STRINGS

Definition: A string is a sequence of characters. In programming, strings are used to store and manipulate text data. They are usually stored as arrays of characters and terminated by a special null character (\0) in languages like C, or as immutable objects in Python, Java, and C++.

Real-life analogy: Think of a string like a train where each compartment (character) is connected in sequence, forming a meaningful message or word.

Example (Python):

```
s = “Hello, World!”

print(s)
```

The following facts make strings Van interesting data structure.

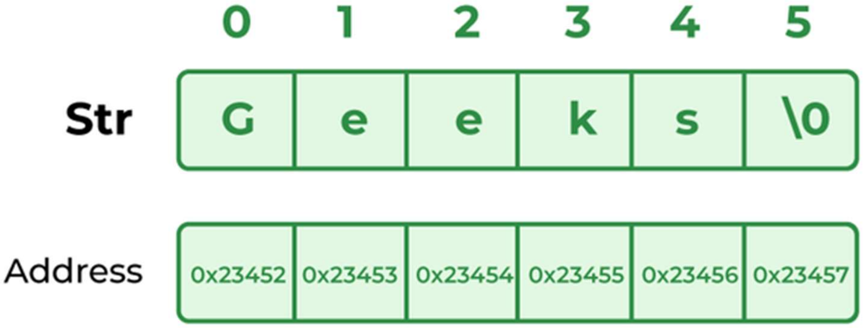
- Small set of elements. Unlike a normal array, strings typically have a smaller set of items. For example, the lowercase English alphabet has only 26 characters. ASCII has only 256 characters.
- Strings are immutable in programming languages like Java, Python, JavaScript, and C#.
- Many String Problems can be optimized using the fact that the character set size is small. For example, sorting can be done faster, counting frequencies of items is faster and many interesting interview questions are based on this.

BASICS

How are Strings represented in Memory?

In C, a string can be referred to either using a character pointer or as a character array. When strings are declared as character arrays, they are stored like other types of arrays in C. String literals (assigned to pointers) are immutable in C and C++.

In C++, strings created using the string class are mutable and internally represented as arrays. In Python, Java, and JavaScript, string characters are stored at contiguous locations (like arrays).



	0	1	2	3	4	5
Str	G	e	e	k	s	\0
Address	0x23452	0x23453	0x23454	0x23455	0x23456	0x23457

How to Declare Strings in various languages?

- **C:** Strings are declared as character arrays or pointers and must end with a null character (\0) to indicate termination.
- **C++:** Supports both C-style character arrays and the std::string class, which provides built-in functions for string manipulation.
- **Java:** Strings are immutable objects of the String class, meaning their values cannot be modified once assigned.
- **Python:** Strings are dynamic and can be declared using single, double, or triple quotes, making them flexible for multi-line text handling.
- **JavaScript:** Strings are primitive data types and can be defined using single, double, or template literals (backticks), allowing for interpolation.
- **C#:** Uses the string keyword, which represents an immutable sequence of characters, similar to Java.
- There is no character type on Python and JavaScript and a single character is also considered as a string.

```
// C++ program to demonstrate String

// using Standard String representation


#include <iostream>

#include <string>

using namespace std;
```

```
int main()
{

    // Declare and initialize the string
    string str1 = "Welcome to GeeksforGeeks!";

    // Initialization by raw string
    string str2("A Computer Science Portal");

    // Print string
    cout << str1 << endl << str2;

    return 0;
}
```

```
// C program to illustrate strings
#include <stdio.h>

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

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

    return 0;
}
```

```java
// Java code to illustrate String
import java.io.*;
import java.lang.*;

class Test {
    public static void main(String[] args)
    {
        // Declare String without using new operator
        String s = "GeeksforGeeks";

        // Prints the String.
        System.out.println("String s = " + s);

        // Declare String using new operator
        String s1 = new String("GeeksforGeeks");

        // Prints the String.
        System.out.println("String s1 = " + s1);
    }
}
```

```python
# Python Program for
# Creation of String

# Creating a String
# with single Quotes
String1 = 'Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)
```

```
# Creating a String
# with double Quotes
String1 = "I'm a Geek"
print("\nString with the use of Double Quotes: ")
print(String1)

# Creating a String
# with triple Quotes
String1 = '''I'm a Geek and I live in a world of "Geeks"'''
print("\nString with the use of Triple Quotes: ")
print(String1)

# Creating String with triple
# Quotes allows multiple lines
String1 = '''Geeks
            For
            Life'''
print("\nCreating a multiline String: ")
print(String1)
```

//C#   Include namespace system

```
using System;

public class Test
{
    public static void Main(String[] args)
    {
        // Declare String without using new operator
        var s = "GeeksforGeeks";
        // Prints the String.
        Console.WriteLine("String s = " + s);
        // Declare String using new operator
        var s1 = new  String("GeeksforGeeks");
        // Prints the String.
        Console.WriteLine("String s1 = " + s1);
    }
}
```

//JavaScript

```
// Declare and initialize the string
let str1 = "Welcome to GeeksforGeeks!";

// Initialization using another method
let str2 = new String("A Computer Science Portal");

// Print strings
console.log(str1);
console.log(str2.toString());
```

<?php

```
// single-quote strings

$site = 'Welcome to GeeksforGeeks';

echo $site;

?>
```

Are Strings Mutable in Different Languages?

- In C/C++, string literals (assigned to pointers) are immutable.
- In C++, string objects are mutable.
- In Python, Java and JavaScript, strings are immutable.

General Operations performed on String

Here we are providing you with some must-know concepts of string:

- **Length of String :** The length of a string refers to the total number of characters present in it, including letters, digits, spaces, and special characters. It is a fundamental property of strings in any programming language and is often used in various operations such as validation, manipulation, and comparison.
- **Search a Character :** Searching for a character in a string means finding the position where a specific character appears. If the character is present multiple times, you might need to find its first occurrence, last occurrence, or all occurrences.
- **Check for Substring :** Checking for a substring means determining whether a smaller sequence of characters exists within a larger string. A substring is a continuous part of a string, and checking for its presence is a common operation in text processing, search algorithms, and data validation.
- **Insert a Character :** Inserting a character into a string means adding a new character at a specific position while maintaining the original order of other characters. Since strings are immutable in many programming languages, inserting a character usually involves creating a new modified string with the desired character placed at the specified position.
- **Delete a Character :** Deleting a character from a string means removing a specific character at a given position while keeping the remaining characters intact. Since strings are immutable in many programming languages, this operation usually involves creating a new string without the specified character.
- **Check for Same Strings :** Checking if two strings are the same means comparing them character by character to determine if they are identical in terms of length, order, and content. If every character in one string matches exactly with the corresponding character in another string, they are considered the same.
- **String Concatenation :** String concatenation is the process of joining two or more strings together to form a single string. This is useful in text processing, formatting messages, constructing file paths, or dynamically creating content.
- **Reverse a String :** Reversing a string means arranging its characters in the opposite order while keeping their original positions intact in the reversed sequence. This

operation is commonly used in text manipulation, data encryption, and algorithm challenges.

- **Rotate a String** Rotating a string means shifting its characters to the left or right by a specified number of positions while maintaining the order of the remaining characters. The characters that move past the boundary wrap around to the other side.
- **Check for Palindrome :** Checking for a palindrome means determining whether a string reads the same forward and backward. A palindrome remains unchanged when reversed, making it a useful concept in text processing, algorithms, and number theory.

Length of a String

Programming Language	In-Built method to find the length of the string
C	strlen()
C++	size()
Java	length()
Python	len()
JavaScript	length
C#	length()

Examples:

Input: *s = "abc"*
Output: *3*

***Input: s** = "GeeksforGeeks"*
***Output:** 13*

***Input: s** = ""*
***Output:** 0*

Time Complexity: O(n), where n is the length of the string.
Auxiliary space: O(1)

Check if two strings are same or not

Given two strings, the task is to check if these two strings are identical(same) or not. Consider **case sensitivity.**

Examples:

Input: *s1 = "abc", s2 = "abc"*
Output: *Yes*

Input: *s1 = "", s2 = ""*
Output: *Yes*

Input: *s1 = "GeeksforGeeks", s2 = "Geeks"*
Output: *No*

Approach- using equals in Java

This approach compares two strings to check if they are the same. It works by using the equality operator (==) to compare both strings character by character. If the strings are identical, it prints "Yes", otherwise, it prints "No". The comparison stops as soon as a mismatch is found, or if one string ends before the other. The program performs this comparison in a straightforward way without any extra steps, simply checking the content of the strings.

```
public class GfG {
    // Function to compare both strings directly
    public static boolean areStringsSame(String s1, String s2) {
        return s1.equals(s2);
    }

    public static void main(String[] args) {
        String s1 = "abc";
        String s2 = "abcd";

        // Call the areStringsSame function to compare strings
        if (areStringsSame(s1, s2)) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}
```

Output

```
No
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Approach- By using String Comparison Functions

In this approach, the program compares two strings to check if they are identical. It uses the strcmp function, which compares the strings character by character. If the strings are exactly the same, strcmp returns 0, indicating no difference between them. If the strings are different, strcmp returns a non-zero value. Based on this result, the program prints "Yes" if the strings match and "No" if they don't. The comparison stops as soon as a mismatch is found or the strings end.

```
import java.util.Arrays;

class GfG {
    // Function to compare two strings using equals
    public static boolean areStringsSame(String s1, String s2) {
        return s1.equals(s2);
    }

    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";

        // Call the areStringsSame function to compare strings
        if (areStringsSame(s1, s2)) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}
```

Output

```
Yes
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Approach - By Writing your Own Method

In this approach, the program compares two strings by first checking if their lengths are the same. If the lengths differ, the strings cannot be identical, so it returns false. If the lengths are the same, it then compares each character of the two strings one by one. If any mismatch is found, it returns false, indicating the strings are not the same. If no differences are found throughout the comparison, it returns true, meaning the strings are identical.

```
public class GfG {
    public static boolean areStringsEqual(String s1, String s2) {
        // Compare lengths first
        if (s1.length() != s2.length()) {
            return false;
        }

        // Compare character by character
        for (int i = 0; i < s1.length(); i++) {
            if (s1.charAt(i) != s2.charAt(i)) {
                return false;
            }
        }

        return true;
    }

    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";


        if (areStringsEqual(s1, s2)) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}
```

Output

```
Yes
```

Time Complexity: O(n)

Auxiliary Space: O(1)

Program to Search a Character in a String

Given a character **ch** and a string **s**, the task is to find the **index** of the **first occurrence** of the character in the string. If the character is **not present** in the string, return **-1**.

Examples:

Input: *s = "geeksforgeeks", ch = 'k'*
Output: *3*
Explanation: *The character 'k' is present at index **3** and **11** in "geeksforgeeks", but it first appears at index 3.*

Input: *s = "geeksforgeeks", ch = 'z'*
Output: *-1*
Explanation: *The character 'z' is not present in "geeksforgeeks".*

Approach - By traversing the string - O(n) Time and O(1) Space

*The idea is to traverse the input string **s** and for each character, check if it is equal to the character we are searching for. If we find a match, return the **index** of the current character.*

*If we reach the **end** of the string without finding any occurrence of the character, return **-1**.*

```
// Java program to search a character in a string


class GfG {
    // function to find the first occurrence of ch in s
    static int findChar(String s, char ch) {
        int n = s.length();
        for (int i = 0; i < n; i++) {

            // If the current character is equal to ch,
            // return the current index
            if (s.charAt(i) == ch)
                return i;
        }

        // If we did not find any occurrence of ch,
        // return -1
        return -1;
    }

    public static void main(String[] args) {
        String s = "geeksforgeeks";
        char ch = 'k';

        System.out.println(findChar(s, ch));
    }
}
```

Output

```
3
```

Approach - By Using in-built library functions - O(n) Time and O(1) Space

We can also use inbuilt library functions to search for a character in a string. This makes the search simple and easier to implement.

```
public class GfG{
    public static int findCharacterIndex(String s, char ch) {
        int idx = s.indexOf(ch);
        return (idx != -1) ? idx : -1;
    }

    public static void main(String[] args) {
        String s = "geeksforgeeks";
        char ch = 'k';

        int index = findCharacterIndex(s, ch);
        System.out.println(index);
    }
}
```

Output

```
3
```

<u>Insert a character in String at a Given Position</u>

Given a string **s**, a character **c** and an integer position **pos**, the task is to insert the character **c** into the string **s** at the specified position **pos**.

Examples:

Input: *s = "Geeks", c = 'A', pos = 3*
Output: *GeeAks*

Input: *s = "HelloWorld", c = '!', pos = 5*
Output: *Hello!World*

[Approach-1] Using Built-In Methods

We will use library methods like StringBuilder insert in Java.

```
// Java program to insert a character at specific
// position using Built in functions

class GfG {
    static String insertChar(StringBuilder sb, char c, int pos) {
        // Insert character at specified position
        sb.insert(pos, c);
        return sb.toString();
    }

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Geeks");
        System.out.println(insertChar(sb, 'A', 3));
    }
}
```

Output

```
GeeAks
```

Time Complexity: O(n) where **n** is the length of the string.

[Approch-2] Using Custom Method

First, iterate through the given string, inserting all the characters into a new string until we reach the position where the given character needs to be inserted. At that position, insert the character, and then append the remaining characters from the input string to the new string.

```
// Java program to insert a character at specific
// position using custom method

class GfG {
    static String insertChar(String s, char c, int pos) {
        StringBuilder res = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {

            // Insert the character at the given position
            if (i == pos)
                res.append(c);

            // Insert the original characters
            res.append(s.charAt(i));
        }

        // If the given pos is beyond the length,
        // append the character at the end
        if (pos >= s.length())
            res.append(c);

        return res.toString();
    }

    public static void main(String[] args) {
        String s = "Geeks";
        System.out.println(insertChar(s, 'A', 3));
    }
}
```

Output

```
GeeAks
```

Time Complexity: O(n) where **n** is the length of the string.

Remove a Character from a Given Position

Given a string and a position (0-based indexing), remove the character at the given position.

Examples:

Input : *s = "abcde", pos = 1*
Output : *s = "acde"*

Input : *s = "a", pos = 0*
Output : *s = ""*

Approach - By Using Built-In Methods

We use the StringBuilderDelete in Java.

```
// Java program to demonstrate
// the deleteCharAt() Method.
```

```
class GFG {
        public static void main(String[] args)
        {

                // create a StringBuilder object
                // with a String pass as parameter
                StringBuilder s
                        = new StringBuilder("abcde");



                // print string after removal of Character
     // at index 1
                System.out.println("Output: " + s.deleteCharAt(1));
        }
}
```

Output

```
Output: acde
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Approach - By Writing Your Own Method

We move all characters after the given position, one index back. To do this we mainly do s]i] = s[i+1] for all indexes i after p.

```
public class Main {
   public static void removeCharAtPosition(StringBuilder s, int pos) {
      // Check for valid position
      if (pos < 0 || pos >= s.length()) {
         return;
      }

      // Shift characters to the left from the position
      for (int i = pos; i < s.length() - 1; i++) {
            s.setCharAt(i, s.charAt(i + 1));
        }

        // Remove the last character
        s.deleteCharAt(s.length() - 1);
    }

    public static void main(String[] args) {
        StringBuilder s = new StringBuilder("abcde");
        int pos = 1;
        removeCharAtPosition(s, pos);
        System.out.println(s);
    }
}
```

Output

```
acde
```

Time Complexity: O(n)
Auxiliary Space: O(1)

How to insert characters in a string at a certain position?

Given a string **str** and an array of indices **chars[]** that describes the indices in the original string where the characters will be added. For this post, let the character to be inserted in star (*). Each star should be inserted before the character at the given index. Return the modified string after the stars have been added.

Examples:

Input: *str = "geeksforgeeks", chars = [1, 5, 7, 9]*
Output: *g*eeks*fo*rg*eeks*
Explanation: *The indices 1, 5, 7, and 9 correspond to the bold characters in "g**e**eks**f**orgeeks".*

Input: *str = "spacing", chars = [0, 1, 2, 3, 4, 5, 6]*
Output: *"*s*p*a*c*i*n*g"*

Approach - Linear Traversal with Index Tracking

*Iterate over the string and keep track of the count of the characters in the string so far and whenever your count becomes equal to the element in the array of stars**,** append a star to the resultant string and move ahead in your star array.*

```
// Java code to implement the approach
import java.io.*;

class GFG
{

  // Function to add stars
  public static String addStars(String s, int stars[])
  {

    // Create a string ans for storing
    // resultant string
    String ans = "";

    int j = 0;

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

      // If the count of characters
      // become equal to the stars[j],
      // append star
      if (j < stars.length && i == stars[j]) {
        ans += '*';
        j++;
      }
      ans += s.charAt(i);
    }

    return ans;
  }
  // Driver Code
  public static void main(String[] args)
  {
    String str = "geeksforgeeks";
    int chars[] = { 1, 5, 7, 9 };
    String ans = addStars(str, chars);

    // Printing the resultant string
    System.out.println(ans);
  }
}
```

Output

```
g*eeks*fo*rg*eeks
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Approach - By Using inbuilt insert function

In this approach we need to increase the length of orignal string as on insert operation the orignal string get modified and so the target index needs to be increased by 1 so we used k.

Step-by-step approach:

- The addStars function takes a string s and a vector of integers stars as input.
- It iterates through the stars vector using a for loop.
- For each position specified in the stars vector, it inserts an asterisk (*) at that position in the string s(using insert function).
- we increment the k on insertion because size increases on an insertion operation.
- The updated string is returned.

```
import java.util.ArrayList;
import java.util.List;

public class GFG {
 // Function to add stars
  public static String addStars(String s, List<Integer> stars) {
    // Iterate through the vector of positions
     int k = 0;
    for (int i = 0; i < stars.size(); i++) {
      // Insert a star at the specified position
       s = s.substring(0, stars.get(i) + k) + "*" + s.substring(stars.get(i) + k);
      k++;
    }
     // Return result
    return s;
  }

  public static void main(String[] args) {
    // Test case
     String str = "geeksforgeeks";
    List<Integer> chars = new ArrayList<>();
    chars.add(1);
    chars.add(5);
    chars.add(7);
    chars.add(9);
    String ans = addStars(str, chars);
    System.out.println(ans);
  }
}
```

Output

```
g*eeks*fo*rg*eeks
```

Time Complexity: O(N*K)
Auxiliary Space: O(N)

Remove all occurrences of a character in a string

Given a string and a character, remove all the occurrences of the character in the string.

Examples:

Input : *s = "geeksforgeeks"*
c = 'e'
Output : *s = "gksforgks"*

Input : *s = "geeksforgeeks"*
c = 'g'
Output : *s = "eeksforeeks"*

Input *: s = "geeksforgeeks"*
c = 'k'
Output *: s = "geesforgees"*

Using Built-In Methods

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
       String s = "ababca";
       char c = 'a';


       // Remove all occurrences of 'c' from 's'
       s = s.replace(String.valueOf(c), "");


       System.out.println(s);
   }
}
```

Output

```
bbc
```

Writing Your Own Method

The idea is to maintain an index of the resultant string.

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

```
public class GFG {


    public static String removechar(String word, char ch)
    {
        StringBuilder s = new StringBuilder(word);
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == ch) {
                s.deleteCharAt(i);
                i--;
            }
        }


        return s.toString();
    }


    // driver's code
    public static void main(String args[])
    {
        String word = "geeksforgeeks";
        char ch = 'e';
        System.out.println(removechar(word, ch));
    }
}
```

Output

```
eeksforeeks
```

Time Complexity : O(n) where **n** is length of input string.
Auxiliary Space : O(1)

Concatenation of Two Strings

String concatenation is the process of joining two strings end-to-end to form a single string.

Examples

Input: *s1 = “Hello”, s2 = “World”*
Output: *“HelloWorld”*
Explanation: *Joining “Hello” and “World” results in “HelloWorld”.*

Input: *s1 = “Good”, s2 = “Morning”*
Output: *“GoodMorning”*
Explanation: *Joining “Good” and “Morning” results in “GoodMorning”*

Using + Operator

Almost all languages support + operator to concatenate two strings.

```
public class GfG {
    public static void main(String[] args) {
        String s1 = "Hello, ";
        String s2 = "World!";

        // Concatenating the strings
        String res = s1 + s2;

        System.out.println(res);
    }
}
```

Output

```
Hello, World!
```

Write your Own Method

- Create an empty result string.
- Traverse through s1 and append all characters to result.
- Traverse through s2 and append all characters to result.

C++CJavaPythonC#JavaScript

```
import java.util.StringBuilder;

public class Main {
    public static String concat(String s1, String s2) {
        StringBuilder res = new StringBuilder();
        // Append s1 to res
        for (char c : s1.toCharArray()) {
            res.append(c);
        }

        // Append s2 to res
        for (char c : s2.toCharArray()) {
            res.append(c);
        }

        return res.toString();
    }

    public static void main(String[] args) {
        String s1 = "Hello, ";
        String s2 = "World!";

        // Call the function to concatenate the strings
        String res = concat(s1, s2);

        System.out.println(res);
    }
}
```

Output

```
Hello, World!
```

Time Complexity : O(m + n) where m and n are lengths of the two strings.

Reverse a String

Given a string **s**, the task is to **reverse** the string. Reversing a string means **rearranging** the characters such that the **first** character becomes the **last**, the **second** character becomes **second last** and so on.

Examples:

Input: *s = "GeeksforGeeks"*
Output: *"skeeGrofskeeG"*
Explanation *: The first character **G** moves to last position, the second character **e** moves to second-last and so on.*

Input: *s = "abdcfe"*
Output: *"efcdba"*
Explanation: *The first character **a** moves to last position, the second character **b** moves to second-last and so on.*

Using backward traversal – O(n) Time and O(n) Space

*The idea is to start at the last character of the string and move backward, appending each character to a new string **res**. This new string **res** will contain the characters of the original string in reverse order.*

```
// Java program to reverse a string using backward traversal


class GfG {

    static String reverseString(String s) {

        StringBuilder res = new StringBuilder();


        // Traverse on s in backward direction

        // and add each character to a new string

        for (int i = s.length() - 1; i >= 0; i--) {

            res.append(s.charAt(i));

        }

        return res.toString();

    }


    public static void main(String[] args) {

        String s = "abdcfe";

        String res = reverseString(s);

        System.out.print(res);

    }

}
```

Output

```
efcdba
```

Time Complexity: O(n) for backward traversal
Auxiliary Space: O(n) for storing the reversed string.

Using Two Pointers - O(n) Time and O(1) Space

*The idea is to maintain two pointers: **left** and **right**, such that **left** points to the **beginning** of the string and **right** points to the **end** of the string.*

*While left pointer is less than the right pointer, swap the characters at these two positions. After each swap, **increment** the **left** pointer and **decrement** the **right** pointer to move towards the center of the string. This will swap all the characters in the first half with their corresponding character in the second half.*

```java
// Java program to reverse a string using two pointers


class GfG {
    static String reverseString(String s) {
        int left = 0, right = s.length() - 1;

        // Use StringBuilder for mutability
        StringBuilder res = new StringBuilder(s);

        // Swap characters from both ends till we reach
        // the middle of the string
        while (left < right) {
            char temp = res.charAt(left);
            res.setCharAt(left, res.charAt(right));
            res.setCharAt(right, temp);
            left++;
            right--;
        }

        // Convert StringBuilder back to string
        return res.toString();
    }


    public static void main(String[] args) {
        String s = "abdcfe";
        System.out.println(reverseString(s));
    }
}
```

Output

```
efcdba
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Using Recursion - O(n) Time and O(n) Space

The idea is to use recursion and define a ***recursive function*** *that takes a string as input and reverses it. Inside the recursive function,*

- *Swap the first and last element.*
- *Recursively call the function with the remaining substring.*

```
// Java program to reverse a string using Recursion


class GfG {


    // Recursive function to reverse a string from l to r
    static void reverseStringRec(char[] s, int l, int r) {
        if (l >= r)
            return;


        // Swap the characters at the ends
        char temp = s[l];
        s[l] = s[r];
        s[r] = temp;
        // Recur for the remaining string
        reverseStringRec(s, l + 1, r - 1);
    }

    // Function to reverse a string
    static String reverseString(String s) {
        char[] arr = s.toCharArray();
        reverseStringRec(arr, 0, arr.length - 1);
        return new String(arr);
    }

    public static void main(String[] args) {
        String s = "abdcfe";
        System.out.println(reverseString(s));
    }
}
```

Output

```
efcdba
```

Time Complexity: O(n) where n is length of string
Auxiliary Space: O(n)

Using Stack - O(n) Time and O(n) Space

The idea is to use stack for reversing a string because Stack follows ***Last In First Out (LIFO) principle.*** *This means the last character you add is the first one you'll take out. So, when we push all the characters of a string into the stack, the last character becomes the first one to pop.*

```
// Java program to reverse a string using stack
import java.util.*;

class GfG {
    static String reverseString(String s) {
        Stack<Character> st = new Stack<>();
        // Push the characters into stack
        for (int i = 0; i < s.length(); i++)
            st.push(s.charAt(i));

        StringBuilder res = new StringBuilder();

        // Pop the characters of stack into the original string
        for (int i = 0; i < s.length(); i++)
            res.append(st.pop());

        return res.toString();
    }

    public static void main(String[] args) {
        String s = "abdcfe";
        System.out.println(reverseString(s));
    }
}
```

Output

```
efcdba
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Using Inbuilt methods - O(n) Time and O(1) Space

*The idea is to use **built-in reverse** method to reverse the string. If **built-in** method for string reversal does not exist, then convert string to array or list and use their built-in method for reverse. Then convert it back to string.*

```
// Java program to reverse a string using StringBuffer class


import java.io.*;
import java.util.*;
```

```
class GFG {
    static String stringReverse(String s) {
            StringBuilder res = new StringBuilder(s);
        res.reverse();
            return res.toString();
    }

    public static void main(String[] args) {
        String s = "abdcfe";
        System.out.println(stringReverse(s));
    }
}
```

Output

```
efcdba
```

Time Complexity: O(n)

Auxiliary Space: O(n) in Java

All substrings of a given String

Given a string s, containing lowercase alphabetical characters. The task is to print all non-empty substrings of the given string.

Examples :

Input *: s = "abc"*
Output *: "a", "ab", "abc", "b", "bc", "c"*

Input *: s = "ab"*
Output *: "a", "ab", "b"*

Input *: s = "a"*
Output *: "a"*

[Expected Approach] - Using Iteration

- The idea is to use two nested loops.
- The outer loop picks the starting index (loop for i from 0 to n-1)
- The inner loop picks the ending ending index (loop for j to n-1)

```java
// Function to find all substrings
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static List<String> findSubstrings(String s) {

        // to store all substrings
        List<String> res = new ArrayList<>();
        for (int i = 0; i < s.length(); i++) {
            for (int j = i; j < s.length(); j++) {

                // substr function takes starting index
                // and ending index + 1 as parameters
                res.add(s.substring(i, j + 1));
            }
        }
        return res;
    }

    public static void main(String[] args) {
        String s = "abc";
        List<String> res = findSubstrings(s);
        for (String i : res) {
            System.out.print(i + " ");
        }
    }
}
```

Output

```
a ab abc b bc c
```

[Interesting Approach] - Using Recursion

The idea is to recursively generate all possible substrings of the given string s. To do so, create an array of string **res[]** to store the substrings of string **s** and an empty string **cur** to store the current string. Start from the **0th** index and for each index **ind**, add the current character **s[ind]** in the string **cur**, and add the string cur in **res[]**. Then, move to index **ind + 1**, and recursively generate the all substrings. At each recursive call, check if string **cur** is empty**,** if so, skip the current character to start the string from next index i**nd + 1**. At last print all the substrings.

```java
// Java program to find all the
// substrings of given string
import java.util.*;

class GFG {

    // Recursive Function to find all
    // substrings of a string
    static void subString(String s, int n, int index,
                          StringBuilder cur, List<String> res) {

        // if we have reached the
        // end of the string
        if (index == n) {
            return;
        }

        // add the character s.charAt(index)
        // to the current string
        cur.append(s.charAt(index));

        // add the current string in result
        res.add(cur.toString());
        // move to next index
        subString(s, n, index + 1, cur, res);

        // remove the current character
        // from the current string
        cur.deleteCharAt(cur.length() - 1);

        // if current string is empty
        // skip the current index to
        // start the new substring
        if (cur.length() == 0) {
            subString(s, n, index + 1, cur, res);
        }
    }

    // Function to find all substrings
    static List<String> findSubstrings(String s) {

        // to store all substrings
        List<String> res = new ArrayList<>();

        // to store current string
        StringBuilder cur = new StringBuilder();
        subString(s, s.length(), 0, cur, res);
        return res;
    }

    public static void main(String[] args) {
        String s = "abc";
        List<String> res = findSubstrings(s);
        for (String str : res) {
            System.out.print(str + " ");
        }
    }
}
```

Output

```
a ab abc b bc c
```

Pattern Matching (Naive Approach)

Concept:

Slide the pattern over the text one by one and check for a match at each position.

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

Code:

```
public class NaiveSearch {
    public static void search(String txt, String pat) {
        int n = txt.length();
        int m = pat.length();
        for (int i = 0; i <= n - m; i++) {
            if (txt.substring(i, i + m).equals(pat)) {
                System.out.println("Pattern found at index " + i);
            }
        }
    }

    public static void main(String[] args) {
        search("ababcabcabababd", "abab");
    }
}
```

Z-Algorithm

Use:

- Pattern Matching in linear time

Z-Array:

Z[i] = Length of the longest substring starting from i that matches prefix of the string.

Code:

```
public class ZAlgorithm {
  public static int[] calculateZ(String s) {
    int n = s.length();
    int[] Z = new int[n];
    int L = 0, R = 0;
    for (int i = 1; i < n; i++) {
      if (i <= R)
        Z[i] = Math.min(R - i + 1, Z[i - L]);
      while (i + Z[i] < n && s.charAt(Z[i]) == s.charAt(i + Z[i]))
        Z[i]++;
      if (i + Z[i] - 1 > R) {
        L = i;
        R = i + Z[i] - 1;
      }
    }
    return Z;
  }
}
```

Rabin-Karp Algorithm

Use:

Uses hashing to compare substrings efficiently.

Code:

```
public class RabinKarp {
  static final int d = 256;
  static final int q = 101;
```

```
    public static void search(String txt, String pat) {
        int m = pat.length();
        int n = txt.length();
        int h = 1;
        for (int i = 0; i < m - 1; i++)
            h = (h * d) % q;

        int p = 0, t = 0;
        for (int i = 0; i < m; i++) {
            p = (d * p + pat.charAt(i)) % q;
            t = (d * t + txt.charAt(i)) % q;
        }

        for (int i = 0; i <= n - m; i++) {
            if (p == t && txt.substring(i, i + m).equals(pat))
                System.out.println("Pattern found at index " + i);
            if (i < n - m) {
                t = (d * (t - txt.charAt(i) * h) + txt.charAt(i + m)) % q;
                if (t < 0) t += q;
            }
        }
    }
}
```

Trie (Prefix Tree)

Use:

- Fast search and insert operations

Code:

```
class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEnd = false;
}

public class Trie {
    private final TrieNode root = new TrieNode();

    public void insert(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            int index = ch - 'a';
            if (node.children[index] == null)
                node.children[index] = new TrieNode();
            node = node.children[index];
        }
        node.isEnd = true;
    }

    public boolean search(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            int index = ch - 'a';
            if (node.children[index] == null)
                return false;
            node = node.children[index];
        }
        return node.isEnd;
    }
}
```

Palindrome Check

Concept:

A string is a palindrome if it reads the same forwards and backwards.

Code:

```
public class PalindromeCheck {
    public static boolean isPalindrome(String s) {
        int l = 0, r = s.length() - 1;
        while (l < r) {
            if (s.charAt(l++) != s.charAt(r--))
                return false;
        }
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isPalindrome("racecar")); // true
    }
}
```

Longest Prefix Suffix (LPS)

Use:

- Core part of the KMP pattern matching algorithm

Code:

```
public class LPS {
    public static int[] computeLPS(String pattern) {
        int n = pattern.length();
        int[] lps = new int[n];
        int len = 0;
        int i = 1;
        while (i < n) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                lps[i++] = ++len;
            } else if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i++] = 0;
            }
        }
        return lps;
    }
}
```

Advanced Practice Challenges

1. **Longest Palindromic Substring**: Given a string s, return the longest palindromic substring in s. Use either dynamic programming or expand-around-center approach.
2. **Longest Common Prefix**: Write a function to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string.
3. **Group Anagrams**: Given an array of strings, group the anagrams together. You may return the answer in any order.
4. **Count and Remove Duplicates**: Write a function to count the frequency of characters in a string and remove all characters that appear more than once.
5. **String Compression**: Implement a method that performs basic string compression using the counts of repeated characters. For example, the string aabcccccaaa becomes a2b1c5a3. Return the original string if the compressed string is not smaller.

RECOMMENDED YOUTUBE LINKS

1. https://youtu.be/vCRD36bG8xQ?si=u16CEV7Mo-y1VQOj
2. https://youtu.be/zL1DPZ0Ovlo?si=dH1ht89eVLA5lhoQ
3. https://youtu.be/MwlXNPFLBzE?si=G-Olz_-De7DeeHu1
4. https://youtu.be/N63JCXwdd14?si=JBjkveQMt5NxR1mH
5. https://youtu.be/Dt6gzsNrghQ?si=kDEnCVDNM6-Iajlr
6. https://youtu.be/GuTPwotSdYw?si=mMf0EKLCDmufhJss
7. https://youtu.be/uAs73RA_BDg?si=gV6zmCFLM6rbhbz3