Java implementation
// of Rabin Karp Algorithm given in the CLRS book

public class Main
{
    // d is the number of characters in the input alphabet
    public final static int d = 256;

    /* pat -> pattern
       txt -> text
       q -> A prime number
    */
    static void search(String pat, String txt, int q)
    {
        int M = pat.length();
        int N = txt.length();
        int i, j;
        int p = 0; // hash value for pattern
        int t = 0; // hash value for txt
        int h = 1;

        // The value of h would be "pow(d, M-1)%q"
        for (i = 0; i < M-1; i++)
            h = (h*d)%q;

        // Calculate the hash value of pattern and first
        // window of text
        for (i = 0; i < M; i++)
        {
            p = (d*p + pat.charAt(i))%q;
            t = (d*t + txt.charAt(i))%q;
        }

        // Slide the pattern over text one by one
        for (i = 0; i <= N - M; i++)
        {
            // Check the hash values of current window of text
            // and pattern. If the hash values match then only
            // check for characters one by one
```



```
        if ( p == t )
        {
            /* Check for characters one by one */
            for (j = 0; j < M; j++)
            {
                if (txt.charAt(i+j) != pat.charAt(j))
                    break;
            }

            // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if (j == M)
                System.out.println("Pattern found at index " + i);
        }

        // Calculate hash value for next window of text: Remove
        // leading digit, add trailing digit
        if ( i < N-M )
        {
            t = (d*(t - txt.charAt(i)*h) + txt.charAt(i+M))%q;

            // We might get negative value of t, converting it
            // to positive
            if (t < 0)
                t = (t + q);
        }
    }
}

/* Driver program to test above function */
public static void main(String[] args)
{
    String txt = "GEEKS FOR GEEKS";
    String pat = "GEEK";
    int q = 101; // A prime number
    search(pat, txt, q);
}

// This code is contributed by nuclide
```

## Python

```
# Following program is the python implementation of
# Rabin Karp Algorithm given in CLRS book

# d is the number of characters in the input alphabet
d = 256
```

```
# pat -> pattern
# txt -> text
# q    -> A prime number

def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    i = 0
    j = 0
    p = 0    # hash value for pattern
    t = 0    # hash value for txt
    h = 1

    # The value of h would be "pow(d, M-1)%q"
    for i in xrange(M-1):
        h = (h*d)%q

    # Calculate the hash value of pattern and first window
    # of text
    for i in xrange(M):
        p = (d*p + ord(pat[i]))%q
        t = (d*t + ord(txt[i]))%q

    # Slide the pattern over text one by one
    for i in xrange(N-M+1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters one by one
        if p==t:
            # Check for characters one by one
            for j in xrange(M):
                if txt[i+j] != pat[j]:
                    break

            j+=1
            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j==M:
                print "Pattern found at index " + str(i)

    # Calculate hash value for next window of text: Remove
    # leading digit, add trailing digit
    if i < N-M:
        t = (d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

        # We might get negative values of t, converting it to
        # positive
        if t < 0:
            t = t+q
```

```
# Driver program to test the above function
txt = "GEEKS FOR GEEKS"
pat = "GEEK"
q = 101 # A prime number
search(pat,txt,q)

# This code is contributed by Bhavya Jain
```

## PHP

```
<?php
// Following program is a PHP
// implementation of Rabin Karp
// Algorithm given in the CLRS book

// d is the number of characters
// in the input alphabet
$d = 256;

/* pat -> pattern
   txt -> text
   q -> A prime number
*/
function search($pat, $txt, $q)
{
    $M = strlen($pat);
    $N = strlen($txt);
    $i, $j;
    $p = 0; // hash value
           // for pattern
    $t = 0; // hash value
           // for txt
    $h = 1;
    $d = 1;

    // The value of h would
    // be "pow(d, M-1)%q"
    for ($i = 0; $i < $M - 1; $i++)
        $h = ($h * $d) % $q;

    // Calculate the hash value
    // of pattern and first
    // window of text
    for ($i = 0; $i < $M; $i++)
    {
        $p = ($d * $p + $pat[$i]) % $q;
        $t = ($d * $t + $txt[$i]) % $q;
```

```
}

// Slide the pattern over
// text one by one
for ($i = 0; $i <= $N - $M; $i++)
{
    // Check the hash values of
    // current window of text
    // and pattern. If the hash
    // values match then only
    // check for characters on
    // by one
    if ($p == $t)
    {
        // Check for characters
        // one by one
        for ($j = 0; $j < $M; $j++)
        {
            if ($txt[$i + $j] != $pat[$j])
                break;
        }

        // if p == t and pat[0...M-1] =
        // txt[i, i+1, ...i+M-1]
        if ($j == $M)
            echo "Pattern found at index ",
                $i, "\n";
    }

    // Calculate hash value for
    // next window of text:
    // Remove leading digit,
    // add trailing digit
    if ($i < $N - $M)
    {
        $t = ($d * ($t - $txt[$i] *
                    $h) + $txt[$i +
                    $M]) % $q;

        // We might get negative
        // value of t, converting
        // it to positive
        if ($t < 0)
            $t = ($t + $q);
    }
}
}
```

```
// Driver Code
$txt = "GEEKS FOR GEEKS";
$pat = "GEEK";
$q = 101; // A prime number
search($pat, $txt, $q);

// This code is contributed
// by ajit
?>
```

**Output:**

```
Pattern found at index 0
Pattern found at index 10
```

The average and best case running time of the Rabin-Karp algorithm is  $O(n+m)$ , but its worst-case time is  $O(nm)$ . Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of `txt[]` match with hash value of `pat[]`. For example `pat[] = "AAA"` and `txt[] = "AAAAAAA"`.

**References:**

<http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap34.htm>

<http://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf>

[http://en.wikipedia.org/wiki/Rabin-Karp\\_string\\_search\\_algorithm](http://en.wikipedia.org/wiki/Rabin-Karp_string_search_algorithm)

**Related Posts:**

[Searching for Patterns | Set 1 \(Naive Pattern Searching\)](#)

[Searching for Patterns | Set 2 \(KMP Algorithm\)](#)

**Improved By :** [Hao Lee](#), [jit\\_t](#)

**Source**

<https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>

## Chapter 68

# KMP Algorithm for Pattern Searching

KMP Algorithm for Pattern Searching - GeeksforGeeks

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  $n > m$ .

**Examples:**

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
```

```
Output: Pattern found at index 10
```

```
Input:  txt[] = "AABAACAADAABAABA"
        pat[] = "AABA"
```

```
Output: Pattern found at index 0
```

```
        Pattern found at index 9
```

```
        Pattern found at index 12
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed Naive pattern searching algorithm in the [previous post](#). The worst case complexity of the Naive algorithm is  $O(m(n-m+1))$ . The time complexity of KMP algorithm is  $O(n)$  in the worst case.

### KMP (Knuth Morris Pratt) Pattern Searching

The [Naive pattern searching algorithm](#) doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] = "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to  $O(n)$ . The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

#### Matching Overview

```
txt = "AAAAABAAABA"
pat = "AAAA"
```

```
We compare first window of txt with pat
txt = "AAAAABAAABA"
pat = "AAAA" [Initial position]
We find a match. This is same as Naive String Matching.
```

```
In the next step, we compare next window of txt with pat.
txt = "AAAAABAAABA"
pat = "AAAA" [Pattern shifted one position]
This is where KMP does optimization over Naive. In this
second window, we only compare fourth A of pattern
with fourth character of current window of text to decide
whether current window matches or not. Since we know
first three characters will anyway match, we skipped
matching first three characters.
```

#### Need of Preprocessing?

An important question arises from the above explanation, how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array `lps[]` that tells us the count of characters to be skipped.

#### Preprocessing Overview:

- KMP algorithm preprocesses `pat[]` and constructs an auxiliary **`lps[]`** of size `m` (same as size of pattern) which is used to skip characters while matching.
- **name `lps` indicates longest proper prefix which is also suffix..** A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".

- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern `pat[0..i]` where  $i = 0$  to  $m-1$ , `lps[i]` stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern `pat[0..i]`.

`lps[i]` = the longest proper prefix of `pat[0..i]`  
          which is also a suffix of `pat[0..i]`.

Examples of `lps[]` construction:

For the pattern "AAAA",

`lps[]` is [0, 1, 2, 3]

For the pattern "ABCDE",

`lps[]` is [0, 0, 0, 0, 0]

For the pattern "AABAACAABAA",

`lps[]` is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern "AAACAAAAAC",

`lps[]` is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern "AABABAA",

`lps[]` is [0, 1, 2, 0, 1, 2, 3]

### Searching Algorithm:

Unlike [Naive algorithm](#), where we slide the pattern by one and compare all characters at each shift, we use a value from `lps[]` to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use `lps[]` to decide next positions (or to know a number of characters to be skipped)?

- We start comparison of `pat[j]` with  $j = 0$  with characters of current window of text.
- We keep matching characters `txt[i]` and `pat[j]` and keep incrementing  $i$  and  $j$  while `pat[j]` and `txt[i]` keep **matching**.
- When we see a **mismatch**
  - We know that characters `pat[0..j-1]` match with `txt[i-j+1...i-1]` (Note that  $j$  starts with 0 and increment it only when there is a match).
  - We also know (from above definition) that `lps[j-1]` is count of characters of `pat[0...j-1]` that are both proper prefix and suffix.
  - From above two points, we can conclude that we do not need to match these `lps[j-1]` characters with `txt[i-j...i-1]` because we know that these characters will anyway match. Let us consider above example to understand this.

`txt[]` = "AAAAABAAABA"



```
pat[] = "AAAA"
lps[] = {0, 1, 2, 3}
```

```
i = 0, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++
```

```
i = 1, j = 1
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++
```

```
i = 2, j = 2
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
pat[i] and pat[j] match, do i++, j++
```

```
i = 3, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++
```

```
i = 4, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3
```

Here unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.

```
i = 4, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++
```

```
i = 5, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3
```

Again unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.

```
i = 5, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[2] = 2
```

```
i = 5, j = 2
txt[] = "AAAAABAAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[1] = 1
```

```
i = 5, j = 1
txt[] = "AAAAABAAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[0] = 0
```

```
i = 5, j = 0
txt[] = "AAAAABAAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j is 0, we do i++.
```

```
i = 6, j = 0
txt[] = "AAAAABAAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++ and j++
```

```
i = 7, j = 1
txt[] = "AAAAABAAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++ and j++
```

We continue this way...

C++

```
// C++ program for implementation of KMP pattern searching
// algorithm
#include <bits/stdc++.h>

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
```

```
computeLPSArray(pat, M, lps);

int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }

    if (j == M) {
        printf("Found pattern at index %d ", i - j);
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.

```

```
        if (len != 0) {
            len = lps[len - 1];

            // Also, note that we do not increment
            // i here
        }
        else // if (len == 0)
        {
            lps[i] = 0;
            i++;
        }
    }
}

// Driver program to test above function
int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

### Java

```
// JAVA program for implementation of KMP pattern
// searching algorithm

class KMP_String_Matching {
    void KMPSearch(String pat, String txt)
    {
        int M = pat.length();
        int N = txt.length();

        // create lps[] that will hold the longest
        // prefix suffix values for pattern
        int lps[] = new int[M];
        int j = 0; // index for pat[]

        // Preprocess the pattern (calculate lps[]
        // array)
        computeLPSArray(pat, M, lps);

        int i = 0; // index for txt[]
        while (i < N) {
            if (pat.charAt(j) == txt.charAt(i)) {
                j++;
            }
        }
    }
}
```

```
        i++;
    }
    if (j == M) {
        System.out.println("Found pattern "
                           + "at index " + (i - j));
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat.charAt(j) != txt.charAt(i)) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}
}

void computeLPSArray(String pat, int M, int lps[])
{
    // length of the previous longest prefix suffix
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M) {
        if (pat.charAt(i) == pat.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = len;
            }
        }
    }
}
```

```
        i++;
    }
}

// Driver program to test above function
public static void main(String args[])
{
    String txt = "ABABDABACDABABCABAB";
    String pat = "ABABCABAB";
    new KMP_String_Matching().KMPSearch(pat, txt);
}
// This code has been contributed by Amit Khandelwal.
```

### Python

```
# Python program for KMP Algorithm
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    # create lps[] that will hold the longest prefix suffix
    # values for pattern
    lps = [0]*M
    j = 0 # index for pat[]

    # Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps)

    i = 0 # index for txt[]
    while i < N:
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == M:
            print "Found pattern at index " + str(i-j)
            j = lps[j-1]

    # mismatch after j matches
    elif i < N and pat[j] != txt[i]:
        # Do not match lps[0..lps[j-1]] characters,
        # they will match anyway
        if j != 0:
            j = lps[j-1]
        else:
```

```
        i += 1

def computeLPSArray(pat, M, lps):
    len = 0 # length of the previous longest prefix suffix

    lps[0] # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i]== pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            # This is tricky. Consider the example.
            # AAACAAAA and i = 7. The idea is similar
            # to search step.
            if len != 0:
                len = lps[len-1]

            # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i += 1

txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
KMPSearch(pat, txt)
```

# This code is contributed by Bhavya Jain

## C#

```
// C# program for implementation of KMP pattern
// searching algorithm
using System;

class GFG {

    void KMPSearch(string pat, string txt)
    {
        int M = pat.Length;
        int N = txt.Length;

        // create lps[] that will hold the longest
        // prefix suffix values for pattern
        int[] lps = new int[M];
```

```
int j = 0; // index for pat[]

// Preprocess the pattern (calculate lps[]
// array)
computeLPSArray(pat, M, lps);

int i = 0; // index for txt[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }
    if (j == M) {
        Console.WriteLine("Found pattern "
                           + "at index " + (i - j));
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}

}

void computeLPSArray(string pat, int M, int[] lps)
{
    // length of the previous longest prefix suffix
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
```



```
        // to search step.
        if (len != 0) {
            len = lps[len - 1];

            // Also, note that we do not increment
            // i here
        }
        else // if (len == 0)
        {
            lps[i] = len;
            i++;
        }
    }
}

// Driver program to test above function
public static void Main()
{
    string txt = "ABABDABACDABABCABAB";
    string pat = "ABABCABAB";
    new GFG().KMPSearch(pat, txt);
}

// This code has been contributed by Amit Khandelwal.
```

Output:

Found pattern at index 10

### Preprocessing Algorithm:

In the preprocessing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]. See computeLPSArray () in the below code for details.

### Illustration of preprocessing (or construction of lps[])

```
pat[] = "AAACAAAA"
```

```
len = 0, i = 0.
lps[0] is always 0, we move
to i = 1
```

```
len = 0, i = 1.
```

```
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 1, lps[1] = 1, i = 2

len = 1, i = 2.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 2, lps[2] = 2, i = 3

len = 2, i = 3.
Since pat[len] and pat[i] do not match, and len > 0,
set len = lps[len-1] = lps[1] = 1

len = 1, i = 3.
Since pat[len] and pat[i] do not match and len > 0,
len = lps[len-1] = lps[0] = 0

len = 0, i = 3.
Since pat[len] and pat[i] do not match and len = 0,
Set lps[3] = 0 and i = 4.

len = 0, i = 4.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 1, lps[4] = 1, i = 5

len = 1, i = 5.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 2, lps[5] = 2, i = 6

len = 2, i = 6.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 3, lps[6] = 3, i = 7

len = 3, i = 7.
Since pat[len] and pat[i] do not match and len > 0,
set len = lps[len-1] = lps[2] = 2

len = 2, i = 7.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 3, lps[7] = 3, i = 8

We stop here as we have constructed the whole lps[].
```

Improved By : [nitin mittal](#), [Krushi Raj](#), [geekyzeus](#), [codekrafter](#)

## **Source**

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

## Chapter 69

# Naive algorithm for Pattern Searching

Naive algorithm for Pattern Searching - GeeksforGeeks

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  $n > m$ .

Examples:

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
```

```
Output: Pattern found at index 10
```

```
Input:  txt[] = "AABAACAADAABAABA"
        pat[] = "AABA"
```

```
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

### Naive Pattern Searching:

Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

C

```
// C program for Naive Pattern Searching algorithm
#include <stdio.h>
```

```
#include <string.h>

void search(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++) {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            printf("Pattern found at index %d \n", i);
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";
    search(pat, txt);
    return 0;
}
```

[/sourcecode]

## Java

```
// Java program for Naive Pattern Searching
public class NaiveSearch {

    public static void search(String txt, String pat)
    {
        int M = pat.length();
        int N = txt.length();

        /* A loop to slide pat one by one */
        for (int i = 0; i <= N - M; i++) {

            int j;
```

```
        /* For current index i, check for pattern
        match */
        for (j = 0; j < M; j++)
            if (txt.charAt(i + j) != pat.charAt(j))
                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            System.out.println("Pattern found at index " + i);
    }
}

public static void main(String[] args)
{
    String txt = "AABAACAADAABAAABAA";
    String pat = "AABA";
    search(txt, pat);
}
// This code is contributed by Harikishore
```

## C#

```
// C# program for Naive Pattern Searching
using System;

class GFG
{
    public static void search(String txt, String pat)
    {
        int M = pat.Length;
        int N = txt.Length;

        /* A loop to slide pat one by one */
        for (int i = 0; i <= N - M; i++)
        {
            int j;

            /* For current index i, check for pattern
            match */
            for (j = 0; j < M; j++)
                if (txt[i + j] != pat[j])
                    break;

            // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if (j == M)
                Console.WriteLine("Pattern found at index " + i);
        }
    }
}
```

```
    }

    // Driver code
    public static void Main()
    {
        String txt = "AABAACAADAABAAABAA";
        String pat = "AABA";
        search(txt, pat);
    }
}
// This code is Contributed by Sam007
```

## PHP

```
<?php
// PHP program for Naive Pattern
// Searching algorithm

function search($pat, $txt)
{
    $M = strlen($pat);
    $N = strlen($txt);

    // A loop to slide pat[]
    // one by one
    for ($i = 0; $i <= $N - $M; $i++)
    {
        // For current index i,
        // check for pattern match
        for ($j = 0; $j < $M; $j++)
            if ($txt[$i + $j] != $pat[$j])
                break;

        // if pat[0...M-1] =
        // txt[i, i+1, ...i+M-1]
        if ($j == $M)
            echo "Pattern found at index ", $i."\n";
    }
}

// Driver Code
$txt = "AABAACAADAABAAABAA";
$pat = "AABA";
search($pat, $txt);

// This code is contributed by Sam007
?>
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

#### What is the best case?

The best case occurs when the first character of the pattern is not present in text at all.

```
txt[] = "AABCCAADDEE";
pat[] = "FAA";
```

The number of comparisons in best case is  $O(n)$ .

#### What is the worst case ?

The worst case of Naive Pattern Searching occurs in following scenarios.

1) When all characters of the text and pattern are same.

```
txt[] = "AAAAAAAAAAAAAAAAAAAA";
pat[] = "AAAAA";
```

2) Worst case also occurs when only the last character is different.

```
txt[] = "AAAAAAAAAAAAAAAAAAB";
pat[] = "AAAAB";
```

The number of comparisons in the worst case is  $O(m*(n-m+1))$ . Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the worst case to  $O(n)$ . We will be covering KMP in the next post. Also, we will be writing more posts to cover all pattern searching algorithms and data structures.

**Improved By :** [Sam007](#), [Brij Raj Kishore](#)

#### Source

<https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/>