Back To Course (/batchPage.php?batchId=308)

LIVE BATCHES

Learn

Learn

Quiz

Learn Problems Quiz

We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

Learn

Introduction to Strings

Strings are defined as a stream of characters. Strings are used to represent text and are generally represented by enclosing text within quotes as: *"This is a sample string!"*.

Different programming languages have different ways of declaring and using Strings. We will learn to implement strings in **C/C++ and Java**.

Strings in C/C++

In C/C++, Strings are defined as an array of characters. The difference between a character array and a string is that the string is terminated with a special character '\0'.

Declaring Strings: Declaring a string is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

```
char str_name[size];
```

In the above syntax, *str_name* is any name given to the string variable and size is used to define the length of the string, i.e the number of characters strings will store. Please keep in mind that there is an extra terminating character which is the Null character ('\0') used to indicate the termination of string which differs strings from normal character arrays.

Initializing a String: A string can be initialized in different ways. We will explain this with the help of an example. Below is an example to declare a string with the name as *str* and initialize it with **"GeeksforGeeks"**.

1. char str[] = "GeeksforGeeks";

2. char str[50] = "GeeksforGeeks";

3. char str[] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};

4. char str[14] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};

Printing a string array: Unlike arrays we do not need to print a string, character by character. The C/C++ language does not provide an inbuilt data type for strings but it has an access specifier "**%s**" which can be used to directly print and read strings.

```
1
2 // C/C++ program to illustrate strings
```

```
#include<bits/stdc++.h>

int main()
{
    // declare and initialize string
    char str[] = "Geeks";

    // print string
    printf("%s",str);

    return 0;
}
```

Run

Output:

```
Geeks
```

Passing strings to function: As strings are character arrays, so we can pass strings to function in the same way we pass an array to a function. Below is a sample program to do this:

```
// C/C++ program to illustrate how to
// pass strings to function

#include<bits/stdc++.h>

void printStr(char str[])
{
    printf("String is : %s",str);
}

int main()
{
    // declare and initialize string
    char str[] = "GeeksforGeeks";

    // print string by passing string
    // to a different function
    printStr(str);

    return 0;
}
```

Run

Output:

```
String is : GeeksforGeeks
```

std::string Class in C++

C++ has in its definition a way to represent the sequence of characters as an object of a class. This class is called **std::string**. The String class stores the characters as a sequence of bytes with functionality of allowing access to single byte character.

string Class vs Character array:

- A character array is simply an array of characters can terminated by a null character. A string is a class which defines objects that be represented as stream of characters.
- Size of the character array has to allocated statically, more memory cannot be allocated at run time if required. Unused allocated memory is wasted in case of character array. In case of strings, memory is allocated dynamically. More memory can be allocated at run time on demand. As no memory is preallocated, no memory is wasted.
- Implementation of character array is faster than std:: string. Strings are slower when compared to implementation than character array.
- Character array does not offer much inbuilt functions to manipulate strings. String class defines a number of functionalities which allow manifold operations on strings.

Declaration Syntax: Declaring string using string class is simple and can be done using the *string* keyword as shown below.

```
string string_name = "Sample String";
```

Sample Program:

```
// C++ program to illustrate strings

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

int main()
{
    // declare and initialize string
    string str = "Geeks";

    // print string
    cout<<str;

    return 0;
}

```

Run

Output:

```
Geeks
```

To learn about std::string in details, refer: std::string class in C++ (https://www.geeksforgeeks.org/stdstring-class-in-c/).

Strings in Java

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String

There are two ways to create string in Java:

- *String literal*

```
String s = “GeeksforGeeks”;
```

- Using *new* keyword

```
String s = new String (“GeeksforGeeks”);
```

String Methods

1. **int length():** Returns the number of characters in the String.

```
"GeeksforGeeks".length();  // returns 13
```

2. **Char charAt(int i) (https://www.geeksforgeeks.org/java-string-charat-method-example/):** Returns the character at i^{th} index.

```
"GeeksforGeeks".charAt(3); // returns  ‘k’
```

3. **String substring (int i) (https://www.geeksforgeeks.org/substring-in-java/):** Return the substring from the i^{th} index character to end.

```
"GeeksforGeeks".substring(3); // returns “ksforGeeks”
```

4. **String substring (int i, int j) (https://www.geeksforgeeks.org/substring-in-java/):** Returns the substring from i to j-1 index.

```
 "GeeksforGeeks".substring(2, 5); // returns “eks”
```

5. **String concat(String str) (https://www.geeksforgeeks.org/java-string-concat-examples/):** Concatenates specified string to the end of this string.

```
String s1 = ”Geeks”;

String s2 = ”forGeeks”;

String output = s1.concat(s2); // returns “GeeksforGeeks”
```

6. **int indexOf (String s) (https://www.geeksforgeeks.org/java-string-indexof/):** Returns the index within the string of the first occurrence of the specified string.

```
String s = ”Learn Share Learn”;

int output = s.indexOf(“Share”); // returns 6
```

7. **int indexOf (String s, int i) (https://www.geeksforgeeks.org/java-string-indexof/):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = ”Learn Share Learn”;

int output = s.indexOf(‘a’,3);// returns 8
```

8. **Int lastIndexOf(String s) (https://www.geeksforgeeks.org/java-lang-string-lastindexof-method/):** Returns the index within the string of the last occurrence of the specified string.

```
String s = ”Learn Share Learn”;

int output = s.lastIndexOf(‘a’); // returns 14
```

9. **boolean equals(Object otherObj):** Compares this string to the specified object.

```
Boolean out = “Geeks”.equals(“Geeks”); // returns true

Boolean out = “Geeks”.equals(“geeks”); // returns false
```

10. **boolean equalsIgnoreCase (String anotherString) (https://www.geeksforgeeks.org/equalsignorecase-in-java/):** Compares string to another string, ignoring case considerations.

```
Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true

Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true
```

11. **int compareTo(String anotherString) (https://www.geeksforgeeks.org/java-lang-string-compareto/):** Compares two string lexicographically.

```
int out = s1.compareTo(s2);  // where s1 ans s2 are

                             // strings to be compared



This returns difference s1-s2. If :

out < 0  // s1 comes before s2

out = 0  // s1 and s2 are equal.

out > 0   // s1 comes after s2.
```

12. **int compareToIgnoreCase(String anotherString):** Compares two string lexicographically, ignoring case considerations.

```
 int out = s1.compareToIgnoreCase(s2);

// where s1 ans s2 are

// strings to be compared



 This returns difference s1-s2. If :

 out < 0  // s1 comes before s2

 out = 0   // s1 and s2 are equal.

 out > 0   // s1 comes after s2.
```

Note- In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).

13. **String toLowerCase() (https://www.geeksforgeeks.org/java-string-tolowercase-examples/):** Converts all the characters in the String to lower case.

```
String word1 = “HeLLo”;

String word3 = word1.toLowerCase(); // returns “hello"
```

14. **String toUpperCase() (https://www.geeksforgeeks.org/java-touppercase-examples/):** Converts all the characters in the String to upper case.

```
String word1 = “HeLLo”;

String word2 = word1.toUpperCase(); // returns “HELLO”
```

15. **String trim() (https://www.geeksforgeeks.org/java-string-trim-method-example/):** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = “ Learn Share Learn “;

String word2 = word1.trim(); // returns “Learn Share Learn”
```

16. **String replace (char oldChar, char newChar) (https://www.geeksforgeeks.org/java-lang-string-replace-method-java/):** Returns new string by replacing all occurrences of *oldChar* with *newChar.*

```
String s1 = “feeksforfeeks“;

String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // returns “geeksgorgeeks”
```

Note:- s1 is still feeksforfeeks and s2 is geeksgorgeeks

Program to illustrate all string methods:

```
// Java code to illustrate different constructors and methods
// String class.

import java.io.*;
import java.util.*;
class Test
{
    public static void main (String[] args)
    {
        String s= "GeeksforGeeks";
        // or String s= new String ("GeeksforGeeks");

        // Returns the number of characters in the String.
        System.out.println("String length = " + s.length());

        // Returns the character at ith index.
        System.out.println("Character at 3rd position = "
                           + s.charAt(3));

        // Return the substring from the ith index character
        // to end of string
        System.out.println("Substring " + s.substring(3));

        // Returns the substring from i to j-1 index.
        System.out.println("Substring = " + s.substring(2,5));

        // Concatenates string2 to the end of string1.
        String s1 = "Geeks";
        String s2 = "forGeeks";
```

Run

Output :

```
String length = 13
Character at 3rd position = k
Substring ksforGeeks
Substring = eks
Concatenated string = GeeksforGeeks
Index of Share 6
Index of a = 8
Checking Equality false
Checking Equality true
Checking Equality false
If s1 = s2 false
Changing to lower Case geekyme
Changing to UPPER Case GEEKYME
Trim the word Learn Share Learn
Original String feeksforfeeks
Replaced f with g -> geeksgorgeeks
```

LIVE BATCHES

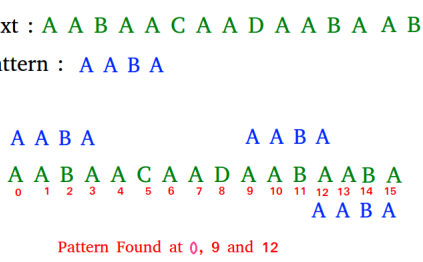
- Naive Pattern Searching Algorithm

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: The idea is to 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.

That is check for the match of the first character of the pattern in the string, if it matches then check for the subsequent characters of the pattern with the respective characters of the string. If a mismatch is found then move forward in the string.

Below is the implementation of the above approach:

C++

```cpp

// C++ program for Naive Pattern
// Searching algorithm
#include<bits/stdc++.h>
using namespace std;

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]
            cout << "Pattern found at index "
                 << i << endl;
    }
}

// Driver Code
int main()
{
```

Run

Java

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.

```
1
2  txt[] = "AABCCAADDEE";
3  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.

```
1
2  txt[] = "AAAAAAAAAAAAAAAAAA";
3  pat[] = "AAAAA";
```

2. Worst case also occurs when only the last character is different.

```
1
2  txt[] = "AAAAAAAAAAAAAAAAAB";
3  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.

− Rabin-Karp Algorithm for Pattern Searching

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.

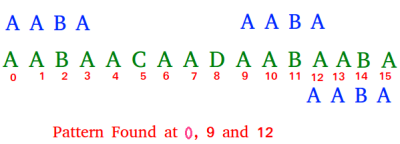
Examples:

```
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
```

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A



The *Naive String Matching* algorithm slides the pattern one by one. After each slide, one by one it checks characters at the current shift and if all characters match then it 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 the text of length m, that is of the length of pattern string.

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 the 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]) mod q*

Where,
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)

Below is the implementation of the above approach:

C/C++

```
/* Following program is a C/C++ implementation
   of Rabin Karp Algorithm */

#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
```

Run

Java

Output:

```
Pattern found at index 0
Pattern found at index 10
```

Time Complexity: 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 the same as the hash values of all the substrings of txt[] match with the hash value of pat[]. For example pat[] = "AAA" and txt[] = "AAAAAAA".

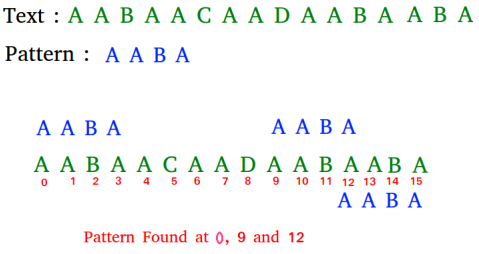
− KMP Algorithm for Pattern Searching

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
```



We have discussed the Naive pattern searching algorithm and the Rabin-Karp algorithm for searching patterns. The worst case complexity of both of the algorithms is O(n*m). Here, we will discuss a new algorithm for searching patterns, KMP algorithm. 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 the 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 the 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].
```

Note : lps[i] could also be defined as longest prefix which is also proper suffix. We need to use properly at one place to make sure that the whole substring is not considered.

```
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 “AAABAAA”,
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 the 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...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[] = "AAAAABAAABA"
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[] = "AAAAABAAABA"
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[] = "AAAAABAAABA"
pat[] =      "AAAA"
txt[i] and pat[j] do NOT match and j is 0, we do i++.

i = 6, j = 0
txt[] = "AAAAABAAABA"
pat[] =       "AAAA"
```

```
txt[i] and pat[j] match, do i++ and j++

i = 7, j = 1
txt[] = "AAAAAB**AAAB**A"
pat[] =       "**AAAA**"
txt[i] and pat[j] match, do i++ and j++

We continue this way...
```

C++

```cpp

// C++ program for implementation of KMP
// pattern searching algorithm

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

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++;
        }

```

Run

Java

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 a 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.
We know that characters pat
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 will stop here as we have constructed the whole lps[].
```

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org (mailto:feedback@geeksforgeeks.org)

(https://www.facebook.com/geeksforgeeks.org/)(https://www.instagram.com/geeks_for_geeks/)(https://in.linkedin.com/company/geeksforgeeks)htt

LIVE BATCHES

Company

About Us (https://www.geeksforgeeks.org/about/)

Careers (https://www.geeksforgeeks.org/careers/)

Privacy Policy (https://www.geeksforgeeks.org/privacy-policy/)

Contact Us (https://www.geeksforgeeks.org/about/contact-us/)

Terms of Service (https://practice.geeksforgeeks.org/terms-of-service/)

Learn

Algorithms (https://www.geeksforgeeks.org/fundamentals-of-algorithms/)

Data Structures (https://www.geeksforgeeks.org/data-structures/)

Languages (https://www.geeksforgeeks.org/category/program-output/)

CS Subjects (https://www.geeksforgeeks.org/articles-on-computer-science-subjects-gq/)

Video Tutorials (https://www.youtube.com/geeksforgeeksvideos/)

Practice

Courses (https://practice.geeksforgeeks.org/courses/)

Company-wise (https://practice.geeksforgeeks.org/company-tags/)

Topic-wise (https://practice.geeksforgeeks.org/topic-tags/)

How to begin? (https://practice.geeksforgeeks.org/faq.php)

Contribute

Write an Article (https://www.geeksforgeeks.org/contribute/)

Write Interview Experience (https://www.geeksforgeeks.org/write-interview-experience/)

Internships (https://www.geeksforgeeks.org/internship/)

Videos (https://www.geeksforgeeks.org/how-to-contribute-videos-to-geeksforgeeks/)