**String**

# Reverse a String

Write a function that reverses a string. The input string is given as an array of characters s.

**Example 1:**

**Input:** s = ["h","e","l","l","o"]

**Output:** ["o","l","l","e","h"]

**Example 2:**

**Input:** s = ["H","a","n","n","a","h"]

**Output:** ["h","a","n","n","a","H"]

**Constraints:**

* 1 <= s.length <= 105
* s[i] is a [printable ascii character](https://en.wikipedia.org/wiki/ASCII#Printable_characters).

## Solution:

class Solution {

public:

void reverseString(vector<char>& s) {

int n = s.size();

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

char temp = s[n-i-1];

s[n-i-1] = s[i];

s[i] = temp;

}

}

};

# Check whether a String is Palindrome or not

Given a string **S**, check if it is palindrome or not.

**Example 1:**

**Input:** S = "abba"

**Output:** 1

**Explanation:** S is a palindrome

**Example 2:**

**Input:** S = "abc"

**Output:** 0

**Explanation:** S is not a palindrome

**Your Task:**  
You don't need to read input or print anything. Complete the function **isPlaindrome()** which accepts string S and returns a boolean value

**Expected Time Complexity:**O(Length of S)   
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 <= Length of S <= 105

## Solution:

class Solution{

public:

int isPalindrome(string S)

{

int n = S.size();

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

if(S[i]!=S[n-1-i]){

return 0;

}

}

return 1;

}

};

# Find Duplicate characters in a string

**Write an efficient program to print all the duplicates and their counts in the input string**

## Solution:

**Method 1:**Using hashing  
**Algorithm:** Let input string be “geeksforgeeks”   
**1:** Construct character count array from the input string.  
count[‘e’] = 4   
count[‘g’] = 2   
count[‘k’] = 2   
……  
**2:**Print all the indexes from the constructed array which have values greater than 1.  
**Solution**

// C++ program to count all duplicates

// from string using hashing

#include <iostream>

using namespace std;

# define NO\_OF\_CHARS 256

class gfg

{

public :

/\* Fills count array with

frequency of characters \*/

void fillCharCounts(char \*str, int \*count)

{

int i;

for (i = 0; \*(str + i); i++)

count[\*(str + i)]++;

}

/\* Print duplicates present

in the passed string \*/

void printDups(char \*str)

{

// Create an array of size 256 and fill

// count of every character in it

int \*count = (int \*)calloc(NO\_OF\_CHARS,

sizeof(int));

fillCharCounts(str, count);

// Print characters having count more than 0

int i;

for (i = 0; i < NO\_OF\_CHARS; i++)

if(count[i] > 1)

printf("%c, count = %d \n", i, count[i]);

free(count);

}

};

/\* Driver code\*/

int main()

{

gfg g ;

char str[] = "test string";

g.printDups(str);

//getchar();

return 0;

}

**Output**

s, count = 2

t, count = 3

**Time Complexity:**O(n), where n = length of the string passed

**Space Complexity**: O(NO\_OF\_CHARS)

**Note:**Hashing involves the use of an array of fixed size each time no matter whatever the string is.

For example, str = “aaaaaaaaaa”.

An array of size 256 is used for str, only 1 block out of total size (256) will be utilized to store the number of occurrences of ‘a’ in str (i.e count[‘a’] = 10).

Rest 256 – 1 = 255 blocks remain unused.

Thus, Space Complexity is potentially high for such cases. So, to avoid any discrepancies and to improve Space Complexity, maps are generally preferred over long-sized arrays.

**Method 2:** Using Maps

**Approach:**The approach is the same as discussed in **Method 1**, but, using a map to store the count.

**Solution:**

// C++ program to count all duplicates

// from string using maps

#include <bits/stdc++.h>

using namespace std;

void printDups(string str)

{

map<char, int> count;

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

count[str[i]]++;

}

for (auto it : count) {

if (it.second > 1)

cout << it.first << ", count = " << it.second

<< "\n";

}

}

/\* Driver code\*/

int main()

{

string str = "test string";

printDups(str);

return 0;

}

**Output**

s, count = 2

t, count = 3

**Time Complexity**: O(N log N), where N = length of the string passed and it generally takes logN time for an element insertion in a map.

**Space Complexity**: O(K), where K = size of the map (**0<=K<=input\_string\_length**).

**Using unordered map**

// C++ program to count all duplicates

// from string using maps

#include <bits/stdc++.h>

using namespace std;

void printDups(string str)

{

unordered\_map<char, int> count;

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

count[str[i]]++; //increase the count of characters by 1

}

for (auto it : count) { //iterating through the unordered map

if (it.second > 1) //if the count of characters is greater then 1 then duplicate found

cout << it.first << ", count = " << it.second

<< "\n";

}

}

/\* Driver code\*/

int main()

{

string str = "test string";

printDups(str);

return 0;

}

**Output**

s, count = 2

t, count = 3

***Time Complexity:***

*O(N), where N = length of the string passed and it takes O(1) time to insert and access any element in an unordered map*

***Space Complexity:***

*O(K), where K = size of the map (0<=K<=input\_string\_length).*

# Why strings are immutable in Java?

# Why String is Immutable or Final in Java

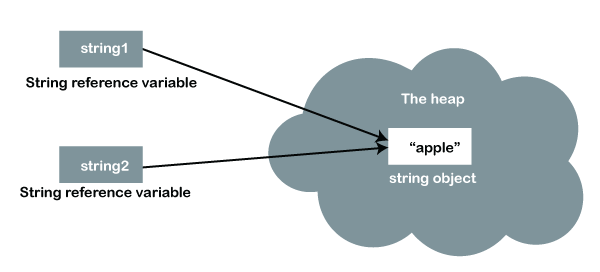
## Solution:

In object-oriented programming, the **immutable string or objects** that cannot be modified once it is created. But we can only change the reference to the object. We restrict to change the object itself. The **String is immutable** in [Java](https://www.javatpoint.com/java-tutorial)

because of the security, synchronization and concurrency, caching, and class loading. The reason of making string final is to destroy the immutability and to not allow others to extend it.

The String objects are cached in the String pool, and it makes the [String immutable](https://www.javatpoint.com/immutable-string)

. The cached String literals are accessed by multiple clients. So, there is always a risk, where action performs by one client affects all other clients. For example, if one client performs an action and changes the string value from Pressure to PRESSURE, all remaining clients will also read that value. For the performance reason, caching of String objects was important, so to remove that risk, we have to make the String Immutable.



These are some more reasons of making String immutable:

* The String pool cannot be possible if String is not immutable in Java. A lot of heap space is saved by [JRE](https://www.javatpoint.com/java-jre)

. The same string variable can be referred to by more than one string variable in the pool. String interning can also not be possible if the String would not be immutable.

* If we don't make the String immutable, it will pose a serious security threat to the application. For example, database usernames, passwords are passed as strings to receive database connections. The [socket programming](https://www.javatpoint.com/socket-programming)

host and port descriptions are also passed as strings. The String is immutable, so its value cannot be changed. If the String doesn't remain immutable, any hacker can cause a security issue in the application by changing the reference value.

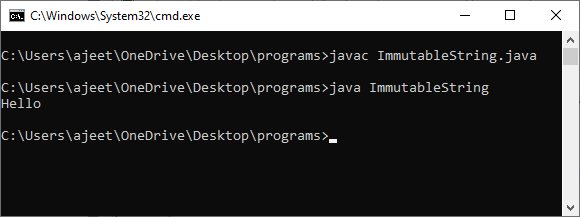
* The String is safe for multithreading because of its immutableness. Different threads can access a single "String instance". It removes the synchronization for thread safety because we make strings thread-safe implicitly.
* Immutability gives the security of loading the correct class by Classloader. For example, suppose we have an instance where we try to load java.sql.Connection class but the changes in the referenced value to the myhacked.Connection class does unwanted things to our database.

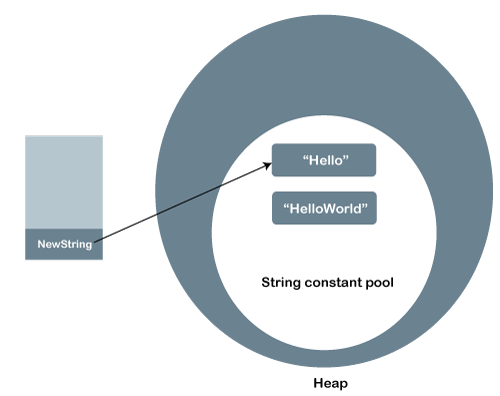
Let's understand the concept of immutable through an example.

**ImmutableString.java**

1. **import** java.util.\*;
2. **class** ImmutableString{
3. **public** **static** **void** main(String args[]){
4. String NewString = "Hello";
5. NewString.concat("World");
6. System.out.println(NewString);
7. }
8. }

**Output:**



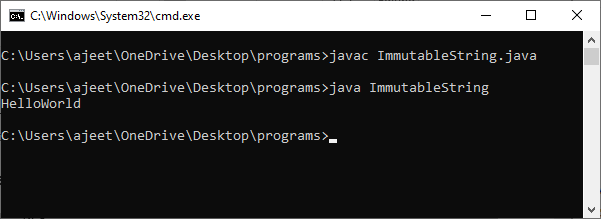
**Description:** We can understand the above example with the help of the following diagram:

In the string constant pool, the **Hello** remains unchanged, and a new string object is created with **HelloWorld**. It shows that the strings are immutable. The reference variable points to the **Hello** not to the **HelloWorld.**

If we want that it refers to the **HelloWorld**, we have to explicitly assign it to that variable. For example:

1. **import** java.util.\*;
2. **class** ImmutableString{
3. **public** **static** **void** main(String args[]){
4. String NewString = "Hello";
5. NewString = NewString.concat("World");
6. System.out.println(NewString);
7. }
8. }

**Output:**



# Write a Code to check whether one string is a rotation of another

Given a string s1 and a string s2, write a snippet to say whether s2 is a rotation of s1?   
(eg given s1 = ABCD and s2 = CDAB, return true, given s1 = ABCD, and s2 = ACBD , return false)

**Algorithm:** areRotations(str1, str2)

1. Create a temp string and store concatenation of str1 to

str1 in temp.

temp = str1.str1

2. If str2 is a substring of temp then str1 and str2 are

rotations of each other.

Example:

str1 = "ABACD"

str2 = "CDABA"

temp = str1.str1 = "ABACDABACD"

Since str2 is a substring of temp, str1 and str2 are

rotations of each other.

## Solution:

#### Approach #1: Brute Force [Accepted]

**Intuition**

For each rotation of A, let's check if it equals B.

**Algorithm**

More specifically, say we rotate A by s. Then, instead of A[0], A[1], A[2], ..., we have A[s], A[s+1], A[s+2], ...; and we should check that A[s] == B[0], A[s+1] == B[1], A[s+2] == B[2], etc.

Java code:

class Solution {

public boolean rotateString(String A, String B) {

if (A.length() != B.length())

return false;

if (A.length() == 0)

return true;

search:

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

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

if (A.charAt((s+i) % A.length()) != B.charAt(i))

continue search;

}

return true;

}

return false;

}

}

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the length of A. For each rotation s, we check up to N*N* elements in A and B.
* Space Complexity: O(1)*O*(1). We only use pointers to elements of A and B.

#### Approach #2: Simple Check [Accepted]

**Intuition and Algorithm**

All rotations of A are contained in A+A. Thus, we can simply check whether B is a substring of A+A. We also need to check A.length == B.length, otherwise we will fail cases like A = "a", B = "aa".

// C++ program to check if two given strings

// are rotations of each other

# include <bits/stdc++.h>

using namespace std;

/\* Function checks if passed strings (str1

and str2) are rotations of each other \*/

bool areRotations(string str1, string str2)

{

/\* Check if sizes of two strings are same \*/

if (str1.length() != str2.length())

return false;

string temp = str1 + str1;

return (temp.find(str2) != string::npos);

}

/\* Driver program to test areRotations \*/

int main()

{

string str1 = "AACD", str2 = "ACDA";

if (areRotations(str1, str2))

printf("Strings are rotations of each other");

else

printf("Strings are not rotations of each other");

return 0;

}

**Output**

Strings are rotations of each other

*Method 3(Using STL):*

**Algorithm :**

1. If the size of both the strings is not equal, then it can never be possible.

2. Push the original string into a queue **q1**.

3. Push the string to be checked inside another queue **q2**.

 4. Keep popping **q2**‘s and pushing it back into it till the number of such operations are less than the size of the string.

5. If **q2** becomes equal to **q1**at any point during these operations, it is possible. Else not.

#include <bits/stdc++.h>

using namespace std;

bool check\_rotation(string s, string goal)

{

if (s.size() != goal.size())

;

queue<char> q1;

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

q1.push(s[i]);

}

queue<char> q2;

for (int i = 0; i < goal.size(); i++) {

q2.push(goal[i]);

}

int k = goal.size();

while (k--) {

char ch = q2.front();

q2.pop();

q2.push(ch);

if (q2 == q1)

return true;

}

return false;

}

int main()

{

string s1 = "ABCD";

string s2 = "CDAB";

if (check\_rotation(s1, s2))

cout << s2 << " is a rotated form of " << s1

<< endl;

else

cout << s2 << " is not a rotated form of " << s1

<< endl;

string s3 = "ACBD";

if (check\_rotation(s1, s3))

cout << s3 << " is a rotated form of " << s1

<< endl;

else

cout << s3 << " is not a rotated form of " << s1

<< endl;

return 0;

}

**Output**

CDAB is a rotated form of ABCD

ACBD is not a rotated form of ABCD

#### Approach #4: KMP (Knuth-Morris-Pratt) [Accepted]

**Intuition**

As before, we want to find whether B exists in A+A. The KMP algorithm is a textbook algorithm that does string matching in linear time, which is faster than brute force.

**Algorithm**

The algorithm is broken up into two steps, building the shifts table (or failure table), and using it to find whether a match exists.

The shift table tells us about the largest prefix of B that ends here. More specifically, B[:shifts[i+1]] == B[i - shifts[i+1] : i] is the largest possible prefix of B ending before B[i].

To build the shift table, we use a dynamic programming approach, where all previously calculated values of shifts are correct. Then, left will be the end of the candidate prefix of B, and right will be the end of the candidate section that should match the prefix B[0], B[1], ..., B[left]. Call positions (left, right) "matching" if the prefix ending at B[left] matches the same length string ending at B[right]. The invariant in our loop will be that (left - 1, right - 1) is matching by the end of each for-block.

In a new for-block, if (left, right) is matching (ie. (left - 1, right - 1) is matching from before, plus B[left] == B[right]), then we know the shift (right - left) is the same number as before. Otherwise, when (left, right) is not matching, we need to find a shorter prefix.

Our strategy is to find a matching of (left2, right) where left2 < left, by finding matchings (left2 - 1, right - 1) plus checking B[left2] == B[right]. Since (left - 1, right - 1) is a matching, by transitivity we want to find matchings (left2 - 1, left - 1). The largest such left2 is left2 = left - shifts[left]. We repeatedly check these left2's in greedy order from largest to smallest.

To find a match of B in A+A with such a shift table ready, we employ a similar strategy. We maintain a matching (match\_len - 1, i - 1), where these positions correspond to strings of length match\_len that end at B[match\_len - 1] and (A+A)[i-1] respectively.

Now when trying to find the largest length matching for (A+A) at position i, it must be at most (match\_len - 1) + 1, where the quantity in brackets is the largest length matching to position i-1.

Again, our strategy is to find a matching (match\_len2 - 1, i - 1) plus check that B[match\_len2] == (A+A)[i]. Similar to before, if B[match\_len] != (A+A)[i], then because (match\_len - 1, i - 1) was a matching, by transitivity (match\_len2 - 1, match\_len - 1) must be a matching, of which the largest is found by match\_len2 = match\_len - shifts[match\_len]. We also repeatedly check these match\_len's in order from largest to smallest.

If at any point in this algorithm our match length is N, we've found B in A+A successfully.

Java code:

class Solution {

public boolean rotateString(String A, String B) {

int N = A.length();

if (N != B.length()) return false;

if (N == 0) return true;

//Compute shift table

int[] shifts = new int[N+1];

Arrays.fill(shifts, 1);

int left = -1;

for (int right = 0; right < N; ++right) {

while (left >= 0 && (B.charAt(left) != B.charAt(right)))

left -= shifts[left];

shifts[right + 1] = right - left++;

}

//Find match of B in A+A

int matchLen = 0;

for (char c: (A+A).toCharArray()) {

while (matchLen >= 0 && B.charAt(matchLen) != c)

matchLen -= shifts[matchLen];

if (++matchLen == N) return true;

}

return false;

}

}

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of A.
* Space Complexity: O(N)*O*(*N*), to create the shift table shifts.

# Write a Program to check whether a string is a valid shuffle of two strings or not

In this example, we will check if a string is the valid shuffle of two other strings in Java.

To understand this example, you should have the knowledge of the following [Java programming](https://www.programiz.com/java-programming) topics:

* [Java Strings](https://www.programiz.com/java-programming/string)
* [Java while and do...while Loop](https://www.programiz.com/java-programming/do-while-loop)

## Solution:

import java.util.Arrays;

class Test {

// length of result string should be equal to sum of two strings

static boolean checkLength(String first, String second, String result) {

if (first.length() + second.length() != result.length()) {

return false;

}

else {

return true;

}

}

// this method converts the string to char array

// sorts the char array

// convert the char array to string and return it

static String sortString(String str) {

char[] charArray = str.toCharArray();

Arrays.sort(charArray);

// convert char array back to string

str = String.valueOf(charArray);

return str;

}

// this method compares each character of the result with

// individual characters of the first and second string

static boolean shuffleCheck(String first, String second, String result) {

// sort each string to make comparison easier

first = sortString(first);

second = sortString(second);

result = sortString(result);

// variables to track each character of 3 strings

int i = 0, j = 0, k = 0;

// iterate through all characters of result

while (k != result.length()) {

// check if first character of result matches

// with first character of first string

if (i < first.length() && first.charAt(i) == result.charAt(k))

i++;

// check if first character of result matches

// with the first character of second string

else if (j < second.length() && second.charAt(j) == result.charAt(k))

j++;

// if the character doesn't match

else {

return false;

}

// access next character of result

k++;

}

// after accessing all characters of result

// if either first or second has some characters left

if(i < first.length() || j < second.length()) {

return false;

}

return true;

}

public static void main(String[] args) {

String first = "XY";

String second = "12";

String[] results = {"1XY2", "Y1X2", "Y21XX"};

// call the method to check if result string is

// shuffle of the string first and second

for (String result : results) {

if (checkLength(first, second, result) == true && shuffleCheck(first, second, result) == true) {

System.out.println(result + " is a valid shuffle of " + first + " and " + second);

}

else {

System.out.println(result + " is not a valid shuffle of " + first + " and " + second);

}

}

}

}

**Output**

1XY2 is a valid shuffle of XY and 12

Y1X2 is a valid shuffle of XY and 12

Y21XX is not a valid shuffle of XY and 12

In the above example, we have a string array named results. It contains three strings: 1XY2, Y1X2, and Y21XX. We are checking if these three strings are valid shuffle of strings first(XY) and second(12).

Here, we have used 3 methods:

**1. checkLength()** - The number of characters in a shuffled string should be equal to the sum of the character in two strings.

So, this method checks if the length of the shuffled string is same as the sum of the length of the first and second strings.

If the length is not equal, there is no need to call the shuffleCheck() method. Hence, we have used the if statement as

// inside main method

if (checkLength(first, second, result) == true && shuffleCheck(first, second, result) == true)

**2. sortString()** - This method converts the string to char Array and then uses the Arrays.sort() method to sort the array. Finally, returns the sorted string.

Since we are comparing the shuffled string with the other two strings, sorting all three strings will make the comparison more efficient.

**3. shuffleCheck()** - This method compares the individual characters of the shuffled string with the characters of first and second strings

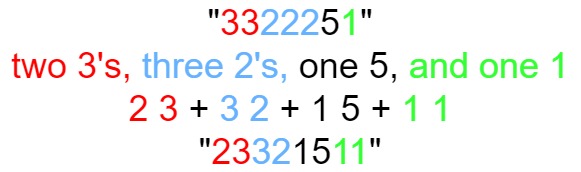
# Count and Say problem

The **count-and-say** sequence is a sequence of digit strings defined by the recursive formula:

* countAndSay(1) = "1"
* countAndSay(n) is the way you would "say" the digit string from countAndSay(n-1), which is then converted into a different digit string.

To determine how you "say" a digit string, split it into the **minimal** number of groups so that each group is a contiguous section all of the **same character.** Then for each group, say the number of characters, then say the character. To convert the saying into a digit string, replace the counts with a number and concatenate every saying.

For example, the saying and conversion for digit string "3322251":



Given a positive integer n, return *the*nth*term of the****count-and-say****sequence*.

**Example 1:**

**Input:** n = 1

**Output:** "1"

**Explanation:** This is the base case.

**Example 2:**

**Input:** n = 4

**Output:** "1211"

**Explanation:**

countAndSay(1) = "1"

countAndSay(2) = say "1" = one 1 = "11"

countAndSay(3) = say "11" = two 1's = "21"

countAndSay(4) = say "21" = one 2 + one 1 = "12" + "11" = "1211"

**Constraints:**

* 1 <= n <= 30

## Solution:

class Solution {

public:

string countAndSay(int n) {

if(n==1) return "1";

if(n==2) return "11";

string s = "11";

for(int i=2;i<n;i++){

string t = "";

s = s+"$";

int count=1;

for(int j=1;j<s.size();j++){

if(s[j]!=s[j-1]){

t = t + to\_string(count) + s[j-1];

count=1;

}

else

count++;

}

s = t;

}

return s;

}

};

**Time Complexity:** O(n^2)

**Space Complexity:** O(1)

# Write a program to find the longest Palindrome in a string.[ Longest palindromic Substring]

Given a string, find the longest substring which is palindrome.

**For example,**

**Input:** Given string :"forgeeksskeegfor",

**Output:** "geeksskeeg"

**Input:** Given string :"Geeks",

**Output:** "ee"

## Solution:

**Method 1:** Brute Force.   
**Approach:** The simple approach is to check each substring whether the substring is a palindrome or not. To do this first, run three nested loops, the outer two loops pick all substrings one by one by fixing the corner characters, the inner loop checks whether the picked substring is palindrome or not.

// A C++ solution for longest palindrome

#include <bits/stdc++.h>

using namespace std;

// Function to print a substring str[low..high]

void printSubStr(string str, int low, int high)

{

for (int i = low; i <= high; ++i)

cout << str[i];

}

// This function prints the

// longest palindrome substring

// It also returns the length

// of the longest palindrome

int longestPalSubstr(string str)

{

// get length of input string

int n = str.size();

// All substrings of length 1

// are palindromes

int maxLength = 1, start = 0;

// Nested loop to mark start and end index

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

for (int j = i; j < str.length(); j++) {

int flag = 1;

// Check palindrome

for (int k = 0; k < (j - i + 1) / 2; k++)

if (str[i + k] != str[j - k])

flag = 0;

// Palindrome

if (flag && (j - i + 1) > maxLength) {

start = i;

maxLength = j - i + 1;

}

}

}

cout << "Longest palindrome substring is: ";

printSubStr(str, start, start + maxLength - 1);

// return length of LPS

return maxLength;

}

// Driver Code

int main()

{

string str = "forgeeksskeegfor";

cout << "\nLength is: "

<< longestPalSubstr(str);

return 0;

}

**Output:**

Longest palindrome subString is: geeksskeeg

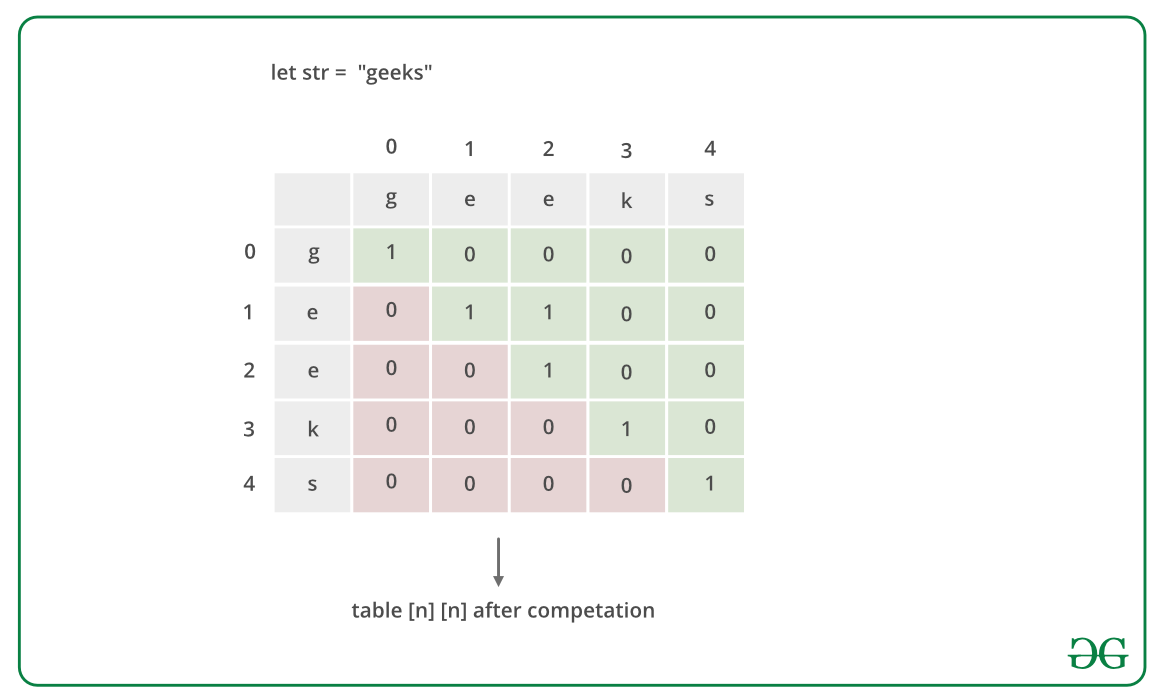
Length is: 10

**Complexity Analysis:**

* **Time complexity:** O(n^3).   
  Three nested loops are needed to find the longest palindromic substring in this approach, so the time complexity is O(n^3).
* **Auxiliary complexity**: O(1).   
  As no extra space is needed.

**Method 2:** Dynamic Programming.   
**Approach:** The time complexity can be reduced by storing results of sub-problems. The idea is similar to [this](https://www.geeksforgeeks.org/archives/19155)post.

1. Maintain a boolean table[n][n] that is filled in bottom up manner.
2. The value of table[i][j] is true, if the substring is palindrome, otherwise false.
3. To calculate table[i][j], check the value of table[i+1][j-1], if the value is true and str[i] is same as str[j], then we make table[i][j] true.
4. Otherwise, the value of table[i][j] is made false.
5. We have to fill table previously for substring of length = 1 and length =2 because   
   as we are finding , if table[i+1][j-1] is true or false , so in case of   
   (i) length == 1 , lets say i=2 , j=2 and i+1,j-1 doesn’t lies between [i , j]   
   (ii) length == 2 ,lets say i=2 , j=3 and i+1,j-1 again doesn’t lies between [i , j].



Below is the implementation of the above approach:

// A C++ dynamic programming

// solution for longest palindrome

#include <bits/stdc++.h>

using namespace std;

// Function to print a substring

// str[low..high]

void printSubStr(

string str, int low, int high)

{

for (int i = low; i <= high; ++i)

cout << str[i];

}

// This function prints the

// longest palindrome substring

// It also returns the length of

// the longest palindrome

int longestPalSubstr(string str)

{

// get length of input string

int n = str.size();

// table[i][j] will be false if substring

// str[i..j] is not palindrome.

// Else table[i][j] will be true

bool table[n][n];

memset(table, 0, sizeof(table));

// All substrings of length 1

// are palindromes

int maxLength = 1;

for (int i = 0; i < n; ++i)

table[i][i] = true;

// check for sub-string of length 2.

int start = 0;

for (int i = 0; i < n - 1; ++i) {

if (str[i] == str[i + 1]) {

table[i][i + 1] = true;

start = i;

maxLength = 2;

}

}

// Check for lengths greater than 2.

// k is length of substring

for (int k = 3; k <= n; ++k) {

// Fix the starting index

for (int i = 0; i < n - k + 1; ++i) {

// Get the ending index of substring from

// starting index i and length k

int j = i + k - 1;

// checking for sub-string from ith index to

// jth index iff str[i+1] to str[j-1] is a

// palindrome

if (table[i + 1][j - 1] && str[i] == str[j]) {

table[i][j] = true;

if (k > maxLength) {

start = i;

maxLength = k;

}

}

}

}

cout << "Longest palindrome substring is: ";

printSubStr(str, start, start + maxLength - 1);

// return length of LPS

return maxLength;

}

// Driver Code

int main()

{

string str = "forgeeksskeegfor";

cout << "\nLength is: "

<< longestPalSubstr(str);

return 0;

}

**Output:**

Longest palindrome substring is: geeksskeeg

Length is: 10

**Complexity Analysis:**

* **Time complexity**: O(n^2).   
  Two nested traversals are needed.
* **Auxiliary Space**: O(n^2).   
  Matrix of size n\*n is needed to store the dp array.

**Approach 3:**

We can find the longest palindrome substring in (n^2) time with O(1) extra space. 

1. The idea is to generate all even length and odd length palindromes and keep track of the longest palindrome seen so far.
2. To generate odd length palindrome, Fix a center and expand in both directions for longer palindromes, i.e. fix i (index) as the center and two indices as i1 = i+1 and i2 = i-1
3. Compare i1 and i2 if equal then decrease i2 and increase i1 and find the maximum length.   
   Use a similar technique to find the even-length palindrome.
4. Take two indices i1 = i and i2 = i-1 and compare characters at i1 and i2 and find the maximum length till all pairs of compared characters are equal and store the maximum length.
5. Print the maximum length.

**// A O(n^2) time and O(1) space program to**

**// find the longest palindromic substring**

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

**using namespace std;**

**// A utility function to print**

**// a substring str[low..high]**

**// This function prints the**

**// longest palindrome substring (LPS)**

**// of str[]. It also returns the**

**// length of the longest palindrome**

**int longestPalSubstr(char\* str)**

**{**

**// The result (length of LPS)**

**int maxLength = 1;**

**int start = 0;**

**int len = strlen(str);**

**int low, high;**

**// One by one consider every**

**// character as center point of**

**// even and length palindromes**

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

**// Find the longest even length palindrome**

**// with center points as i-1 and i.**

**low = i - 1;**

**high = i;**

**while (low >= 0 && high < len**

**&& str[low] == str[high]) {**

**--low;**

**++high;**

**}**

**// Move back to the last possible valid palindrom substring**

**// as that will anyway be the longest from above loop**

**++low; --high;**

**if (str[low] == str[high] && high - low + 1 > maxLength) {**

**start = low;**

**maxLength = high - low + 1;**

**}**

**// Find the longest odd length**

**// palindrome with center point as i**

**low = i - 1;**

**high = i + 1;**

**while (low >= 0 && high < len**

**&& str[low] == str[high]) {**

**--low;**

**++high;**

**}**

**// Move back to the last possible valid palindrom substring**

**// as that will anyway be the longest from above loop**

**++low; --high;**

**if (str[low] == str[high] && high - low + 1 > maxLength) {**

**start = low;**

**maxLength = high - low + 1;**

**}**

**}**

**cout << "Longest palindrome substring is: ";**

**int ans=maxlength;**

**while(ans--)**

**cout<<str[start++];**

**return maxLength;**

**}**

**// Driver program to test above functions**

**int main()**

**{**

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

**cout << "\nLength is: "**

**<< longestPalSubstr(str)**

**<< endl;**

**return 0;**

**}**

**Output:**

Longest palindrome substring is: geeksskeeg

Length is: 10

**Complexity Analysis:**

* **Time complexity:**O(n^2), where n is the length of the input string.   
  A nested traversal of the string is needed. So time complexity is O(n^2).
* **Auxiliary Space:** O(1).   
  No extra space is needed.

# Find Longest Recurring Subsequence in String

Given a string str, find the length of the longest repeating subsequence such that it can be found twice in the given string. The two identified subsequences A and B can use the same ith character from string str if and only if that ith character has different indices in A and B.

**Example 1:**

**Input:**

str = "axxxy"

**Output:** 2

**Explanation:**

The given array with indexes looks like

a x x x y

0 1 2 3 4

The longest subsequence is "xx".

It appears twice as explained below.

**subsequence A**

x x

0 1 <-- index of subsequence A

------

1 2 <-- index of str

**subsequence B**

x x

0 1 <-- index of subsequence B

------

2 3 <-- index of str

We are able to use character 'x'

(at index 2 in str) in both subsequences

as it appears on index 1 in subsequence A

and index 0 in subsequence B.

**Example 2:**

**Input:**

str = "aab"

**Output:** 1

**Explanation:**

The longest reapting subsequenece is "a".

**Your Task:**  
You don't need to read or print anything. Your task is to complete the **LongestRepeatingSubsequence()** which takes str as input parameter and returns the length of the longest repeating subsequnece.

**Expected Time Complexity:** O(n2)  
**Expected Space Complexity:** O(n2)

**Constraints:**  
1 <= |str| <= 103

## Solution:

This problem is just the modification of [Longest Common Subsequence problem](https://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/). The idea is to find the**LCS(str, str)where str is the input string with the restriction that when both the characters are same, they shouldn’t be on the same index in the two strings.**

Below is the implementation of the idea.

// C++ program to find the longest repeating

// subsequence

#include <iostream>

#include <string>

using namespace std;

// This function mainly returns LCS(str, str)

// with a condition that same characters at

// same index are not considered.

int findLongestRepeatingSubSeq(string str)

{

int n = str.length();

// Create and initialize DP table

int dp[n+1][n+1];

for (int i=0; i<=n; i++)

for (int j=0; j<=n; j++)

dp[i][j] = 0;

// Fill dp table (similar to LCS loops)

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

{

for (int j=1; j<=n; j++)

{

// If characters match and indexes are

// not same

if (str[i-1] == str[j-1] && i != j)

dp[i][j] = 1 + dp[i-1][j-1];

// If characters do not match

else

dp[i][j] = max(dp[i][j-1], dp[i-1][j]);

}

}

return dp[n][n];

}

// Driver Program

int main()

{

string str = "aabb";

cout << "The length of the largest subsequence that"

" repeats itself is : "

<< findLongestRepeatingSubSeq(str);

return 0;

}

**Output:**

The length of the largest subsequence that repeats itself is : 2

**Another approach:**(Using recursion)

// C++ program to find the longest repeating

// subsequence using recursion

#include <bits/stdc++.h>

using namespace std;

int dp[1000][1000];

// This function mainly returns LCS(str, str)

// with a condition that same characters at

// same index are not considered.

int findLongestRepeatingSubSeq(string X, int m, int n)

{

if(dp[m][n]!=-1)

return dp[m][n];

// return if we have reached the end of either string

if (m == 0 || n == 0)

return dp[m][n] = 0;

// if characters at index m and n matches

// and index is different

if (X[m - 1] == X[n - 1] && m != n)

return dp[m][n] = findLongestRepeatingSubSeq(X,

m - 1, n - 1) + 1;

// else if characters at index m and n don't match

return dp[m][n] = max (findLongestRepeatingSubSeq(X, m, n - 1),

findLongestRepeatingSubSeq(X, m - 1, n));

}

// Longest Repeated Subsequence Problem

int main()

{

string str = "aabb";

int m = str.length();

memset(dp,-1,sizeof(dp));

cout << "The length of the largest subsequence that"

" repeats itself is : "

<< findLongestRepeatingSubSeq(str,m,m);

return 0;

}

**Output:**

The length of the largest subsequence that repeats itself is : 2

**Approach 3:**

To find the length of the Longest Repeating Subsequence  dynamic  programming Top-down Approach:

* Take the input string.
* Perform the Longest common subsequence where s1[i]==s1[j] and i!=j.
* Return the length.

import java.lang.\*;

import java.io.\*;

import java.util.\*;

class GFG

{

static int lrs(StringBuilder s1, int i, int j, int[][] dp)

{

if(i >= s1.length() || j >= s1.length())

{

return 0;

}

if(dp[i][j] != -1)

{

return dp[i][j];

}

if(dp[i][j] == -1)

{

if(s1.charAt(i) == s1.charAt(j) && i != j)

{

dp[i][j] = 1 + lrs(s1, i + 1, j + 1, dp);

}

else

{

dp[i][j] = Math.max(lrs(s1, i, j + 1, dp), lrs(s1, i + 1, j, dp));

}

}

return dp[i][j];

}

// Driver code

public static void main (String[] args)

{

String s1 = "aabb";

StringBuilder input1 = new StringBuilder();

// append a string into StringBuilder input1

input1.append(s1);

// reverse StringBuilder input1

input1.reverse();

int[][] dp = new int[1000][1000];

for(int[] row : dp)

{

Arrays.fill(row, -1);

}

System.out.println("LENGTH OF LONGEST REPEATING SUBSEQUENCE IS :" +

lrs(input1, 0, 0, dp));

}

}

**Output**

LENGTH OF LONGEST REPEATING SUBSEQUENCE IS : 2

# Print all Subsequences of a string.

Given a string, we have to find out all subsequences of it. A String is a subsequence of a given String, that is generated by deleting some character of a given string without changing its order.

Examples:

Input : abc

Output : a, b, c, ab, bc, ac, abc

Input : aaa

Output : a, aa, aaa

## Solution:

**Method 1 (Pick and Don’t Pick Concept)**

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Find all subsequences

void printSubsequence(string input, string output)

{

// Base Case

// if the input is empty print the output string

if (input.empty()) {

cout << output << endl;

return;

}

// output is passed with including

// the Ist character of

// Input string

printSubsequence(input.substr(1), output + input[0]);

// output is passed without

// including the Ist character

// of Input string

printSubsequence(input.substr(1), output);

}

// Driver code

int main()

{

// output is set to null before passing in as a

// parameter

string output = "";

string input = "abcd";

printSubsequence(input, output);

return 0;

}

**Output**

abcd

abc

abd

ab

acd

ac

ad

a

bcd

bc

bd

b

cd

c

d

**Method 2**   
**Explanation :**

Step 1: Iterate over the entire String

Step 2: Iterate from the end of string

in order to generate different substring

add the substring to the list

Step 3: Drop kth character from the substring obtained

from above to generate different subsequence.

Step 4: if the subsequence is not in the list then recur.

Below is the implementation of the approach.

// CPP program to print all subsequence of a

// given string.

#include <bits/stdc++.h>

using namespace std;

// set to store all the subsequences

unordered\_set<string> st;

// Function computes all the subsequence of an string

void subsequence(string str)

{

// Iterate over the entire string

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

// Iterate from the end of the string

// to generate substrings

for (int j = str.length(); j > i; j--) {

string sub\_str = str.substr(i, j);

st.insert(sub\_str);

// Drop kth character in the substring

// and if its not in the set then recur

for (int k = 1; k < sub\_str.length(); k++) {

string sb = sub\_str;

// Drop character from the string

sb.erase(sb.begin() + k);

subsequence(sb);

}

}

}

}

// Driver Code

int main()

{

string s = "aabc";

subsequence(s);

for (auto i : st)

cout << i << " ";

cout << endl;

return 0;

}

**Output**

aab aa aac bc b abc aabc ab ac a c

**Method 3 :**  
One by one fix characters and recursively generates all subsets starting from them. After every recursive call, we remove last character so that the next permutation can be generated.

// CPP program to generate power set in

// lexicographic order.

#include <bits/stdc++.h>

using namespace std;

// str : Stores input string

// n : Length of str.

// curr : Stores current permutation

// index : Index in current permutation, curr

void printSubSeqRec(string str, int n, int index = -1,

string curr = "")

{

// base case

if (index == n)

return;

if (!curr.empty()) {

cout << curr << "\n";

}

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

curr += str[i];

printSubSeqRec(str, n, i, curr);

// backtracking

curr = curr.erase(curr.size() - 1);

}

return;

}

// Generates power set in lexicographic

// order.

void printSubSeq(string str)

{

printSubSeqRec(str, str.size());

}

// Driver code

int main()

{

string str = "cab";

printSubSeq(str);

return 0;

}

**Output**

c

ca

cab

cb

a

ab

b

# Print all the permutations of the given string

Given a string S. The task is to print all permutations of a given string.

**Example 1:**

**Input:** ABC

**Output:**

ABC ACB BAC BCA CAB CBA

**Explanation:**

Given string ABC has permutations in 6

forms as ABC, ACB, BAC, BCA, CAB and CBA .

**Example 2:**

**Input:** ABSG

**Output:**

ABGS ABSG AGBS AGSB ASBG ASGB BAGS

BASG BGAS BGSA BSAG BSGA GABS GASB

GBAS GBSA GSAB GSBA SABG SAGB SBAG

SBGA SGAB SGBA

**Explanation:**

Given string ABSG has 24 permutations.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **find\_permutaion()**which takes the string S as input parameter and returns a vector of string in lexicographical order.

**Expected Time Complexity:**O(n! \* n)

**Expected Space Complexity:**O(n)

**Constraints:**  
1 <= length of string <= 5

## Solution:

A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation.

Below are the permutations of string ABC.   
ABC ACB BAC BCA CBA CAB

Here is a solution that is used as a basis in backtracking.



// C++ program to print all

// permutations with duplicates allowed

#include <bits/stdc++.h>

using namespace std;

// Function to print permutations of string

// This function takes three parameters:

// 1. String

// 2. Starting index of the string

// 3. Ending index of the string.

void permute(string a, int l, int r)

{

// Base case

if (l == r)

cout<<a<<endl;

else

{

// Permutations made

for (int i = l; i <= r; i++)

{

// Swapping done

swap(a[l], a[i]);

// Recursion called

permute(a, l+1, r);

//backtrack

swap(a[l], a[i]);

}

}

}

// Driver Code

int main()

{

string str = "ABC";

int n = str.size();

permute(str, 0, n-1);

return 0;

}

**Output:**

ABC

ACB

BAC

BCA

CBA

CAB

**Algorithm Paradigm:**Backtracking

**Time Complexity:**O(n\*n!) Note that there are n! permutations and it requires O(n) time to print a permutation.

**Auxiliary Space:**O(r – l)

**Note :** The above solution prints duplicate permutations if there are repeating characters in input string. Please see below link for a solution that prints only distinct permutations even if there are duplicates in input.

#include <bits/stdc++.h>

#include <string>

using namespace std;

void permute(string s , string answer)

{

if(s.length() == 0)

{

cout<<answer<<" ";

return;

}

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

{

char ch = s[i];

string left\_substr = s.substr(0,i);

string right\_substr = s.substr(i+1);

string rest = left\_substr + right\_substr;

permute(rest , answer+ch);

}

}

int main()

{

string s;

string answer="";

cout<<"Enter the string : ";

cin>>s;

cout<<"\nAll possible strings are : ";

permute(s , answer);

return 0;

}

**Output:**

Enter the string : abc

All possible strings are : abc acb bac bca cab cba

**Time Complexity:** O(n\*n!) The time complexity is the same as the above approach, i.e. there are n! permutations and it requires O(n) time to print a permutation.

# Split the Binary string into two substring with equal 0’s and 1’s

Given a binary string **str** of length **N**, the task is to find the maximum count of consecutive substrings **str** can be divided into such that all the substrings are balanced i.e. they have equal number of **0s** and **1s**. If it is not possible to split **str** satisfying the conditions then print **-1**.  
**Example:** 

***Input:****str = “0100110101”****Output:****4   
The required substrings are “01”, “0011”, “01” and “01”.****Input:****str = “0111100010”****Output:****3*

***Input:****str = “0000000000”*

***Output:****-1*

## Solution:

**Approach:** Initialize **count = 0** and traverse the string character by character and keep track of the number of **0s** and **1s** so far, whenever the count of **0s** and **1s** become equal increment the count. As in the given question, if it is not possible to split string then **we must not have any sub string having equal number of 0’s and 1’s** on that time **count = 0**then return **-1** else print the value of count after the traversal of the complete string.  
Below is the implementation of the above approach:

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

// Function to return the count

// of maximum substrings str

// can be divided into

int maxSubStr(string str, int n)

{

// To store the count of 0s and 1s

int count0 = 0, count1 = 0;

// To store the count of maximum

// substrings str can be divided into

int cnt = 0;

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

if (str[i] == '0') {

count0++;

}

else {

count1++;

}

if (count0 == count1) {

cnt++;

}

}

// It is not possible to

// split the string

if (cnt == 0) {

return -1;

}

return cnt;

}

// Driver code

int main()

{

string str = "0100110101";

int n = str.length();

cout << maxSubStr(str, n);

return 0;

}

**Time complexity:** O(N) where N is the length of string   
**Space Complexity:** O(1)

# Word Wrap Problem [VERY IMP].

Given an array **nums[]** of size **n**, where **nums[i]** denotes the number of characters in one word. Let **K** be the limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly.  
Assume that the length of each word is smaller than the line width. When line breaks are inserted there is a possibility that extra spaces are present in each line. The extra spaces include spaces put at the end of every line **except the last one**.

You have to **minimize**the following total cost where **total cost**= Sum of cost of all lines, where cost of line is = (Number of extra spaces in the line)2.

**Example 1:**

**Input:** nums = {3,2,2,5}, k = 6

**Output:** 10

**Explanation**: Given a line can have 6

characters,

Line number 1: From word no. 1 to 1

Line number 2: From word no. 2 to 3

Line number 3: From word no. 4 to 4

So total cost = (6-3)2 + (6-2-2-1)2 = 32+12 = 10.

As in the first line word length = 3 thus

extra spaces = 6 - 3 = 3 and in the second line

there are two word of length 2 and there already

1 space between two word thus extra spaces

= 6 - 2 -2 -1 = 1. As mentioned in the problem

description there will be no extra spaces in

the last line. Placing first and second word

in first line and third word on second line

would take a cost of 02 + 42 = 16 (zero spaces

on first line and 6-2 = 4 spaces on second),

which isn't the minimum possible cost.

**Example 2:**

**Input:** nums = {3,2,2}, k = 4

**Output:** 5

**Explanation:** Given a line can have 4

characters,

Line number 1: From word no. 1 to 1

Line number 2: From word no. 2 to 2

Line number 3: From word no. 3 to 3

Same explaination as above total cost

= (4 - 3)2 + (4 - 2)2 = 5**.**

**Your Task:**

You don't need to read or print anyhting. Your task is to complete the function **solveWordWrap()**which takes nums and k as input paramater and returns the minimized total cost.

**Expected Time Complexity:**O(n2)

**Expected Space Complexity:**O(n)

**Constraints:**

1 ≤ n ≤ 500  
1 ≤ nums[i] ≤ 1000  
max(nums[i]) ≤ k ≤ 1500

## Solution:

**Approach:**We have discussed a [Dynamic Programming based solution](https://www.geeksforgeeks.org/dynamic-programming-set-18-word-wrap/) of word wrap problem. The solution discussed used O(n^2) auxiliary space. The auxiliary space used can be reduced to O(n). The idea is to use two 1-D arrays dp[] and ans[], where dp[i] represents minimum cost of the line in which arr[i] is the first word and ans[i] represents index of last word present in line in which word arr[i] is the first word. Let k represents limit on number of characters in each line. Suppose for any line l the first word in that line is at index i in arr[]. The minimum cost of that line is stored in dp[i]. The last word in that line is at index j in arr[], where j can vary from i to n. Iterate over all values of j and keep track of number of characters added so far in line l. If number of characters are less than k then find cost of current line with these number of characters. Compare this cost with minimum cost find so far for this line in dp[i] and update dp[i] and ans[i] accordingly. Repeat above procedure for each value of i, 1 <= i <= n. The starting and ending words of each line will be at index i and index ans[i], where next value of i for line l+1 is ans[i] + 1.

**Below Implementation prints the starting and ending index value of word of each line. To print minimum cost, we can simply print dp[0].**

// C++ program for space optimized

// solution of Word Wrap problem.

#include <bits/stdc++.h>

using namespace std;

// Function to find space optimized

// solution of Word Wrap problem.

void solveWordWrap(int arr[], int n, int k)

{

int i, j;

// Variable to store number of

// characters in given line.

int currlen;

// Variable to store possible

// minimum cost of line.

int cost;

// DP table in which dp[i] represents

// cost of line starting with word

// arr[i].

int dp[n];

// Array in which ans[i] store index

// of last word in line starting with

// word arr[i].

int ans[n];

// If only one word is present then

// only one line is required. Cost

// of last line is zero. Hence cost

// of this line is zero. Ending point

// is also n-1 as single word is

// present.

dp[n - 1] = 0;

ans[n - 1] = n - 1;

// Make each word first word of line

// by iterating over each index in arr.

for (i = n - 2; i >= 0; i--) {

currlen = -1;

dp[i] = INT\_MAX;

// Keep on adding words in current

// line by iterating from starting

// word upto last word in arr.

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

// Update number of characters

// in current line. arr[j] is

// number of characters in

// current word and 1

// represents space character

// between two words.

currlen += (arr[j] + 1);

// If limit of characters

// is violated then no more

// words can be added to

// current line.

if (currlen > k)

break;

// If current word that is

// added to line is last

// word of arr then current

// line is last line. Cost of

// last line is 0. Else cost

// is square of extra spaces

// plus cost of putting line

// breaks in rest of words

// from j+1 to n-1.

if (j == n - 1)

cost = 0;

else

cost = (k - currlen) \* (k - currlen) + dp[j + 1];

// Check if this arrangement gives

// minimum cost for line starting

// with word arr[i].

if (cost < dp[i]) {

dp[i] = cost;

ans[i] = j;

}

}

}

// Print starting index and ending index

// of words present in each line.

i = 0;

while (i < n) {

cout << i + 1 << " " << ans[i] + 1 << " ";

i = ans[i] + 1;

}

}

// Driver function

int main()

{

int arr[] = { 3, 2, 2, 5 };

int n = sizeof(arr) / sizeof(arr[0]);

int M = 6;

solveWordWrap(arr, n, M);

return 0;

}

**Output:** 

1 1 2 3 4 4

**Time Complexity:**O(n^2)   
**Auxiliary Space:**O(n)

# EDIT Distance [Very Imp]

Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’.

1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.

**Examples:**

Input: str1 = "geek", str2 = "gesek"

Output: 1

We can convert str1 into str2 by inserting a 's'.

Input: str1 = "cat", str2 = "cut"

Output: 1

We can convert str1 into str2 by replacing 'a' with 'u'.

Input: str1 = "sunday", str2 = "saturday"

Output: 3

Last three and first characters are same. We basically

need to convert "un" to "atur". This can be done using

below three operations.

Replace 'n' with 'r', insert t, insert a

## Solution:

**What are the subproblems in this case?**   
The idea is process all characters one by one staring from either from left or right sides of both strings.   
Let us traverse from right corner, there are two possibilities for every pair of character being traversed.

**m:** Length of str1 (first string)

**n:** Length of str2 (second string)

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on ‘str1’, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
   1. Insert: Recur for m and n-1
   2. Remove: Recur for m-1 and n
   3. Replace: Recur for m-1 and n-1

Below is implementation of above Naive recursive solution.

// A Naive recursive C++ program to find minimum number

// operations to convert str1 to str2

#include <bits/stdc++.h>

using namespace std;

// Utility function to find minimum of three numbers

int min(int x, int y, int z) { return min(min(x, y), z); }

int editDist(string str1, string str2, int m, int n)

{

// If first string is empty, the only option is to

// insert all characters of second string into first

if (m == 0)

return n;

// If second string is empty, the only option is to

// remove all characters of first string

if (n == 0)

return m;

// If last characters of two strings are same, nothing

// much to do. Ignore last characters and get count for

// remaining strings.

if (str1[m - 1] == str2[n - 1])

return editDist(str1, str2, m - 1, n - 1);

// If last characters are not same, consider all three

// operations on last character of first string,

// recursively compute minimum cost for all three

// operations and take minimum of three values.

return 1

+ min(editDist(str1, str2, m, n - 1), // Insert

editDist(str1, str2, m - 1, n), // Remove

editDist(str1, str2, m - 1,

n - 1) // Replace

);

}

// Driver code

int main()

{

// your code goes here

string str1 = "sunday";

string str2 = "saturday";

cout << editDist(str1, str2, str1.length(),

str2.length());

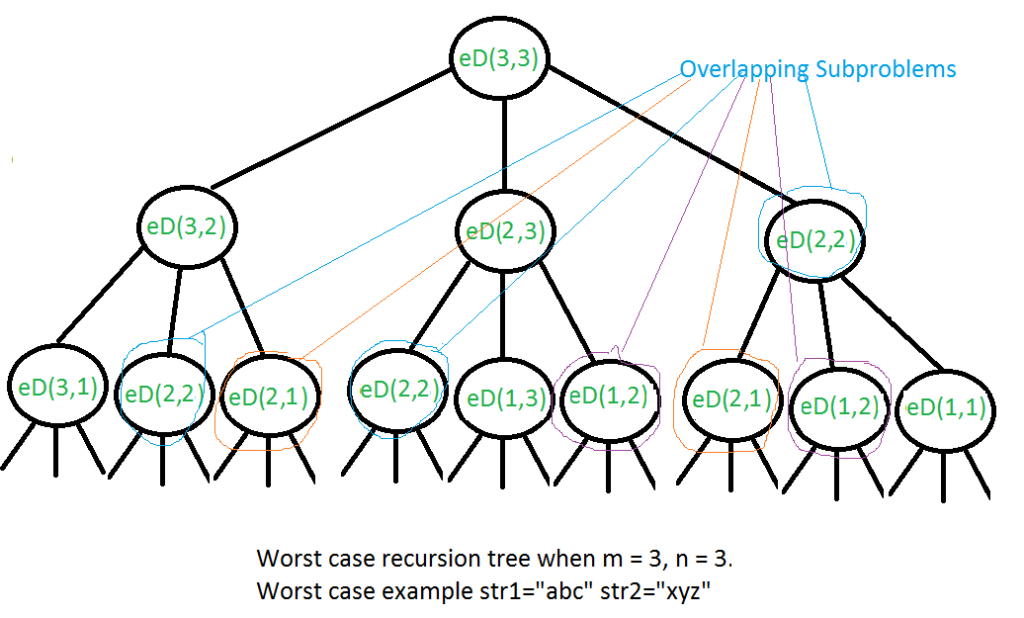
return 0;

}

**Output**

3

The time complexity of above solution is exponential. In worst case, we may end up doing O(3m) operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.



We can see that many subproblems are solved, again and again, for example, eD(2, 2) is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subproblems.

// A Dynamic Programming based C++ program to find minimum

// number operations to convert str1 to str2

#include <bits/stdc++.h>

using namespace std;

// Utility function to find the minimum of three numbers

int min(int x, int y, int z) { return min(min(x, y), z); }

int editDistDP(string str1, string str2, int m, int n)

{

// Create a table to store results of subproblems

int dp[m + 1][n + 1];

// Fill d[][] in bottom up manner

for (int i = 0; i <= m; i++) {

for (int j = 0; j <= n; j++) {

// If first string is empty, only option is to

// insert all characters of second string

if (i == 0)

dp[i][j] = j; // Min. operations = j

// If second string is empty, only option is to

// remove all characters of second string

else if (j == 0)

dp[i][j] = i; // Min. operations = i

// If last characters are same, ignore last char

// and recur for remaining string

else if (str1[i - 1] == str2[j - 1])

dp[i][j] = dp[i - 1][j - 1];

// If the last character is different, consider

// all possibilities and find the minimum

else

dp[i][j]

= 1

+ min(dp[i][j - 1], // Insert

dp[i - 1][j], // Remove

dp[i - 1][j - 1]); // Replace

}

}

return dp[m][n];

}

// Driver code

int main()

{

// your code goes here

string str1 = "sunday";

string str2 = "saturday";

cout << editDistDP(str1, str2, str1.length(),

str2.length());

return 0;

}

**Output**

3

**Time Complexity:** O(m x n)   
**Auxiliary Space:** O(m x n)

**Space Complex Solution**: In the above-given method we require O(m x n) space. This will not be suitable if the length of strings is greater than 2000 as it can only create 2D array of 2000 x 2000. To fill a row in DP array we require only one row the upper row. For example, if we are filling the i = 10 rows in DP array we require only values of 9th row. So we simply create a DP array of 2 x str1 length. This approach reduces the space complexity. Here is the C++ implementation of the above-mentioned problem

// A Space efficient Dynamic Programming

// based C++ program to find minimum

// number operations to convert str1 to str2

#include <bits/stdc++.h>

using namespace std;

void EditDistDP(string str1, string str2)

{

int len1 = str1.length();

int len2 = str2.length();

// Create a DP array to memoize result

// of previous computations

int DP[2][len1 + 1];

// To fill the DP array with 0

memset(DP, 0, sizeof DP);

// Base condition when second string

// is empty then we remove all characters

for (int i = 0; i <= len1; i++)

DP[0][i] = i;

// Start filling the DP

// This loop run for every

// character in second string

for (int i = 1; i <= len2; i++) {

// This loop compares the char from

// second string with first string

// characters

for (int j = 0; j <= len1; j++) {

// if first string is empty then

// we have to perform add character

// operation to get second string

if (j == 0)

DP[i % 2][j] = i;

// if character from both string

// is same then we do not perform any

// operation . here i % 2 is for bound

// the row number.

else if (str1[j - 1] == str2[i - 1]) {

DP[i % 2][j] = DP[(i - 1) % 2][j - 1];

}

// if character from both string is

// not same then we take the minimum

// from three specified operation

else {

DP[i % 2][j] = 1 + min(DP[(i - 1) % 2][j],min(DP[i % 2][j- 1],DP[(i - 1) % 2][j - 1]));

}

}

}

// after complete fill the DP array

// if the len2 is even then we end

// up in the 0th row else we end up

// in the 1th row so we take len2 % 2

// to get row

cout << DP[len2 % 2][len1] << endl;

}

// Driver program

int main()

{

string str1 = "food";

string str2 = "money";

EditDistDP(str1, str2);

return 0;

}

**Output**

4

**Time Complexity:** O(m x n)   
**Auxiliary Space:** O( m )

This is a memoized version of recursion i.e. Top-Down DP:

#include <bits/stdc++.h>

using namespace std;

int minDis(string s1, string s2, int n, int m, vector<vector<int>> &dp){

// If any string is empty,

// return the remaining characters of other string

if(n==0) return m;

if(m==0) return n;

// To check if the recursive tree

// for given n & m has already been executed

if(dp[n][m]!=-1) return dp[n][m];

// If characters are equal, execute

// recursive function for n-1, m-1

if(s1[n-1]==s2[m-1]){

if(dp[n-1][m-1]==-1){

return dp[n][m] = minDis(s1, s2, n-1, m-1, dp);

}

else

return dp[n][m] = dp[n-1][m-1];

}

// If characters are nt equal, we need to

// find the minimum cost out of all 3 operations.

else{

int m1, m2, m3; // temp variables

if(dp[n-1][m]!=-1){

m1 = dp[n-1][m];

}

else{

m1 = minDis(s1, s2, n-1, m, dp);

}

if(dp[n][m-1]!=-1){

m2 = dp[n][m-1];

}

else{

m2 = minDis(s1, s2, n, m-1, dp);

}

if(dp[n-1][m-1]!=-1){

m3 = dp[n-1][m-1];

}

else{

m3 = minDis(s1, s2, n-1, m-1, dp);

}

return dp[n][m] = 1+min(m1, min(m2, m3));

}

}

// Driver program

int main() {

string str1 = "voldemort";

string str2 = "dumbledore";

int n= str1.length(), m = str2.length();

vector<vector<int>> dp(n+1, vector<int>(m+1, -1));

cout<<minDis(str1, str2, n, m, dp);

return 0;

}

**Output**

7

# Find next greater number with same set of digits. [Very Very IMP]

Same as question 15 (Arrays).

# Balanced Parenthesis problem.[Imp]

Given an expression string **x**. Examine whether the pairs and the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp.  
For example, the function should return 'true' for exp = “[()]{}{[()()]()}” and 'false' for exp = “[(])”.

**Example 1:**

**Input**:

{([])}

**Output**:

true

**Explanation**:

{ ( [ ] ) }. Same colored brackets can form

balaced pairs, with 0 number of

unbalanced bracket.

**Example 2:**

**Input**:

()

**Output**:

true

**Explanation**:

(). Same bracket can form balanced pairs,

and here only 1 type of bracket is

present and in balanced way.

**Example 3:**

**Input**:

([]

**Output**:

false

**Explanation**:

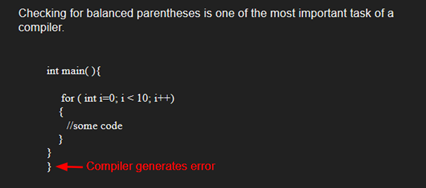
([]. Here square bracket is balanced but

the small bracket is not balanced and

Hence , the output will be unbalanced.

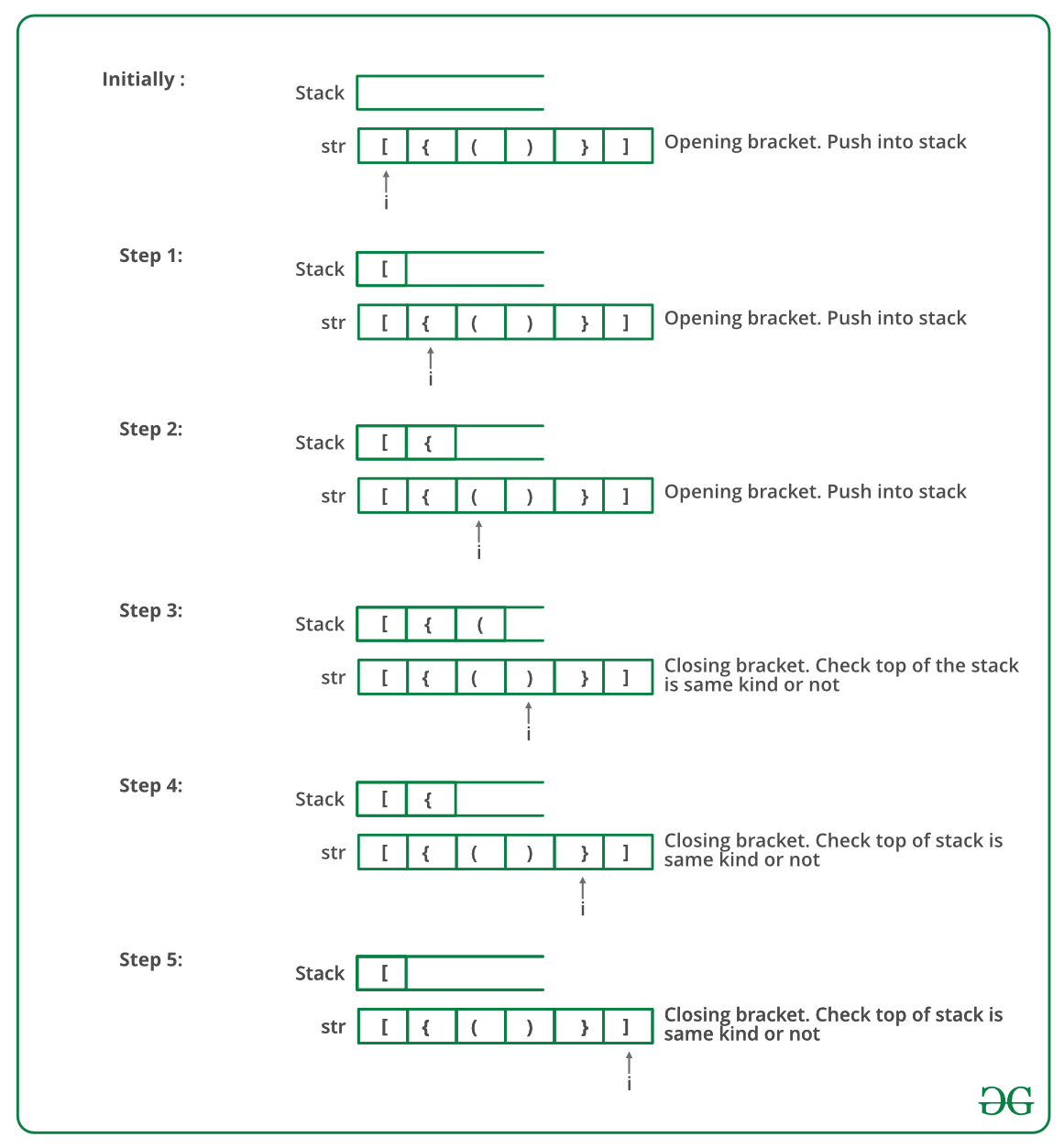
**Your Task:**  
This is a **function**problem. You only need to complete the function **ispar()**that takes a **string**as a **parameter**and returns a boolean value **true**if **brackets**are **balanced**else **returns false**. The **printing**is done **automatically**by the **driver code**.  
  
**Expected Time Complexity**: O(|x|)  
**Expected Auixilliary Space**: O(|x|)  
  
**Constraints:**  
1 ≤ |x| ≤ 32000

## Solution:



**Algorithm:**

* Declare a character [stack](http://www.geeksforgeeks.org/stack-data-structure/) S.
* Now traverse the expression string exp.
  1. If the current character is a starting bracket (**‘(‘ or ‘{‘ or ‘[‘**) then push it to stack.
  2. If the current character is a closing bracket (**‘)’ or ‘}’ or ‘]’**) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.
* After complete traversal, if there is some starting bracket left in stack then “not balanced”

Below image is a dry run of the above approach: 

Below is the implementation of the above approach:

bool ispar(string x)

{

// Your code here

stack<char> st;

int n = x.size();

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

if(x[i]=='{' || x[i]=='[' || x[i]=='(')

st.push(x[i]);

else{

if(st.empty())

return false;

if( (st.top()=='{' && x[i]!='}') || (st.top()=='[' && x[i]!=']') || (st.top()=='(' && x[i]!=')') )

return false;

st.pop();

}

}

if(!st.empty())

return false;

return true;

}

**Output**

Balanced

**Time Complexity:** O(n)   
**Auxiliary Space:** O(n) for stack.

# [Word break Problem[ Very Imp]](https://practice.geeksforgeeks.org/problems/word-break/0)

Given a string **A**and a dictionary of n words **B**, find out if A can be segmented into a space-separated sequence of dictionary words.

**Note:** From the dictionary **B**each word can be taken any number of times and in any order.  
**Example 1:**

**Input:**

n = 12

B = { "i", "like", "sam",

"sung", "samsung", "mobile",

"ice","cream", "icecream",

"man", "go", "mango" }

A = "ilike"

**Output:**

1

**Explanation:**

The string can be segmented as "i like".

**Example 2:**

**Input**:

n = 12

B = { "i", "like", "sam",

"sung", "samsung", "mobile",

"ice","cream", "icecream",

"man", "go", "mango" }

A = "ilikesamsung"

**Output:**

1

**Explanation**:

The string can be segmented as

"i like samsung" or "i like sam sung".

**Your Task:**  
Complete **wordBreak()** function which takes a string and list of strings as a parameter and returns 1 if it is possible to break words, else return 0. You don't need to read any input or print any output, it is done by driver code.

**Expected time complexity:**O(s2)

**Expected auxiliary space:** O(s) , where s = length of string A

**Constraints**:  
1 ≤ N ≤ 12  
1 ≤ s ≤ 1100 , where s = length of string A  
 The length of each word is less than 15.

## Solution:

**Recursive implementation:**   
The idea is simple, we consider each prefix and search it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix).

def wordBreak(wordList, word):

if word == '':

return True

else:

wordLen = len(word)

return any([(word[:i] in wordList) and wordBreak(wordList, word[i:]) for i in range(1, wordLen+1)])

**Output**

If the recursive call for suffix returns true, we return true, otherwise we try next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.  
We strongly recommend to see [**substr**](http://www.cplusplus.com/reference/string/string/substr/)function which is used extensively in following implementations.

// A recursive program to test whether a given

// string can be segmented into space separated

// words in dictionary

#include <iostream>

using namespace std;

/\* A utility function to check whether a word is

present in dictionary or not. An array of strings

is used for dictionary. Using array of strings for

dictionary is definitely not a good idea. We have

used for simplicity of the program\*/

int dictionaryContains(string word)

{

string dictionary[] = {"mobile","samsung","sam","sung",

"man","mango","icecream","and",

"go","i","like","ice","cream"};

int size = sizeof(dictionary)/sizeof(dictionary[0]);

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

if (dictionary[i].compare(word) == 0)

return true;

return false;

}

// returns true if string can be segmented into space

// separated words, otherwise returns false

bool wordBreak(string str)

{

int size = str.size();

// Base case

if (size == 0) return true;

// Try all prefixes of lengths from 1 to size

for (int i=1; i<=size; i++)

{

// The parameter for dictionaryContains is

// str.substr(0, i) which is prefix (of input

// string) of length 'i'. We first check whether

// current prefix is in dictionary. Then we

// recursively check for remaining string

// str.substr(i, size-i) which is suffix of

// length size-i

if (dictionaryContains( str.substr(0, i) ) &&

wordBreak( str.substr(i, size-i) ))

return true;

}

// If we have tried all prefixes and

// none of them worked

return false;

}

// Driver program to test above functions

int main()

{

wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";

wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";

wordBreak("")? cout <<"Yes\n": cout << "No\n";

wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";

wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";

wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";

return 0;

}

**Output**

Yes

Yes

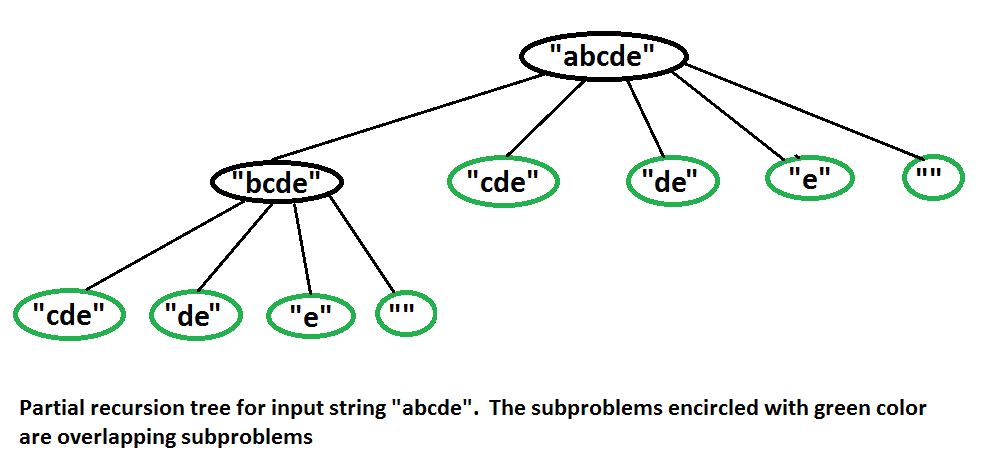
Yes

Yes

Yes

No

**Dynamic Programming**   
Why Dynamic Programming? The above problem exhibits overlapping sub-problems. For example, see the following partial recursion tree for string “abcde” in worst case.



// A Dynamic Programming based program to test whether a given string can

// be segmented into space separated words in dictionary

#include <iostream>

#include <string.h>

using namespace std;

/\* A utility function to check whether a word is present in dictionary or not.

An array of strings is used for dictionary. Using array of strings for

dictionary is definitely not a good idea. We have used for simplicity of

the program\*/

int dictionaryContains(string word)

{

string dictionary[] = {"mobile","samsung","sam","sung","man","mango",

"icecream","and","go","i","like","ice","cream"};

int size = sizeof(dictionary)/sizeof(dictionary[0]);

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

if (dictionary[i].compare(word) == 0)

return true;

return false;

}

// Returns true if string can be segmented into space separated

// words, otherwise returns false

bool wordBreak(string str)

{

int size = str.size();

if (size == 0) return true;

// Create the DP table to store results of subroblems. The value wb[i]

// will be true if str[0..i-1] can be segmented into dictionary words,

// otherwise false.

bool wb[size+1];

memset(wb, 0, sizeof(wb)); // Initialize all values as false.

for (int i=1; i<=size; i++)

{

// if wb[i] is false, then check if current prefix can make it true.

// Current prefix is "str.substr(0, i)"

if (wb[i] == false && dictionaryContains( str.substr(0, i) ))

wb[i] = true;

// wb[i] is true, then check for all substrings starting from

// (i+1)th character and store their results.

if (wb[i] == true)

{

// If we reached the last prefix

if (i == size)

return true;

for (int j = i+1; j <= size; j++)

{

// Update wb[j] if it is false and can be updated

// Note the parameter passed to dictionaryContains() is

// substring starting from index 'i' and length 'j-i'

if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))

wb[j] = true;

// If we reached the last character

if (j == size && wb[j] == true)

return true;

}

}

}

/\* Uncomment these lines to print DP table "wb[]"

for (int i = 1; i <= size; i++)

cout << " " << wb[i]; \*/

// If we have tried all prefixes and none of them worked

return false;

}

// Driver program to test above functions

int main()

{

wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";

wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";

wordBreak("")? cout <<"Yes\n": cout << "No\n";

wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";

wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";

wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";

return 0;

}

**Output**

Yes

Yes

Yes

Yes

Yes

No

**Optimized Dynamic Programming**:   
In this approach, apart from the dp table, we also maintain all the indexes which have matched earlier. Then we will check the substrings from those indexes to the current index. If anyone of that matches then we can divide the string up to that index.  
In this program, we are using some extra space. However, its time complexity is O(n\*s) where s is the length of the largest string in the dictionary and n is the length of the given string.

// A Dynamic Programming based program to test

// whether a given string can be segmented into

// space separated words in dictionary

#include <bits/stdc++.h>

using namespace std;

/\* A utility function to check whether a word

is present in dictionary or not. An array of

strings is used for dictionary. Using array

of strings for dictionary is definitely not

a good idea. We have used for simplicity of

the program\*/

int dictionaryContains(string word)

{

string dictionary[]

= { "mobile", "samsung", "sam", "sung", "man",

"mango", "icecream", "and", "go", "i",

"like", "ice", "cream" };

int size = sizeof(dictionary) / sizeof(dictionary[0]);

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

if (dictionary[i].compare(word) == 0)

return true;

return false;

}

// Returns true if string can be segmented into space

// separated words, otherwise returns false

bool wordBreak(string s)

{

int n = s.size();

if (n == 0)

return true;

// Create the DP table to store results of subroblems.

// The value dp[i] will be true if str[0..i] can be

// segmented into dictionary words, otherwise false.

vector<bool> dp(n + 1, 0); // Initialize all values

// as false.

// matched\_index array represents the indexes for which

// dp[i] is true. Initially only -1 element is present

// in this array.

vector<int> matched\_index;

matched\_index.push\_back(-1);

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

int msize = matched\_index.size();

// Flag value which tells that a substring matches

// with given words or not.

int f = 0;

// Check all the substring from the indexes matched

// earlier. If any of that substring matches than

// make flag value = 1;

for (int j = msize - 1; j >= 0; j--) {

// sb is substring starting from

// matched\_index[j]

// + 1 and of length i - matched\_index[j]

string sb = s.substr(matched\_index[j] + 1,

i - matched\_index[j]);

if (dictionaryContains(sb)) {

f = 1;

break;

}

}

// If substring matches than do dp[i] = 1 and

// push that index in matched\_index array.

if (f == 1) {

dp[i] = 1;

matched\_index.push\_back(i);

}

}

return dp[n - 1];

}

// Driver code

int main()

{

wordBreak("ilikesamsung") ? cout << "Yes\n"

: cout << "No\n";

wordBreak("iiiiiiii") ? cout << "Yes\n"

: cout << "No\n";

wordBreak("") ? cout << "Yes\n" : cout << "No\n";

wordBreak("ilikelikeimangoiii") ? cout << "Yes\n"

: cout << "No\n";

wordBreak("samsungandmango") ? cout << "Yes\n"

: cout << "No\n";

wordBreak("samsungandmangok") ? cout << "Yes\n"

: cout << "No\n";

return 0;

}

**Output**

Yes

Yes

Yes

Yes

Yes

No

This solution can be further optimized as we don't really need dp table. Below is the C++ implementation of this idea.

// A Dynamic Programming based program to test

// whether a given string can  be segmented into

// space separated words in dictionary

#include <bits/stdc++.h>

using namespace std;

/\* A utility function to check whether a word

  is present in dictionary or not. An array of

  strings is used for dictionary.  Using array

  of strings for dictionary is definitely not

  a good idea. We have used for simplicity of

  the program\*/

int dictionaryContains(string word)

{

   string dictionary[]

       = { "mobile", "samsung",  "sam",  "sung", "man",

           "mango",  "icecream", "and",  "go",   "i",

           "like",   "ice",      "cream" };

   int size = sizeof(dictionary) / sizeof(dictionary[0]);

   for (int i = 0; i < size; i++)

       if (dictionary[i].compare(word) == 0)

           return true;

   return false;

}

// Returns true if string can be segmented into space

// separated words, otherwise returns false

bool wordBreak(string s)

{

   int n = s.size();

   if (n == 0)

       return true;

   // matched\_index array represents the indexes for which

   // dp[i] is true. Initially only -1 element is present

   // in this array.

   vector<int> matched\_index;

   matched\_index.push\_back(-1);

   bool f = false;

   for (int i = 0; i < n; i++) {

       int msize = matched\_index.size();

       // Flag value which tells that a substring matches

       // with given words or not.

       f = false;

       // Check all the substring from the indexes matched

       // earlier. If any of that substring matches than

       // make flag value = 1;

       for (int j = msize - 1; j >= 0; j--) {

           // sb is substring starting from

           // matched\_index[j]

           // + 1  and of length i - matched\_index[j]

           string sb = s.substr(matched\_index[j] + 1,

                                i - matched\_index[j]);

           if (dictionaryContains(sb)) {

               f = true;

               break;

           }

       }

       // If substring matches than do dp[i] = 1 and

       // push that index in matched\_index array.

       if (f == true) {

           matched\_index.push\_back(i);

       }

   }

   return f;

}

// Driver code

int main()

{

   wordBreak("ilikesamsung") ? cout << "Yes\n"

                             : cout << "No\n";

   wordBreak("iiiiiiii") ? cout << "Yes\n"

                         : cout << "No\n";

   wordBreak("") ? cout << "Yes\n" : cout << "No\n";

   wordBreak("ilikelikeimangoiii") ? cout << "Yes\n"

                                   : cout << "No\n";

   wordBreak("samsungandmango") ? cout << "Yes\n"

                                : cout << "No\n";

   wordBreak("samsungandmangok") ? cout << "Yes\n"

                                 : cout << "No\n";

   return 0;

  //This code is contributed by Akansha Mittal

}

Output

Yes

Yes

Yes

Yes

Yes

No

[Rabin Karp Algo](https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/)

[Rabin Karp Algo](https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/)

[Rabin Karp Algo](https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/)

[Rabin Karp Algo](https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/)

# Rabin Karp Algo

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

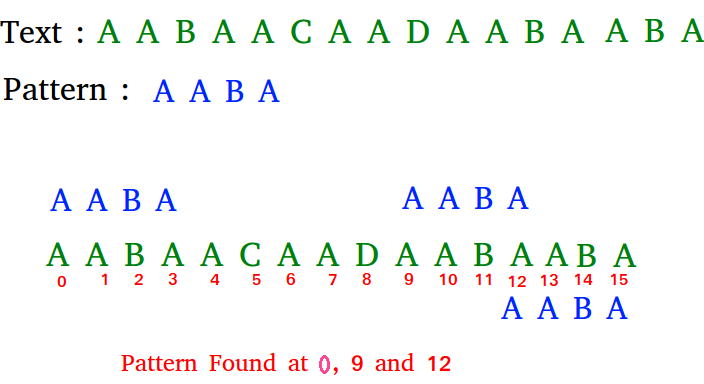
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



## Solution:

**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++ 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()

{

char txt[] = "AABAACAADAABAAABAA";

char pat[] = "AABA";

search(pat, txt);

return 0;

}

**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[] = "AAAAAAAAAAAAAAAAAA";

pat[] = "AAAAA";

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

txt[] = "AAAAAAAAAAAAAAAAAB";

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

The [Naive String Matching](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/) 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 the 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 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 the 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*  
*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)*

**How does the** **above expression work?**

*This is simple mathematics, we compute decimal value of current window from previous window.   
For example pattern length is 3 and string is “23456”   
You compute the value of first window (which is “234”) as 234.   
How how will you compute value of next window “345”? You will do (234 – 2\*100)\*10 + 5 and get 345.*

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

Algorithm given in the CLRS book \*/

#include <bits/stdc++.h>

using namespace std;

// 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 )

{

bool flag = true;

/\* Check for characters one by one \*/

for (j = 0; j < M; j++)

{

if (txt[i+j] != pat[j])

{

flag = false;

break;

}

if(flag)

cout<<i<<" ";

}

// if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]

if (j == M)

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

}

// 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 \*/

int main()

{

char txt[] = "GEEKS FOR GEEKS";

char pat[] = "GEEK";

// A prime number

int q = 101;

// Function Call

search(pat, txt, q);

return 0;

}

**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 same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example pat[] = “AAA” and txt[] = “AAAAAAA”.

# KMP Algo

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

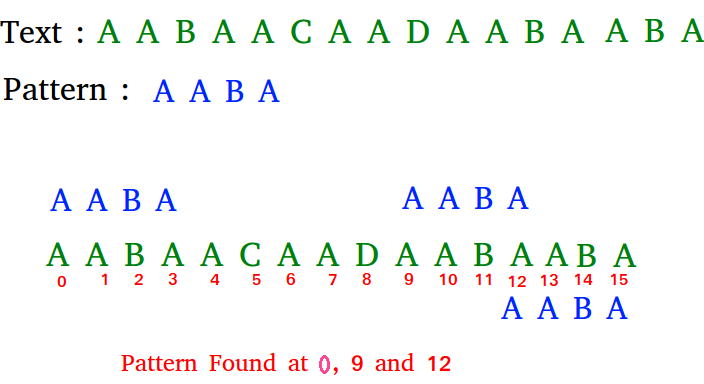
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



## Solution:

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](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/). 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](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/) 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 = "**AAAA**ABAAABA"

pat = "**AAAA**" [Initial position]

We find a match. This is same as [Naive String Matching](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/).

In the next step, we compare next window of **txt** with **pat**.

txt = "**AAAAA**BAAABA"

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](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/), 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…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[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

lps[] = {0, 1, 2, 3}

i = 0, j = 0

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

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

i = 1, j = 1

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

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

i = 2, j = 2

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

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

i = 3, j = 3

txt[] = "**AAAA**ABAAABA"

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[] = "A**AAAA**BAAABA"

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[] = "AA**AAAB**AAABA"

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[] = "AAA**AABA**AABA"

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[] = "AAAA**ABAA**ABA"

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[] = "AAAAA**BAAA**BA"

pat[] = "**AAAA**"

txt[i] and pat[j] do NOT match and j is 0, we do i++.

i = 6, j = 0

txt[] = "AAAAAB**AAABA**"

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++ 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 patttern 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;

}

**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.

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 stop here as we have constructed the whole lps[].

# Convert a Sentence into its equivalent mobile numeric keypad sequence.

Given a sentence in the form of a string, convert it into its equivalent mobile numeric keypad sequence. 



**Examples :** 

Input : GEEKSFORGEEKS

Output : 4333355777733366677743333557777

For obtaining a number, we need to press a

number corresponding to that character for

number of times equal to position of the

character. For example, for character C,

we press number 2 three times and accordingly.

Input : HELLO WORLD

Output : 4433555555666096667775553

## Solution:

Follow the steps given below to convert a sentence into its equivalent mobile numeric keypad sequence. 

* For each character, store the sequence which should be obtained at its respective position in an array, i.e. for Z, store 9999. For Y, store 999. For K, store 55 and so on.
* For each character, subtract ASCII value of ‘A’ and obtain the position in the array pointed   
  by that character and add the sequence stored in that array to a string.
* If the character is a space, store 0
* Print the overall sequence.

Below is the implementation of above method :

// C++ implementation to convert a

// sentence into its equivalent

// mobile numeric keypad sequence

#include <bits/stdc++.h>

using namespace std;

// Function which computes the sequence

string printSequence(string arr[],

string input)

{

string output = "";

// length of input string

int n = input.length();

for (int i=0; i<n; i++)

{

// Checking for space

if (input[i] == ' ')

output = output + "0";

else

{

// Calculating index for each

// character

int position = input[i]-'A';

output = output + arr[position];

}

}

// Output sequence

return output;

}

// Driver function

int main()

{

// storing the sequence in array

string str[] = {"2","22","222",

"3","33","333",

"4","44","444",

"5","55","555",

"6","66","666",

"7","77","777","7777",

"8","88","888",

"9","99","999","9999"

};

string input = "GEEKSFORGEEKS";

cout << printSequence(str, input);

return 0;

}

**Output :** 

4333355777733366677743333557777

**Time complexity :** O(n)

# Minimum number of bracket reversals needed to make an expression balanced.

Given a string **S** consisting of only opening and closing curly brackets**'{'** and**'}',** find out the minimum number of reversals required to convert the string into a balanced expression.  
A reversal means changing **'{'** to **'}'** or vice-versa.

**Example 1:**

**Input:**

S = "}{{}}{{{"

**Output:** 3

**Explanation**: One way to balance is:

"**{**{{}}{**}}**". There is no balanced sequence

that can be formed in lesser reversals.

â€‹**Example 2:**

**Input**:

S = "{{}{{{}{{}}{{"

**Output:** -1

**Explanation**: There's no way we can balance

this sequence of braces.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **countRev()**which takes the string S as input parameter and returns the minimum number of reversals required to balance the bracket sequence. If balancing is not possible, return -1.

**Expected Time Complexity:**O(|S|).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 ≤ |S| ≤ 105

## Solution:

One simple observation is, the string can be balanced only if total number of brackets is even (there must be equal no of ‘{‘ and ‘}’)  
A **Naive Solution** is to consider every bracket and recursively count number of reversals by taking two cases (i) keeping the bracket as it is (ii) reversing the bracket. If we get a balanced expression, we update result if number of steps followed for reaching here is smaller than the minimum so far. Time complexity of this solution is O(2n).

An **Efficient Solution** can solve this problem in O(n) time. The idea is to first remove all balanced part of expression. For example, convert “}**{{}}**{{{” to “}{{{” by removing highlighted part. If we take a closer look, we can notice that, after removing balanced part, we always end up with an expression of the form }}…}{{…{, an expression that contains 0 or more number of closing brackets followed by 0 or more numbers of opening brackets.   
How many minimum reversals are required for an expression of the form “}}..}{{..{” ?. Let m be the total number of closing brackets and n be the number of opening brackets. We need ⌈m/2⌉ + ⌈n/2⌉ reversals. For example }}}}{{ requires 2+1 reversals.  
Below is implementation of above idea:

// C++ program to find minimum number of

// reversals required to balance an expression

#include<bits/stdc++.h>

using namespace std;

// Returns count of minimum reversals for making

// expr balanced. Returns -1 if expr cannot be

// balanced.

int countMinReversals(string expr)

{

int len = expr.length();

// length of expression must be even to make

// it balanced by using reversals.

if (len%2)

return -1;

// After this loop, stack contains unbalanced

// part of expression, i.e., expression of the

// form "}}..}{{..{"

stack<char> s;

for (int i=0; i<len; i++)

{

if (expr[i]=='}' && !s.empty())

{

if (s.top()=='{')

s.pop();

else

s.push(expr[i]);

}

else

s.push(expr[i]);

}

// Length of the reduced expression

// red\_len = (m+n)

int red\_len = s.size();

// count opening brackets at the end of

// stack

int n = 0;

while (!s.empty() && s.top() == '{')

{

s.pop();

n++;

}

// return ceil(m/2) + ceil(n/2) which is

// actually equal to (m+n)/2 + n%2 when

// m+n is even.

return (red\_len/2 + n%2);

}

// Driver program to test above function

int main()

{

string expr = "}}{{";

cout << countMinReversals(expr);

return 0;

}

**Output:**

2

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

Another **efficient solution**solve the problem in O(1) i.e. constant space. Since the expression only contains one type of brackets, the idea is to maintain two variables to keep count of left bracket as well as right bracket as we did in [Length of the longest valid substring](https://www.geeksforgeeks.org/length-of-the-longest-valid-substring/). If the expression has balanced brackets, then we decrement left variable else we increment right variable. Then all we need to return is ceil(left/2) + ceil(right/2).

// C++ program to find minimum number of

// reversals required to balance an expression

#include <bits/stdc++.h>

using namespace std;

// Returns count of minimum reversals for making

// expr balanced. Returns -1 if expr cannot be

// balanced.

int countMinReversals(string expr)

{

int len = expr.length();

// Expressions of odd lengths

// cannot be balanced

if (len % 2 != 0) {

return -1;

}

int left\_brace = 0, right\_brace = 0;

int ans;

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

// If we find a left bracket then we simply

// increment the left bracket

if (expr[i] == '{') {

left\_brace++;

}

// Else if left bracket is 0 then we find

// unbalanced right bracket and increment

// right bracket or if the expression

// is balanced then we decrement left

else {

if (left\_brace == 0) {

right\_brace++;

}

else {

left\_brace--;

}

}

}

ans = ceil(left\_brace / 2.0) + ceil(right\_brace / 2.0);

return ans;

}

// Driver program to test above function

int main()

{

string expr = "}}{{";

cout << countMinReversals(expr);

return 0;

}

**Output**

2

**Time Complexity**: O(n)

**Auxiliary Space**: O(1)

**My approach:**

Instead of maintaining two different variables for left brace and right brace, we can do it using a single temporary variable.

Traverse the array. For each '{' , increment the value of temp by 1 and for each '}', if value of temp >0, then decrement the value of temp by 1 else, increment the value of result as well as temp by 1. At end, add half of the value of temp to the result.

**int countRev (string s)**

**{**

**int temp=0, res=0, n=s.size();**

**if(n%2!=0)**

**return -1;**

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

**if(s[i]=='{')**

**temp++;**

**else{**

**if(temp==0){**

**res++;**

**temp++;**

**}**

**else**

**temp--;**

**}**

**}**

**if(temp>0)**

**res += temp/2;**

**return res;**

**}**

**Time Complexity**: O(n)

**Auxiliary Space**: O(1)

# [Count All Palindromic Subsequence in a given String.](https://practice.geeksforgeeks.org/problems/count-palindromic-subsequences/1)

Find how many palindromic subsequences (need not necessarily be distinct) can be formed in a given string. Note that the empty string is not considered as a palindrome.   
**Examples:**

Input : str = "abcd"

Output : 4

Explanation :- palindromic subsequence are : "a" ,"b", "c" ,"d"

Input : str = "aab"

Output : 4

Explanation :- palindromic subsequence are :"a", "a", "b", "aa"

Input : str = "aaaa"

Output : 15

## Solution:

The above problem can be recursively defined.

Initial Values : i= 0, j= n-1;

CountPS(i,j)

// Every single character of a string is a palindrome

// subsequence

if i == j

return 1 // palindrome of length 1

// If first and last characters are same, then we

// consider it as palindrome subsequence and check

// for the rest subsequence (i+1, j), (i, j-1)

Else if (str[i] == str[j)]

return countPS(i+1, j) + countPS(i, j-1) + 1;

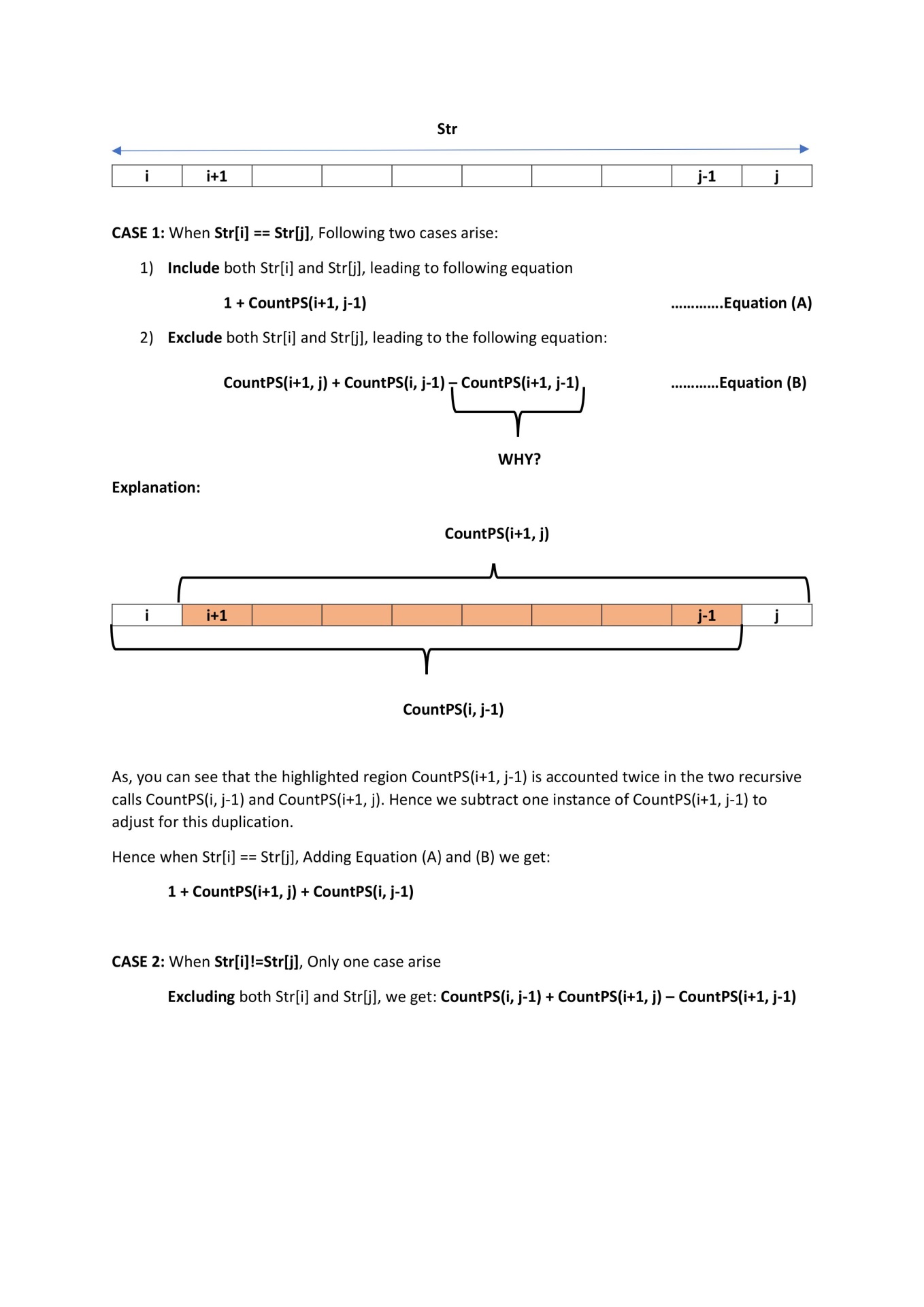
else

// check for rest sub-sequence and remove common

// palindromic subsequences as they are counted

// twice when we do countPS(i+1, j) + countPS(i,j-1)

return countPS(i+1, j) + countPS(i, j-1) - countPS(i+1, j-1)



If we draw recursion tree of above recursive solution, we can observe [overlapping Subproblems](https://www.geeksforgeeks.org/dynamic-programming-set-1/). Since the problem has overlapping subproblems, we can solve it efficiently using Dynamic Programming. Below is Dynamic Programming based solution.

// Counts Palindromic Subsequence in a given String

#include <cstring>

#include <iostream>

using namespace std;

// Function return the total palindromic subsequence

int countPS(string str)

{

int N = str.length();

// create a 2D array to store the count of palindromic

// subsequence

int cps[N + 1][N + 1];

memset(cps, 0, sizeof(cps));

// palindromic subsequence of length 1

for (int i = 0; i < N; i++)

cps[i][i] = 1;

// check subsequence of length L is palindrome or not

for (int L = 2; L <= N; L++) {

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

int k = L + i - 1;

if (str[i] == str[k])

cps[i][k]

= cps[i][k - 1] + cps[i + 1][k] + 1;

else

cps[i][k] = cps[i][k - 1] + cps[i + 1][k]

- cps[i + 1][k - 1];

}

}

// return total palindromic subsequence

return cps[0][N - 1];

}

// Driver program

int main()

{

string str = "abcb";

cout << "Total palindromic subsequence are : "

<< countPS(str) << endl;

return 0;

}

**Output:**

Total palindromic subsequence are : 6

**Time Complexity :** O(N2)

**Another approach:**(Using recursion)

// C++ program to counts Palindromic Subsequence

// in a given String using recursion

#include <bits/stdc++.h>

using namespace std;

int n, dp[1000][1000];

string str = "abcb";

// Function return the total

// palindromic subsequence

int countPS(int i, int j)

{

if (i > j)

return 0;

if (dp[i][j] != -1)

return dp[i][j];

if (i == j)

return dp[i][j] = 1;

else if (str[i] == str[j])

return dp[i][j]

= countPS(i + 1, j) +

countPS(i, j - 1) + 1;

else

return dp[i][j] = countPS(i + 1, j) +

countPS(i, j - 1) -

countPS(i + 1, j - 1);

}

// Driver code

int main()

{

memset(dp, -1, sizeof(dp));

n = str.size();

cout << "Total palindromic subsequence are : "

<< countPS(0, n - 1) << endl;

return 0;

}

**Output:**

Total palindromic subsequence are : 6

**Question**

Given a string str of length N, you have to find number of palindromic subsequence (need not necessarily be distinct) which could be formed from the string str.  
Note: You have to return the answer module 109+7;

**Example 1:**

**Input:**

Str = "abcd"

**Output:**

4

**Explanation:**

palindromic subsequence are : "a" ,"b", "c" ,"d"

**Example 2:**

**Input:**

Str = "aab"

**Output:**

4

**Explanation:**

palindromic subsequence are :"a", "a", "b", "aa"

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **countPs()** which takes a string str as input parameter and returns the number of palindromic subsequence.

**Expected Time Complexity:** O(N\*N)  
**Expected Auxiliary Space:** O(N\*N)

**Constraints:**  
1<=length of string str <=1000

Solution:

long long int countPS(string str)

{

//Your code here

int n = str.size();

if(n<2)

return n;

long long dp[n][n];

memset(dp,0,sizeof(dp));

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

dp[i][i] = 1;

}

for(int k=1;k<n;k++){

int i=0,j=k;

while(j<n){

if(str[i]==str[j]){

dp[i][j] = (dp[i+1][j] + dp[i][j-1] + 1)%((long long)pow(10,9)+7);

}

else{

dp[i][j] = (dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1])%((long long)pow(10,9)+7);

if(dp[i][j]<0)

dp[i][j] += ((long long)pow(10,9)+7);

}

i++;

j=i+k;

}

}

return (dp[0][n-1])%((long long)pow(10,9)+7);

}

# [Count of number of given string in 2D character array](https://www.geeksforgeeks.org/find-count-number-given-string-present-2d-character-array/)

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

**Examples:**

**Input :** a ={

{D,D,D,G,D,D},

{B,B,D,E,B,S},

{B,S,K,E,B,K},

{D,D,D,D,D,E},

{D,D,D,D,D,E},

{D,D,D,D,D,G}

}

str= "GEEKS"

**Output :**2

**Input :** a = {

{B,B,M,B,B,B},

{C,B,A,B,B,B},

{I,B,G,B,B,B},

{G,B,I,B,B,B},

{A,B,C,B,B,B},

{M,C,I,G,A,M}

}

str= "MAGIC"

**Output :**3

## Solution:

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

// of string in a given 2D

// character array.

#include <bits/stdc++.h>

using namespace std;

#define ARRAY\_SIZE(a) (sizeof(a) / sizeof(\*a))

// utility function to search

// complete string from any

// given index of 2d char array

int internalSearch(string needle, int row,

int col, string hay[],

int row\_max, int col\_max, int xx)

{

int found = 0;

if (row >= 0 && row <= row\_max && col >= 0 &&

col <= col\_max && needle[xx] == hay[row][col])

{

char match = needle[xx];

xx += 1;

hay[row][col] = 0;

if (needle[xx] == 0)

{

found = 1;

}

else

{

// through Backtrack searching

// in every directions

found += internalSearch(needle, row,col + 1, hay,row\_max, col\_max,xx);

found += internalSearch(needle, row, col - 1, hay, row\_max, col\_max,xx);

found += internalSearch(needle, row + 1, col, hay, row\_max, col\_max,xx);

found += internalSearch(needle, row - 1, col, hay, row\_max, col\_max,xx);

}

hay[row][col] = match;

}

return found;

}

// Function to search the string in 2d array

int searchString(string needle, int row, int col,

string 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, 0);

}

}

return found;

}

// Driver code

int main()

{

string needle = "MAGIC";

string input[] = { "BBABBM",

"CBMBBA",

"IBABBG",

"GOZBBI",

"ABBBBC",

"MCIGAM" };

string str[ARRAY\_SIZE(input)];

int i;

for (i = 0; i < ARRAY\_SIZE(input); ++i)

{

str[i] = input[i];

}

cout << "count: " << searchString(needle, 0, 0, str,

ARRAY\_SIZE(str),

str[0].size()) << endl;

return 0;

}

**Output**

count: 3

**Time Complexity:** O(row\*col\*4^(length of string)

# Search a Word in a 2D Grid of characters.

Given a 2D grid of characters and a word, find all occurrences of the given word in the grid. A word can be matched in all 8 directions at any point. Word is said to be found in a direction if all characters match in this direction (not in zig-zag form).  
The 8 directions are, Horizontally Left, Horizontally Right, Vertically Up, Vertically Down and 4 Diagonal directions.  
**Example:**

**Input:** grid[][] = {"GEEKSFORGEEKS",

"GEEKSQUIZGEEK",

"IDEQAPRACTICE"};

word = "GEEKS"

**Output:** pattern found at 0, 0

pattern found at 0, 8

pattern found at 1, 0

**Explanation:** 'GEEKS' can be found as prefix of

1st 2 rows and suffix of first row

**Input:** grid[][] = {"GEEKSFORGEEKS",

"GEEKSQUIZGEEK",

"IDEQAPRACTICE"};

word = "EEE"

**Output:** pattern found at 0, 2

pattern found at 0, 10

pattern found at 2, 2

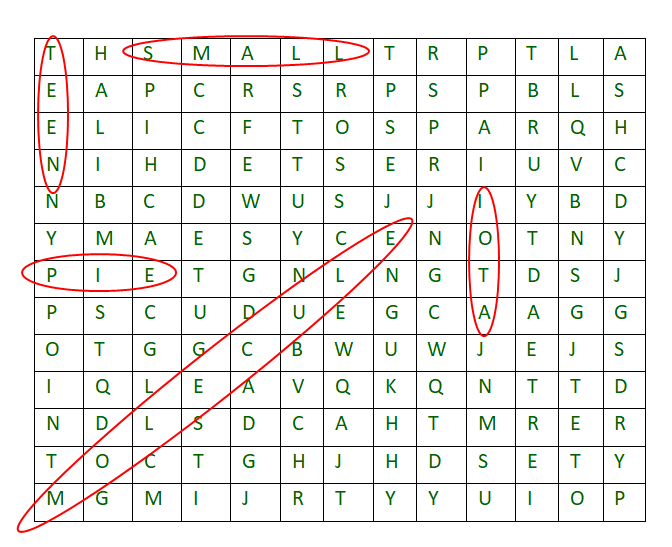
pattern found at 2, 12

**Explanation:** EEE can be found in first row

twice at index 2 and index 10

and in second row at 2 and 12

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



## Solution

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

// C++ programs to search a word in a 2D grid

#include <bits/stdc++.h>

using namespace std;

// 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, int row, int col,

string word, int R, int C)

{

// If first character of word doesn't

// match with given starting point in grid.

if (\*(grid+row\*C+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\*C+cd) != word[k])

break;

// Moving in particular direction

rd += x[dir], cd += y[dir];

}

// If all character matched, then value of k 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, string word,

int R, int C)

{

// 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, R, C))

cout << "pattern found at "

<< row << ", "

<< col << endl;

}

// Driver program

int main()

{

int R = 3, C = 13;

char grid[R][C] = { "GEEKSFORGEEKS",

"GEEKSQUIZGEEK",

"IDEQAPRACTICE" };

patternSearch((char \*)grid, "GEEKS", R, C);

cout << endl;

patternSearch((char \*)grid, "EEE", R, C);

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

**Complexity Analysis:**

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

# Boyer Moore Algorithm for Pattern Searching.

Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file, 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] where n is the length of the text and m is the length of the pattern, 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

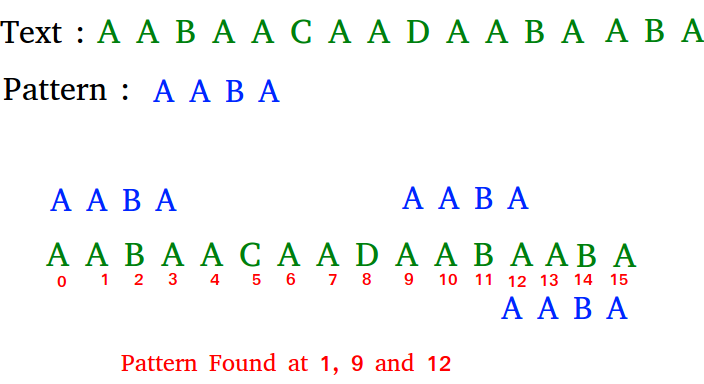
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

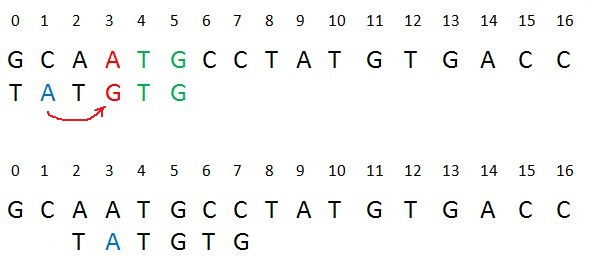
Pattern found at index 12



## Solution:

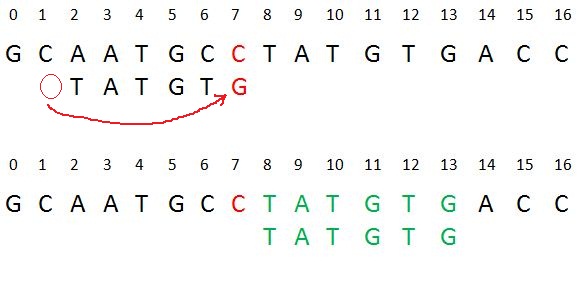
In this post, we will discuss the Boyer Moore pattern searching algorithm. Like [KMP](https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/)and [Finite Automata](https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/)algorithms, Boyer Moore algorithm also preprocesses the pattern.   
Boyer Moore is a combination of the 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](https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/), it slides the pattern over the text one by one. [KMP algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-2-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 processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. **So it uses greatest offset suggested by the two heuristics at every step**.   
Unlike the previous pattern searching algorithms, the **Boyer Moore algorithm starts matching from the last character of the pattern.**  
 **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 the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –   
1) The mismatch becomes a match  
2) Pattern P moves past the mismatched character.  
**Case 1 – Mismatch become match**   
We will lookup the position of the last occurrence of the mismatched character in the pattern, and if the mismatched character exists in the pattern, then we’ll shift the pattern such that it becomes aligned to the mismatched character in the text T. 



*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” does not exist in the pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.  
In the 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 time in the best case.

/\* C++ Program for Bad Character Heuristic of Boyer

Moore String Matching Algorithm \*/

#include <bits/stdc++.h>

using namespace std;

# define NO\_OF\_CHARS 256

// The preprocessing function for Boyer Moore's

// bad character heuristic

void badCharHeuristic( string 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( string txt, string pat)

{

int m = pat.size();

int n = txt.size();

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)

{

cout << "pattern occurs at shift = " << s << endl;

/\* 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 code \*/

int main()

{

string txt= "ABAAABCD";

string pat = "ABC";

search(txt, pat);

return 0;

}

**Output:**

pattern occurs at shift = 4

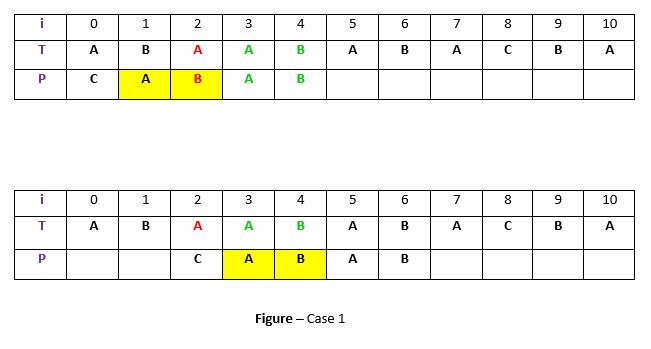
The Bad Character Heuristic may take time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”. The Bad Character Heuristic may take O(n/m) in the best case. The best case occurs when all all the characters of the text and pattern are different.

**Good Suffix heuristic**

Just like bad character heuristic, a preprocessing table is generated for good suffix heuristic.

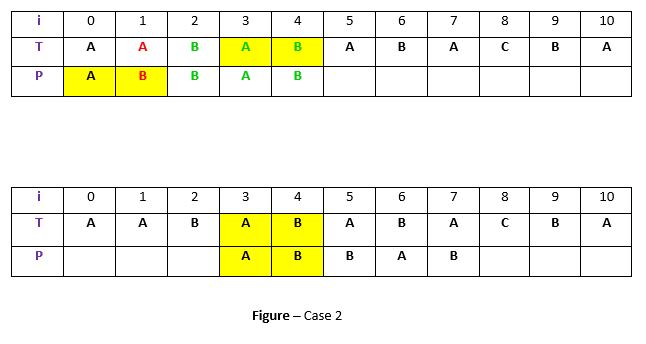
Let **t** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern until :  
1) Another occurrence of t in P matched with t in T.  
2) A prefix of P, which matches with suffix of t  
3) P moves past t

**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-



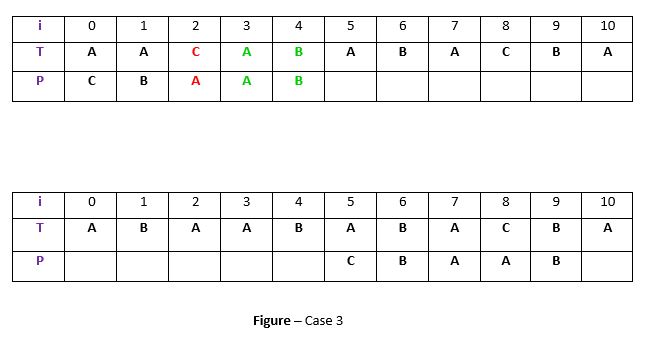
**Explanation:** In the above example, we have got a substring t of text T matched with pattern P (in green) before mismatch at index 2. Now we will search for occurrence of t (“AB”) in P. We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times 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 the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes we can search for some **suffix of t** matching with some **prefix of P** and try to align them by shifting P. For example –



**Explanation:** In above example, we have got t (“BAB”) matched with P (in green) at index 2-4 before mismatch . But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t. We have found prefix “AB” (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t “AB” starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.

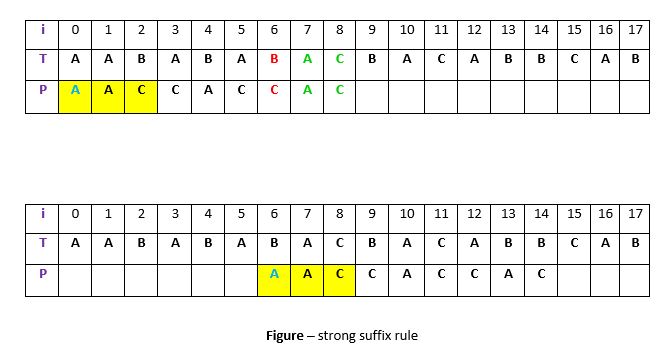
**Case 3: P moves past t**  
If the above two cases are not satisfied, we will shift the pattern past the t. For example –



**Explanation:** If above example, there exist no occurrence of t (“AB”) in P and also there is no prefix in P which matches with the 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 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 –



**Explanation:** In above example, **q = P[7 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 the 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 the 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 = “C”** causes the mismatch. So any occurrence of t preceded by c will again cause 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 **φ** 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 φ(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.

We can understand **bpos[i] = j** using following example –

https://media.geeksforgeeks.org/wp-content/uploads/5-10.jpg

If character **#** Which is at position **i-1** is equivalent to character **?** at position **j-1**, we know that border will be **? + border of suffix at position i** which is starting at position **j** which is equivalent to saying that **border of suffix at i-1 begin at j-1** or **bpos[ i-1 ] = j-1** or in the code –

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 –

* 1. Border width will be smaller than the border starting at position **j** ie. smaller than **x…φ**
  2. Border has to begin with **#** and end with **φ** or could be empty (no border exist).

With above two facts we will continue our search in sub string **x…φ** 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 **shift[j] = j-i** is recorder for **j**. So whenever any mismatch occur at position **j** we will shift the pattern **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 **shift[j]** value from suffix having same border. For example, Consider pattern **P = “addbddcdd”**, when we calculate bpos[ i-1 ] for i = 4 then j = 7 in this case. we will be eventually setting value of shift[ 7 ] = 3. Now if we calculate bpos[ i-1 ] for i = 1 then j = 7 and we will be setting value shift[ 7 ] = 6 again if there is no test 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 **bpos[0]**  
In the following preprocessing algorithm, this value bpos[0] is stored initially in all free entries of array shift. But when the suffix of the pattern becomes shorter than bpos[0], the algorithm continues with the next-wider border of the pattern, i.e. with bpos[j].

Following is the implementation of the 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 the 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 the first character of the pattern

to all indices in array shift having shift[i] = 0 \*/

if(shift[i]==0)

shift[i] = j;

/\* suffix becomes 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 the 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

# Converting Roman Numerals to Decimal

Given a string in roman no format (s)  your task is to convert it to an integer . Various symbols and their values are given below.  
I 1  
V 5  
X 10  
L 50  
C 100  
D 500  
M 1000

**Example 1:**

**Input:**

s = V

**Output:** 5

**Example 2:**

**Input:**

s = III

**Output:** 3

**Your Task:**  
Complete the function**romanToDecimal()** which takes a string as input parameter and returns the equivalent decimal number.

**Expected Time Complexity:**O(|S|), |S| = length of string S.  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1<=roman no range<=3999

## Solution:

**Approach:** A number in Roman Numerals is a string of these symbols written in descending order(e.g. M's first, followed by D's, etc.). However, in a few specific cases, to avoid four characters being repeated in succession(such as IIII or XXXX), **subtractive notation** is often used as follows:

* **I** placed before **V** or **X** indicates one less, so four is **IV** (one less than 5) and 9 is IX (one less than 10).
* **X** placed before **L** or **C** indicates ten less, so forty is **XL** (10 less than 50) and 90 is **XC** (ten less than a hundred).
* **C** placed before **D** or **M** indicates a hundred less, so four hundred is **CD** (a hundred less than five hundred) and nine hundred is **CM** (a hundred less than a thousand).

**Algorithm to convert Roman Numerals to Integer Number:**

1. Split the Roman Numeral string into Roman Symbols (character).
2. Convert each symbol of Roman Numerals into the value it represents.
3. Take symbol one by one from starting from index 0:   
   1. If current value of symbol is greater than or equal to the value of next symbol, then add this value to the running total.
   2. else subtract this value by adding the value of next symbol to the running total.

Following is the implementation of the above algorithm:

// Program to convert Roman

// Numerals to Numbers

#include <bits/stdc++.h>

using namespace std;

// This function returns value

// of a Roman symbol

int value(char r)

{

if (r == 'I')

return 1;

if (r == 'V')

return 5;

if (r == 'X')

return 10;

if (r == 'L')

return 50;

if (r == 'C')

return 100;

if (r == 'D')

return 500;

if (r == 'M')

return 1000;

return -1;

}

// Returns decimal value of

// roman numaral

int romanToDecimal(string& str)

{

// Initialize result

int res = 0;

// Traverse given input

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

{

// Getting value of symbol s[i]

int s1 = value(str[i]);

if (i + 1 < str.length())

{

// Getting value of symbol s[i+1]

int s2 = value(str[i + 1]);

// Comparing both values

if (s1 >= s2)

{

// Value of current symbol

// is greater or equal to

// the next symbol

res = res + s1;

}

else

{

// Value of current symbol is

// less than the next symbol

res = res + s2 - s1;

i++;

}

}

else {

res = res + s1;

}

}

return res;

}

// Driver Code

int main()

{

// Considering inputs given are valid

string str = "MCMIV";

cout << "Integer form of Roman Numeral is "

<< romanToDecimal(str) << endl;

return 0;

}

**Output**

Integer form of Roman Numeral is 1904

**Complexity Analysis:**

* **Time Complexity:** O(n), where n is the length of the string.   
  Only one traversal of the string is required.
* **Space Complexity:** O(1).   
  As no extra space is required.

Another solution –

// Program to convert Roman

// Numerals to Numbers

#include <bits/stdc++.h>

using namespace std;

// This function returns value

// of a Roman symbol

int romanToDecimal(string& str)

{

map<char, int> m;

m.insert({ 'I', 1 });

m.insert({ 'V', 5 });

m.insert({ 'X', 10 });

m.insert({ 'L', 50 });

m.insert({ 'C', 100 });

m.insert({ 'D', 500 });

m.insert({ 'M', 1000 });

int sum = 0;

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

{

/\*If present value is less than next value,

subtract present from next value and add the

resultant to the sum variable.\*/

if (m[str[i]] < m[str[i + 1]])

{

sum+=m[str[i+1]]-m[str[i]];

i++;

continue;

}

sum += m[str[i]];

}

return sum;

}

// Driver Code

int main()

{

// Considering inputs given are valid

string str = "MCMIV";

cout << "Integer form of Roman Numeral is "

<< romanToDecimal(str) << endl;

return 0;

}

**Output**

Integer form of Roman Numeral is 1904

***Time complexity****- O(N)*  
***Auxiliary Space****- O(1)*

**Another Solution:**Shorter code using python

def romanToInt(s):

translations = {

"I": 1,

"V": 5,

"X": 10,

"L": 50,

"C": 100,

"D": 500,

"M": 1000

}

number = 0

s = s.replace("IV", "IIII").replace("IX", "VIIII")

s = s.replace("XL", "XXXX").replace("XC", "LXXXX")

s = s.replace("CD", "CCCC").replace("CM", "DCCCC")

for char in s:

number += translations[char]

print(number)

romanToInt('MCMIV')

**Output**

1904

**Time complexity** - O(N)

**Auxiliary Space** - O(1)

# 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 "".

**Example 1:**

**Input:** strs = ["flower","flow","flight"]

**Output:** "fl"

**Example 2:**

**Input:** strs = ["dog","racecar","car"]

**Output:** ""

**Explanation:** There is no common prefix among the input strings.

**Constraints:**

* 1 <= strs.length <= 200
* 0 <= strs[i].length <= 200
* strs[i] consists of only lower-case English letters.

## Solution:

#### Approach 1: Horizontal scanning

**Intuition**

For a start we will describe a simple way of finding the longest prefix shared by a set of strings LCP(S\_1 \ldots S\_n)*LCP*(*S*1​…*Sn*​). We will use the observation that :

LCP(S\_1 \ldots S\_n) = LCP(LCP(LCP(S\_1, S\_2),S\_3),\ldots S\_n)*LCP*(*S*1​…*Sn*​)=*LCP*(*LCP*(*LCP*(*S*1​,*S*2​),*S*3​),…*Sn*​)

**Algorithm**

To employ this idea, the algorithm iterates through the strings [S\_1 \ldots S\_n][*S*1​…*Sn*​], finding at each iteration i*i* the longest common prefix of strings LCP(S\_1 \ldots S\_i)*LCP*(*S*1​…*Si*​) When LCP(S\_1 \ldots S\_i)*LCP*(*S*1​…*Si*​) is an empty string, the algorithm ends. Otherwise after n*n* iterations, the algorithm returns LCP(S\_1 \ldots S\_n)*LCP*(*S*1​…*Sn*​).

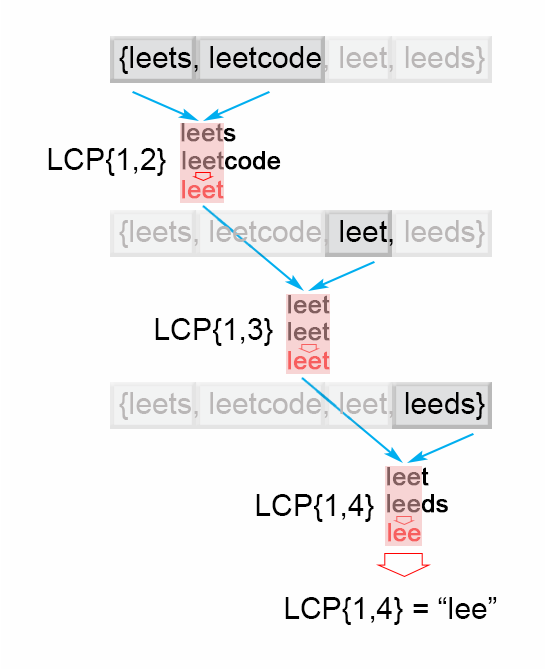


Figure 1. Finding the longest common prefix (Horizontal scanning)

Java code:

public String longestCommonPrefix(String[] strs) {

if (strs.length == 0) return "";

String prefix = strs[0];

for (int i = 1; i < strs.length; i++)

while (strs[i].indexOf(prefix) != 0) {

prefix = prefix.substring(0, prefix.length() - 1);

if (prefix.isEmpty()) return "";

}

return prefix;

}

**Complexity Analysis**

* Time complexity : O(S)*O*(*S*) , where S is the sum of all characters in all strings.

In the worst case all n*n* strings are the same. The algorithm compares the string S1*S*1 with the other strings [S\_2 \ldots S\_n][*S*2​…*Sn*​] There are S*S* character comparisons, where S*S* is the sum of all characters in the input array.

* Space complexity : O(1)*O*(1). We only used constant extra space.

#### Approach 2: Vertical scanning

**Algorithm**

Imagine a very short string is the common prefix at the end of the array. The above approach will still do S*S* comparisons. One way to optimize this case is to do vertical scanning. We compare characters from top to bottom on the same column (same character index of the strings) before moving on to the next column.

Java code:

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0) return "";

for (int i = 0; i < strs[0].length() ; i++){

char c = strs[0].charAt(i);

for (int j = 1; j < strs.length; j ++) {

if (i == strs[j].length() || strs[j].charAt(i) != c)

return strs[0].substring(0, i);

}

}

return strs[0];

}

**Complexity Analysis**

* Time complexity : O(S)*O*(*S*) , where S is the sum of all characters in all strings. In the worst case there will be n*n* equal strings with length m*m* and the algorithm performs S = m \cdot n*S*=*m*⋅*n* character comparisons. Even though the worst case is still the same as [Approach 1](https://leetcode.com/problems/longest-common-prefix/solution/#approach-1-horizontal-scanning), in the best case there are at most n \cdot minLen*n*⋅*minLen* comparisons where minLen*minLen* is the length of the shortest string in the array.
* Space complexity : O(1)*O*(1). We only used constant extra space.

#### Approach 3: Divide and conquer

**Intuition**

The idea of the algorithm comes from the associative property of LCP operation. We notice that : LCP(S\_1 \ldots S\_n) = LCP(LCP(S\_1 \ldots S\_k), LCP (S\_{k+1} \ldots S\_n))*LCP*(*S*1​…*Sn*​)=*LCP*(*LCP*(*S*1​…*Sk*​),*LCP*(*Sk*+1​…*Sn*​)) , where LCP(S\_1 \ldots S\_n)*LCP*(*S*1​…*Sn*​) is the longest common prefix in set of strings [S\_1 \ldots S\_n][*S*1​…*Sn*​] , 1 < k < n1<*k*<*n*

**Algorithm**

To apply the observation above, we use divide and conquer technique, where we split the LCP(S\_i \ldots S\_j)*LCP*(*Si*​…*Sj*​) problem into two subproblems LCP(S\_i \ldots S\_{mid})*LCP*(*Si*​…*Smid*​) and LCP(S\_{mid+1} \ldots S\_j)*LCP*(*Smid*+1​…*Sj*​), where mid is \frac{i + j}{2}2*i*+*j*​. We use their solutions lcpLeft and lcpRight to construct the solution of the main problem LCP(S\_i \ldots S\_j)*LCP*(*Si*​…*Sj*​). To accomplish this we compare one by one the characters of lcpLeft and lcpRight till there is no character match. The found common prefix of lcpLeft and lcpRight is the solution of the LCP(S\_i \ldots S\_j)*LCP*(*Si*​…*Sj*​).

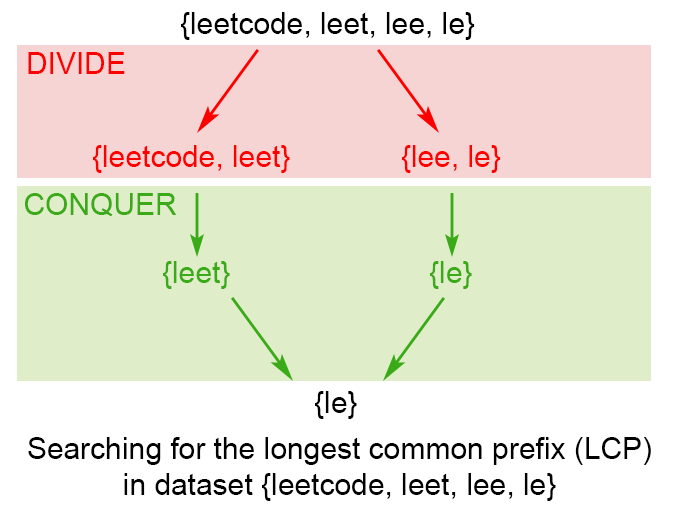


Figure 2. Finding the longest common prefix of strings using divide and conquer technique

Java code:

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0) return "";

return longestCommonPrefix(strs, 0 , strs.length - 1);

}

private String longestCommonPrefix(String[] strs, int l, int r) {

if (l == r) {

return strs[l];

}

else {

int mid = (l + r)/2;

String lcpLeft = longestCommonPrefix(strs, l , mid);

String lcpRight = longestCommonPrefix(strs, mid + 1,r);

return commonPrefix(lcpLeft, lcpRight);

}

}

String commonPrefix(String left,String right) {

int min = Math.min(left.length(), right.length());

for (int i = 0; i < min; i++) {

if ( left.charAt(i) != right.charAt(i) )

return left.substring(0, i);

}

return left.substring(0, min);

}

**Complexity Analysis**

In the worst case we have n*n* equal strings with length m*m*

* Time complexity : O(S)*O*(*S*), where S*S* is the number of all characters in the array, S = m \cdot n*S*=*m*⋅*n* Time complexity is 2 \cdot T\left ( \frac{n}{2} \right ) + O(m)2⋅*T*(2*n*​)+*O*(*m*). Therefore time complexity is O(S)*O*(*S*). In the best case this algorithm performs O(minLen \cdot n)*O*(*minLen*⋅*n*) comparisons, where minLen*minLen* is the shortest string of the array
* Space complexity : O(m \cdot \log n)*O*(*m*⋅log*n*)

There is a memory overhead since we store recursive calls in the execution stack. There are \log nlog*n* recursive calls, each store need m*m* space to store the result, so space complexity is O(m \cdot \log n)*O*(*m*⋅log*n*)

#### Approach 4: Binary search

The idea is to apply binary search method to find the string with maximum value L, which is common prefix of all of the strings. The algorithm searches space is the interval (0 \ldots minLen)(0…*minLen*), where minLen is minimum string length and the maximum possible common prefix. Each time search space is divided in two equal parts, one of them is discarded, because it is sure that it doesn't contain the solution. There are two possible cases:

* S[1...mid] is not a common string. This means that for each j > i S[1..j] is not a common string and we discard the second half of the search space.
* S[1...mid] is common string. This means that for for each i < j S[1..i] is a common string and we discard the first half of the search space, because we try to find longer common prefix.

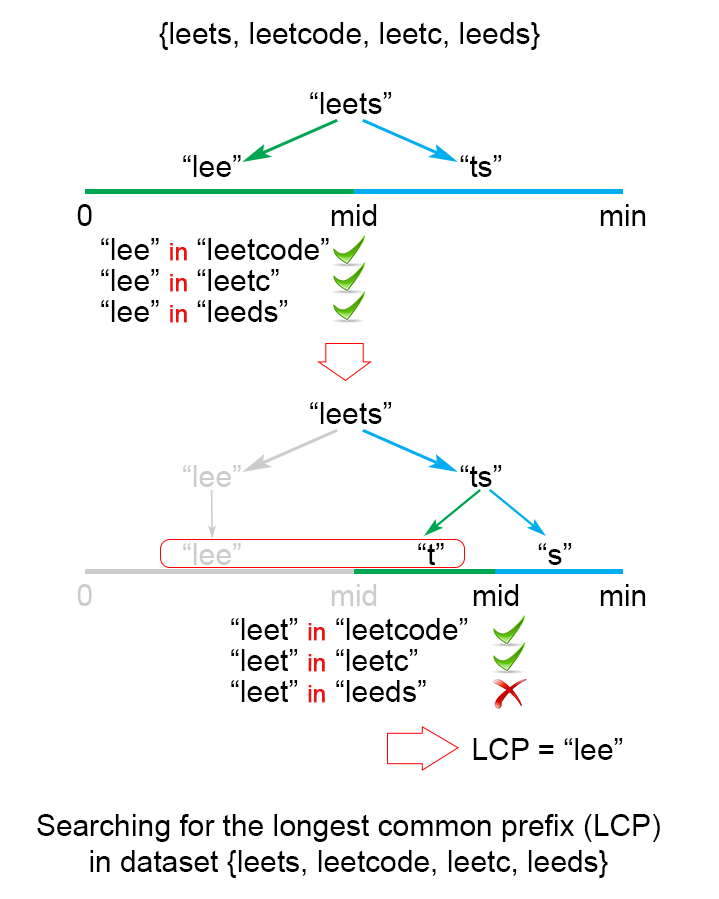


Figure 3. Finding the longest common prefix of strings using binary search technique

Java code:

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0)

return "";

int minLen = Integer.MAX\_VALUE;

for (String str : strs)

minLen = Math.min(minLen, str.length());

int low = 1;

int high = minLen;

while (low <= high) {

int middle = (low + high) / 2;

if (isCommonPrefix(strs, middle))

low = middle + 1;

else

high = middle - 1;

}

return strs[0].substring(0, (low + high) / 2);

}

private boolean isCommonPrefix(String[] strs, int len){

String str1 = strs[0].substring(0,len);

for (int i = 1; i < strs.length; i++)

if (!strs[i].startsWith(str1))

return false;

return true;

}

**Complexity Analysis**

In the worst case we have n*n* equal strings with length m*m*

* Time complexity : O(S \cdot \log m)*O*(*S*⋅log*m*), where S*S* is the sum of all characters in all strings.

The algorithm makes \log mlog*m* iterations, for each of them there are S = m \cdot n*S*=*m*⋅*n* comparisons, which gives in total O(S \cdot \log m)*O*(*S*⋅log*m*) time complexity.

* Space complexity : O(1)*O*(1). We only used constant extra space.

# Number of flips to make binary string alternate

Given a binary string, that is it contains only 0s and 1s. We need to make this string a sequence of alternate characters by flipping some of the bits, our goal is to minimize the number of bits to be flipped.

**Example 1:**

**Input:**

S = "001"

**Output:** 1

**Explanation**: We can flip the 0th bit to 1

to have "101".

â€‹**Example 2:**

**Input**:

S = "0001010111"

**Output:** 2

**Explanation**: We can flip the 1st and 8th bit

to have "0101010101".

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **minFlips()**which takes the string S as input and returns the minimum number of flips required.

**Expected Time Complexity:**O(|S|).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1<=|S|<=105

## Solution:

We can solve this problem by considering all possible results, As we are supposed to get alternate string, there are only 2 possibilities, alternate string starting with 0 and alternate string starting with 1. We will try both cases and choose the string which will require minimum number of flips as our final answer.   
Trying a case requires O(n) time in which we will loop over all characters of given string, if current character is expected character according to alternation then we will do nothing otherwise we will increase flip count by 1. After trying strings starting with 0 and starting with 1, we will choose the string with minimum flip count.   
Total time complexity of solution will be O(n)

// C/C++ program to find minimum number of

// flip to make binary string alternate

#include <bits/stdc++.h>

using namespace std;

// Utility method to flip a character

char flip(char ch)

{

return (ch == '0') ? '1' : '0';

}

// Utility method to get minimum flips when

// alternate string starts with expected char

int getFlipWithStartingCharcter(string str,

char expected)

{

int flipCount = 0;

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

{

// if current character is not expected,

// increase flip count

if (str[i] != expected)

flipCount++;

// flip expected character each time

expected = flip(expected);

}

return flipCount;

}

// method return minimum flip to make binary

// string alternate

int minFlipToMakeStringAlternate(string str)

{

// return minimum of following two

// 1) flips when alternate string starts with 0

// 2) flips when alternate string starts with 1

return min(getFlipWithStartingCharcter(str, '0'),

getFlipWithStartingCharcter(str, '1'));

}

// Driver code to test above method

int main()

{

string str = "0001010111";

cout << minFlipToMakeStringAlternate(str);

return 0;

}

**Output :**

2

**Time Complexity:** O(N)  
**Auxiliary Space:**O(1)

**My code using same approach:**

int minFlips (string S)

{

int swaps01=0,swaps10=0;

for(int i=0;i<S.size();i++){

if(i%2==0){

if(S[i]=='0'){

swaps10++;

}

else{

swaps01++;

}

}

else{

if(S[i]=='0'){

swaps01++;

}

else{

swaps10++;

}

}

}

return min(swaps01,swaps10);

}

# Find the second most repeated string in a sequence.

Given a sequence of strings, the task is to find out the second most repeated (or frequent) string in the given sequence.

**Note:** No two strings are the second most repeated, there will be always a single string.

**Example 1:**

**Input:**

N = 6

arr[] = {aaa, bbb, ccc, bbb, aaa, aaa}

**Output:** bbb

**Explanation**: "bbb" is the second most

occurring string with frequency 2.

**Example 2:**

**Input**:

N = 6

arr[] = {geek, for, geek, for, geek, aaa}

**Output:** for

**Explanation**: "for" is the second most

occurring string with frequency 2.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **secFrequent()**which takes the string array arr[] and its size N as inputs and returns the second most frequent string in the array.

**Expected Time Complexity:**O(N\*max(|Si|).  
**Expected Auxiliary Space:**O(N\*max(|Si|).

**Constraints:**  
1<=N<=103

## Solution:

1. Store all the words in a map with their occurrence with word as key and its occurrence as value.
2. Find the second largest value in the map.
3. Traverse the map again and return the word with occurrence value equals to second max value.

// C++ program to find out the second

// most repeated word

#include <bits/stdc++.h>

using namespace std;

// Function to find the word

string secMostRepeated(vector<string> seq)

{

// Store all the words with its occurrence

unordered\_map<string, int> occ;

for (int i = 0; i < seq.size(); i++)

occ[seq[i]]++;

// find the second largest occurrence

int first\_max = INT\_MIN, sec\_max = INT\_MIN;

for (auto it = occ.begin(); it != occ.end(); it++) {

if (it->second > first\_max) {

sec\_max = first\_max;

first\_max = it->second;

}

else if (it->second > sec\_max &&

it->second != first\_max)

sec\_max = it->second;

}

// Return string with occurrence equals

// to sec\_max

for (auto it = occ.begin(); it != occ.end(); it++)

if (it->second == sec\_max)

return it->first;

}

// Driver program

int main()

{

vector<string> seq = { "ccc", "aaa", "ccc",

"ddd", "aaa", "aaa" };

cout << secMostRepeated(seq);

return 0;

}

**Output:**

ccc

**My code using same approach:**

string secFrequent (string arr[], int n)

{

unordered\_map<string,int> m;

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

m[arr[i]]++;

}

int maxi=0,secondMax=0;

string res, temp = "";

for(auto itr=m.begin();itr!=m.end();itr++){

if(itr->second > maxi){

secondMax = maxi;

res = temp;

maxi = itr->second;

temp = itr->first;

}

if(itr->second < maxi && itr->second > secondMax){

res = itr->first;

secondMax = itr->second;

}

}

return res;

}

# Minimum number of swaps for bracket balancing.

You are given a string S of 2N characters consisting of N ‘[‘ brackets and N ‘]’ brackets. A string is considered balanced if it can be represented in the for S2[S1] where S1 and S2 are balanced strings. We can make an unbalanced string balanced by swapping **adjacent** characters. Calculate the minimum number of swaps necessary to make a string balanced.  
Note - Strings S1 and S2 can be empty.

**Example 1:**

**Input** : []][][

**Output** : 2

**Explanation** :

First swap: Position 3 and 4

[][]][

Second swap: Position 5 and 6

[][][]

**Example 2:**

**Input** : [[][]]

**Output** : 0

**Explanation**:

String is already balanced.

**Your Task:**

You don't need to read input or print anything. Your task is to complete the function **minimumNumberOfSwaps()** which takes the string S and return minimum number of operations required to balance the bracket sequence.

Expected Time Complexity: O(N).  
Expected Auxiliary Space: O(1).

**Constraints:**

1<=|S|<=100000

## Solution:

We can solve this problem by using greedy strategies. If the first X characters form a balanced string, we can neglect these characters and continue on. If we encounter a ‘]’ before the required ‘[‘, then we must start swapping elements to balance the string.

**Naive Approach**   
Initialize sum = 0 where **sum** stores result. Go through the string maintaining a **count** of the number of ‘[‘ brackets encountered. Reduce this count when we encounter a ‘]’ character. If the count hits negative, then we must start balancing the string.   
Let index ‘i’ represent the position we are at. We now move forward to the next ‘[‘ at index j. Increase sum by j – i. Move the ‘[‘ at position j, to position i, and shift all other characters to the right. Set the count back to 0 and continue traversing the string. In the end, ‘sum’ will have the required value.

Time Complexity = O(N^2)   
Extra Space = O(1)

**Optimized approach**   
We can initially go through the string and store the positions of ‘[‘ in a vector say ‘**pos**‘. Initialize ‘p’ to 0. We shall use p to traverse the vector ‘pos’. Similar to the naive approach, we maintain a count of encountered ‘[‘ brackets. When we encounter a ‘[‘ we increase the count and increase ‘p’ by 1. When we encounter a ‘]’ we decrease the count. If the count ever goes negative, this means we must start swapping. The element pos[p] tells us the index of the next ‘[‘. We increase the sum by pos[p] – i, where i is the current index. We can swap the elements in the current index and pos[p] and reset the count to 0 and increment p so that it pos[p] indicates to the next ‘[‘.  
Since we have converted a step that was O(N) in the naive approach, to an O(1) step, our new time complexity reduces.

Time Complexity = O(N)   
Extra Space = O(N)

// C++ program to count swaps required to balance string

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to calculate swaps required

long swapCount(string s)

{

// Keep track of '['

vector<int> pos;

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

if (s[i] == '[')

pos.push\_back(i);

int count = 0; // To count number of encountered '['

int p = 0; // To track position of next '[' in pos

long sum = 0; // To store result

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

{

// Increment count and move p to next position

if (s[i] == '[')

{

++count;

++p;

}

else if (s[i] == ']')

--count;

// We have encountered an unbalanced part of string

if (count < 0)

{

// Increment sum by number of swaps required

// i.e. position of next '[' - current position

sum += pos[p] - i;

swap(s[i], s[pos[p]]);

++p;

// Reset count to 1

count = 1;

}

}

return sum;

}

// Driver code

int main()

{

string s = "[]][][";

cout << swapCount(s) << "\n";

s = "[[][]]";

cout << swapCount(s) << "\n";

return 0;

}

**Output:**

2

0

**Another Method:**   
Time Complexity = O(N)   
Extra Space = O(1)   
We can do without having to store the positions of ‘[‘.

Below is the implementation :

// C++ program to count swaps required

// to balance string

#include <bits/stdc++.h>

using namespace std;

long swapCount(string chars)

{

// Stores total number of Left and

// Right brackets encountered

int countLeft = 0, countRight = 0;

// swap stores the number of swaps

// required imbalance maintains

// the number of imbalance pair

int swap = 0 , imbalance = 0;

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

{

if (chars[i] == '[')

{

// Increment count of Left bracket

countLeft++;

if (imbalance > 0)

{

// swaps count is last swap count + total

// number imbalanced brackets

swap += imbalance;

// imbalance decremented by 1 as it solved

// only one imbalance of Left and Right

imbalance--;

}

}

else if(chars[i] == ']' )

{

// Increment count of Right bracket

countRight++;

// imbalance is reset to current difference

// between Left and Right brackets

imbalance = (countRight - countLeft);

}

}

return swap;

}

// Driver code

int main()

{

string s = "[]][][";

cout << swapCount(s) << endl;

s = "[[][]]";

cout << swapCount(s) << endl;

return 0;

}

**Output:**

2

0

**My code using same approach:**

No need to maintain two separate variable for opening and closing bracket. Just use single temporary variable.

int minimumNumberOfSwaps(string S){

int temp=0, res=0;

for(int i=0;i<S.size();i++){

if(S[i]==']')

temp--;

else{

if(temp>=0)

temp++;

else{

res += abs(temp);

temp++;

}

}

}

return res;

}

Time Complexity = O(N)   
Extra Space = O(1)

# Find the longest common subsequence between two strings.

Given two sequences, find the length of longest subsequence present in both of them. Both the strings are of uppercase.

**Example 1:**

**Input:**

A = 6, B = 6

str1 = ABCDGH

str2 = AEDFHR

**Output:** 3

**Explanation:** LCS for input Sequences

“ABCDGH” and “AEDFHR” is “ADH” of

length 3.

**Example 2:**

**Input:**

A = 3, B = 2

str1 = ABC

str2 = AC

**Output:** 2

**Explanation:** LCS of "ABC" and "AC" is

"AC" of length 2.

**Your Task:**  
Complete the function **lcs()** which takes the length of two strings respectively and two strings as input parameters and returns the length of the longest subsequence present in both of them.

**Expected Time Complexity** : O(|str1|\*|str2|)  
**Expected Auxiliary Space**: O(|str1|\*|str2|)

**Constraints:**  
1<=size(str1),size(str2)<=103

## Solution:

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1,2,..n-1. Recall from theory of permutation and combination that number of combinations with 1 element are nC1. Number of combinations with 2 elements are nC2 and so forth and so on. We know that nC0 + nC1 + nC2 + … nCn = 2n. So a string of length n has 2n-1 different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be O(n \* 2n). Note that it takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

It is a classic computer science problem, the basis of [diff](http://en.wikipedia.org/wiki/Diff)(a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

**Examples:**   
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.   
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

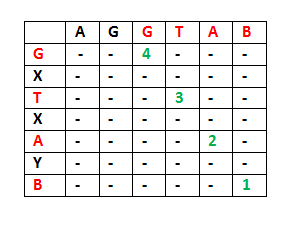
The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

**1) Optimal Substructure:**  
Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then   
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then   
L(X[0..m-1], Y[0..n-1]) = MAX ( L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]) )

Examples:   
1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:   
L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)



2) Consider the input strings “ABCDGH” and “AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:   
L(“ABCDGH”, “AEDFHR”) = MAX ( L(“ABCDG”, “AEDFH**R**”), L(“ABCDG**H**”, “AEDFH”) )  
So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

**2) Overlapping Subproblems:**   
Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

/\* A Naive recursive implementation of LCS problem \*/

#include <bits/stdc++.h>

using namespace std;

int max(int a, int b);

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

int lcs( char \*X, char \*Y, int m, int n )

{

if (m == 0 || n == 0)

return 0;

if (X[m-1] == Y[n-1])

return 1 + lcs(X, Y, m-1, n-1);

else

return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));

}

/\* Utility function to get max of 2 integers \*/

int max(int a, int b)

{

return (a > b)? a : b;

}

/\* Driver code \*/

int main()

{

char X[] = "AGGTAB";

char Y[] = "GXTXAYB";

int m = strlen(X);

int n = strlen(Y);

cout<<"Length of LCS is "<< lcs( X, Y, m, n ) ;

return 0;

}

**Output:**

Length of LCS is 4

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”

lcs("AXYT", "AYZX")

/

lcs("AXY", "AYZX") lcs("AXYT", "AYZ")

/ /

lcs("AX", "AYZX") lcs("AXY", "AYZ") lcs("AXY", "AYZ") lcs("AXYT", "AY")

In the above partial recursion tree, lcs(“AXY”, “AYZ”) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

/\* Dynamic Programming C implementation of LCS problem \*/

#include<bits/stdc++.h>

int max(int a, int b);

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

int lcs( char \*X, char \*Y, int m, int n )

{

int L[m+1][n+1];

int i, j;

/\* Following steps build L[m+1][n+1] in bottom up fashion. Note

that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/

for (i=0; i<=m; i++)

{

for (j=0; j<=n; j++)

{

if (i == 0 || j == 0)

L[i][j] = 0;

else if (X[i-1] == Y[j-1])

L[i][j] = L[i-1][j-1] + 1;

else

L[i][j] = max(L[i-1][j], L[i][j-1]);

}

}

/\* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] \*/

return L[m][n];

}

/\* Utility function to get max of 2 integers \*/

int max(int a, int b)

{

return (a > b)? a : b;

}

/\* Driver program to test above function \*/

int main()

{

char X[] = "AGGTAB";

char Y[] = "GXTXAYB";

int m = strlen(X);

int n = strlen(Y);

printf("Length of LCS is %d", lcs( X, Y, m, n ) );

return 0;

}

**Output:**

Length of LCS is 4

Time Complexity of the above implementation is O(mn) which is much better than the worst-case time complexity of Naive Recursive implementation

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.

# Printing Longest Common Subsequence

Given two sequences, print the longest subsequence present in both of them.

**Examples:**  
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.  
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

We have discussed [Longest Common Subsequence (LCS)](https://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/) problem in a [previous post](https://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/). The function discussed there was mainly to find the length of LCS. To find length of LCS, a 2D table L[][] was constructed. In this post, the function to construct and print LCS is discussed.

Following is detailed algorithm to print the LCS. It uses the same 2D table L[][].

**1)** Construct L[m+1][n+1] using the steps discussed in [previous post](https://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence).

**2)** The value L[m][n] contains length of LCS. Create a character array lcs[] of length equal to the length of lcs plus 1 (one extra to store \0).

**2)** Traverse the 2D array starting from L[m][n]. Do following for every cell L[i][j]  
…..**a)** If characters (in X and Y) corresponding to L[i][j] are same (Or X[i-1] == Y[j-1]), then include this character as part of LCS.  
…..**b)** Else compare values of L[i-1][j] and L[i][j-1] and go in direction of greater value.

The following table (taken from [Wiki](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)) shows steps (highlighted) followed by the above algorithm.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Ø | M | Z | J | A | W | X | U |
| 0 | Ø | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | M | 0 | **1** | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | J | 0 | 1 | 1 | **2** | 2 | 2 | 2 | 2 |
| 4 | Y | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| 5 | A | 0 | 1 | 1 | 2 | **3** | 3 | 3 | 3 |
| 6 | U | 0 | 1 | 1 | 2 | 3 | 3 | 3 | **4** |
| 7 | Z | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 |

Following is the implementation of above approach.

/\* Dynamic Programming implementation of LCS problem \*/

#include<iostream>

#include<cstring>

#include<cstdlib>

using namespace std;

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

void lcs( char \*X, char \*Y, int m, int n )

{

int L[m+1][n+1];

/\* Following steps build L[m+1][n+1] in bottom up fashion. Note

that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/

for (int i=0; i<=m; i++)

{

for (int j=0; j<=n; j++)

{

if (i == 0 || j == 0)

L[i][j] = 0;

else if (X[i-1] == Y[j-1])

L[i][j] = L[i-1][j-1] + 1;

else

L[i][j] = max(L[i-1][j], L[i][j-1]);

}

}

// Following code is used to print LCS

int index = L[m][n];

// Create a character array to store the lcs string

char lcs[index+1];

lcs[index] = '\0'; // Set the terminating character

// Start from the right-most-bottom-most corner and

// one by one store characters in lcs[]

int i = m, j = n;

while (i > 0 && j > 0)

{

// If current character in X[] and Y are same, then

// current character is part of LCS

if (X[i-1] == Y[j-1])

{

lcs[index-1] = X[i-1]; // Put current character in result

i--; j--; index--; // reduce values of i, j and index

}

// If not same, then find the larger of two and

// go in the direction of larger value

else if (L[i-1][j] > L[i][j-1])

i--;

else

j--;

}

// Print the lcs

cout << "LCS of " << X << " and " << Y << " is " << lcs;

}

/\* Driver program to test above function \*/

int main()

{

char X[] = "AGGTAB";

char Y[] = "GXTXAYB";

int m = strlen(X);

int n = strlen(Y);

lcs(X, Y, m, n);

return 0;

}

**Output:**

LCS of AGGTAB and GXTXAYB is GTAB

# Program to generate all possible valid IP addresses from given string.

Given a string containing only digits, restore it by returning all possible valid IP address combinations.  
A valid IP address must be in the form of A.B.C.D, where A, B, C, and D are numbers from 0-255. The numbers cannot be 0 prefixed unless they are 0.

**Input:** 25525511135

**Output:** [“255.255.11.135”, “255.255.111.35”]

**Explanation:**

These are the only valid possible

IP addresses.

**Input:** "25505011535"

**Output:** []

**Explanation:**

We cannot generate a valid IP

address with this string.

## Solution:

First, we will place 3 dots in the given string and then try out all the possible combinations for the 3 dots.   
Corner case for validity:

For string "25011255255"

25.011.255.255 is not valid as 011 is not valid.

25.11.255.255 is not valid either as you are not

allowed to change the string.

250.11.255.255 is valid.

**Approach:** Split the string with ‘ . ‘ and then check for all corner cases. Before entering the loop, check the size of the string. Generate all the possible combinations using looping through the string. If IP is found to be valid then return the IP address, else simply return the empty list.  
Below is the implementation of the above approach:

// C++ program to generate all possible

// valid IP addresses from given string

#include <bits/stdc++.h>

using namespace std;

// Function checks whether IP digits

// are valid or not.

int is\_valid(string ip)

{

// Splitting by "."

vector<string> ips;

string ex = "";

for (int i = 0; i < ip.size(); i++) {

if (ip[i] == '.') {

ips.push\_back(ex);

ex = "";

}

else {

ex = ex + ip[i];

}

}

ips.push\_back(ex);

// Checking for the corner cases

// cout << ip << endl;

for (int i = 0; i < ips.size(); i++) {

// cout << ips[i] <<endl;

if (ips[i].length() > 3

|| stoi(ips[i]) < 0

|| stoi(ips[i]) > 255)

return 0;

if (ips[i].length() > 1

&& stoi(ips[i]) == 0)

return 0;

if (ips[i].length() > 1

&& stoi(ips[i]) != 0

&& ips[i][0] == '0')

return 0;

}

return 1;

}

// Function converts string to IP address

void convert(string ip)

{

int l = ip.length();

// Check for string size

if (l > 12 || l < 4) {

cout << "Not Valid IP Address";

}

string check = ip;

vector<string> ans;

// Generating different combinations.

for (int i = 1; i < l - 2; i++) {

for (int j = i + 1; j < l - 1; j++) {

for (int k = j + 1; k < l; k++) {

check = check.substr(0, k) + "."

+ check.substr(k);

check

= check.substr(0, j) + "."

+ check.substr(j);

check

= check.substr(0, i) + "."

+ check.substr(i);

// cout<< check <<endl;

// Check for the validity of combination

if (is\_valid(check)) {

ans.push\_back(check);

std::cout << check << '\n';

}

check = ip;

}

}

}

}

// Driver code

int main()

{

string A = "25525511135";

string B = "25505011535";

convert(A);

convert(B);

return 0;

}

**Output:**

['255.255.11.135', '255.255.111.35']

**Complexity Analysis:**

* **Time Complexity:** O(n^3), where n is the length of the string   
  Three nested traversal of the string is needed, where n is always less than 12.
* **Auxiliary Space:** O(n).   
  As as extra space is needed.

**Another Efficient Approach (Dynamic Programming):**There is a dp approach exist for this problem and can be solved in time complexity O(n\*4\*3)=O(12n)=O(n) and space complexity O(4n).

**Approach:**We know that there are only 4 parts of the IP. We start iterating from the end of string and goes to the start of string. We create a dp 2D-array of size (4 x N). There can be only 2 values in the dp array i.e. 1(true) or 0(false). dp[0][i] tells if we can create 1 part of IP from the substring starting from i to end of string. Similarly, dp[1][i] tells if we can create 2 parts of IP from the substring starting from i to end of string.

After creating the dp array, we start creating the valid IP addresses. We start from the bottom left corner of the 2D dp array. We only iterate 12 times(worst case) but those also will be the valid IP addresses because we only form valid IP addresses.

// Java Program to generate all possible

// valid IP addresses from given string

import java.util.\*;

class GFG

{

public static ArrayList<String> list;

// Function to restore Ip Addresses

public static ArrayList<String>

restoreIpAddresses(String s)

{

int n = s.length();

list = new ArrayList<>();

if (n < 4 || n > 12)

return list;

// initialize the dp array

int dp[][] = new int[4][n];

for (int i = 0; i < 4; i++)

{

for (int j = n - 1; j >= 0; j--)

{

if (i == 0)

{

// take the substring

String sub = s.substring(j);

if (isValid(sub))

{

dp[i][j] = 1;

}

else if (j < n - 3)

{

break;

}

}

else

{

if (j <= n - i)

{

for (int k = 1;

k <= 3 && j + k <= n;

k++)

{

String temp

= s.substring(j, j + k);

if (isValid(temp))

{

if (j + k < n

&& dp[i - 1][j + k]

== 1)

{

dp[i][j] = 1;

break;

}

}

}

}

}

}

}

if (dp[3][0] == 0)

return list;

// Call function createfromDp

createIpFromDp(dp, 3, 0, s, "");

return list;

}

public static void createIpFromDp(int dp[][],

int r,

int c, String s,

String ans)

{

if (r == 0)

{

ans += s.substring(c);

list.add(ans);

return;

}

for (int i = 1;

i <= 3 && c + i < s.length();

i++)

{

if (isValid(s.substring(c, c + i))

&& dp[r - 1] == 1)

{

createIpFromDp(dp, r - 1, c + i, s,

ans +

s.substring(c, c + i)

+ ".");

}

}

}

private static boolean isValid(String ip)

{

String a[] = ip.split("[.]");

for (String s : a)

{

int i = Integer.parseInt(s);

if (s.length() > 3 || i < 0 || i > 255)

{

return false;

}

if (s.length() > 1 && i == 0)

return false;

if (s.length() > 1 && i != 0

&& s.charAt(0) == '0')

return false;

}

return true;

}

// Driver Code

public static void main(String[] args)

{

// Function call

System.out.println(

restoreIpAddresses("25525511135").toString());

}

}

**Output**

[255.255.11.135, 255.255.111.35]

**Complexity Analysis:**

* **Time Complexity**: O(n), where n is the length of the string. The dp array creation would take O(4\*n\*3) = O(12n) = O(n). Valid IP creation from dp array would take O(n).
* **Auxiliary Space**: O(n). As dp array has extra space of size (4 x n). It means O(4n).

**My recursive approach:**

**class Solution {**

**public:**

**vector<string> ans;**

**vector<string> restoreIpAddresses(string s) {**

**help(s,"",4);**

**return ans;**

**}**

**void help(string s, string temp, int octet){**

**if(octet<1||s.size()==0)**

**return;**

**stringstream ss(s);**

**int num;**

**ss>>num;**

**if(octet==1){**

**if((s[0]=='0' && s.size()>1)|| s.size()==0)**

**return;**

**if(num>=0 && num<=255){**

**ans.push\_back(temp+s);**

**return;**

**}**

**}**

**else{**

**if(s[0]=='0'){**

**temp += "0.";**

**help(s.substr(1),temp,octet-1);**

**return;**

**}**

**if(s.size()>1){**

**string str1 = s.substr(0,1);**

**stringstream s1(str1); int num1;**

**s1>>num1;**

**if(num1>=0 && num1<=255){**

**help(s.substr(1),temp+str1+".",octet-1);**

**}**

**}**

**if(s.size()>2){**

**string str2 = s.substr(0,2);**

**stringstream s2(str2);**

**int num2;**

**s2>>num2;**

**if(num2>=0 && num2<=255){**

**help(s.substr(2),temp+str2+".",octet-1);**

**}**

**}**

**if(s.size()>3){**

**string str3 = s.substr(0,3);**

**stringstream s3(str3);**

**int num3;**

**s3>>num3;**

**if(num3>=0 && num3<=255){**

**help(s.substr(3),temp+str3+".",octet-1);**

**}**

**}**

**}**

**}**

**};**

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

**Space Complexity:** O(1)

# Write a program tofind the smallest window that contains all characters of string itself.

Given a string, find the smallest window length with all distinct characters of the given string. For eg. str = “aabcbcdbca”, then the result would be 4 as of the smallest window will be “dbca” .  
**Examples:**

**Input:** aabcbcdbca

**Output:** dbca

**Explanation:**

Possible substrings= {aabcbcd, abcbcd,

bcdbca, dbca....}

Of the set of possible substrings 'dbca'

is the shortest substring having all the

distinct characters of given string.

**Input:** aaab

**Output:** ab

**Explanation:**

Possible substrings={aaab, aab, ab}

Of the set of possible substrings 'ab'

is the shortest substring having all

the distinct characters of given string.

## Solution:

**Solution:** Above problem states that we have to find the smallest window that contains all the distinct characters of the given string even if the smallest string contains repeating elements.   
For example, in “aabcbcdb”, the smallest string that contains all the characters is “abcbcd”.  
**Method 1:** This is the Brute Force method of solving the problem using HashMap.

* **Approach :** For solving the problem we first have to find out all the distinct characters present in the string. This can be done using a [**HashMap**](http://www.geeksforgeeks.org/java-util-hashmap-in-java/). The next thing is to generate all the possible substrings. This follows by checking whether a substring generated has all the required characters(stored in the hash\_map) or not. If yes, then compare its length with the minimum substring length which follows the above constraints, found till now.  
  [*HashMap*](http://www.geeksforgeeks.org/java-util-hashmap-in-java/)*:* HashMap is a part of Java’s collection since Java 1.2. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. HashMap is known as HashMap because it uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents the same String. A shorter value helps in indexing and faster searches. HashSet also uses HashMap internally. It internally uses a link list to store key-value pairs already explained in HashSet in detail and further articles.
* **Algorithm :**
  1. Store all distinct characters of the given string in a hash\_map.
  2. Take a variable count and initialize it with value 0.
  3. Generate the substrings using two pointers.
  4. **Now check whether generated substring is valid or not-:**
     1. As soon we find that the character of the substring generated has not been encountered before, increment count by **1**.
     2. We can use a visited array of **max\_chars** size to find whether the current character has been encountered before or not.
     3. If count is equal to equal to size of hash\_map the substring generated is valid
     4. If it is a valid substring, compare it with the minimum length substring already generated.
* **Pseudo Code:**

maphash\_map;

for ( i=0 to str.length())

hash\_map[str[i]]++;//finding all distinct characters of string

minimum\_size=INT\_MAX

Distinct\_chars=hash\_map.size()

for(i=0 to str.length())

count=0;

sub\_str="";

visited[256]={0};

for(j=i to n)

sub\_str+=str[j]

if(visited[str[j]]==0)

count++

visited[str[j]]=1;

if(count==Distinct\_chars)

end loop

if(sub\_str.length()<minimum\_size&&

count==Distinct\_chars)

ans=sub\_str;

return ans

* **Implementation:**

// C++ program to find the smallest

// window containing all characters

// of a pattern.

#include <bits/stdc++.h>

using namespace std;

const int MAX\_CHARS = 256;

// Function to find smallest window containing

// all distinct characters

string findSubString(string str)

{

int n = str.length();

// Count all distinct characters.

int dist\_count = 0;

unordered\_map<int, int> hash\_map;

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

hash\_map[str[i]]++;

}

dist\_count = hash\_map.size();

int size = INT\_MAX;

string res;

// Now follow the algorithm discussed in below

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

int count = 0;

int visited[256] = { 0 };

string sub\_str = "";

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

if (visited[str[j]] == 0) {

count++;

visited[str[j]] = 1;

}

sub\_str += str[j];

if (count == dist\_count)

break;

}

if (sub\_str.length() < size && count == dist\_count)

{

res = sub\_str;

size=res.length();

}

}

return res;

}

// Driver Code

int main()

{

string str = "aabcbcdbca";

cout << "Smallest window containing all distinct"

" characters is: "

<< findSubString(str);

return 0;

}

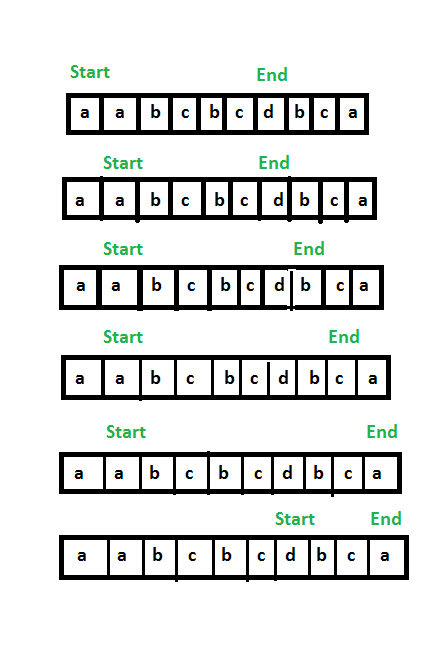
**Output**

Smallest window containing all distinct characters is: dbca

* **Complexity Analysis:**
  + **Time Complexity:** O(N^2).   
    This time is required to generate all possible sub-strings of a string of length “N”.
  + **Space Complexity:** O(N).   
    As a **hash\_map** has been used of size N.

**Method 2:** Here we have used [Sliding Window](https://www.geeksforgeeks.org/window-sliding-technique/) technique to arrive at the solution. This technique shows how a nested for loop in few problems can be converted to single for loop and hence reducing the time complexity.

* **Approach:** Basically a window of characters is maintained by using two pointers namely **start** and **end**. These **start** and **end** pointers can be used to shrink and increase the size of window respectively. Whenever the window contains all characters of given string, the window is shrinked from left side to remove extra characters and then its length is compared with the smallest window found so far.   
  If in the present window, no more characters can be deleted then we start increasing the size of the window using the **end** until all the distinct characters present in the string are also there in the window. Finally, find the minimum size of each window.



* **Algorithm :**
  1. Maintain an array **(visited)** of maximum possible characters (256 characters) and as soon as we find any in the string, mark that index in the array **(this is to count all distinct characters in the string).**
  2. Take two pointers **start** and **end** which will mark the start and end of window.
  3. Take a **counter=0** which will be used to count distinct characters in the window.
  4. Now start reading the characters of the given string and if we come across a character which has not been visited yet increment the counter by **1**.
  5. If the **counter** is equal to total number of distinct characters, Try to shrink the window.
  6. **For shrinking the window -:**
     1. If the **frequency** of character at start pointer is **greater than 1** increment the pointer as it is redundant.
     2. Now compare the length of present window with the minimum window length.
* **Implementation:**

// C++ program to find the smallest

// window containing all characters

// of a pattern.

#include <bits/stdc++.h>

using namespace std;

const int MAX\_CHARS = 256;

// Function to find smallest window containing

// all distinct characters

string findSubString(string str)

{

int n = str.length();

// if string is empty or having one char

if (n <= 1)

return str;

// Count all distinct characters.

int dist\_count = 0;

bool visited[MAX\_CHARS] = { false };

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

if (visited[str[i]] == false) {

visited[str[i]] = true;

dist\_count++;

}

}

// Now follow the algorithm discussed in below

// post. We basically maintain a window of characters

// that contains all characters of given string.

int start = 0, start\_index = -1, min\_len = INT\_MAX;

int count = 0;

int curr\_count[MAX\_CHARS] = { 0 };

for (int j = 0; j < n; j++) {

// Count occurrence of characters of string

curr\_count[str[j]]++;

// If any distinct character matched,

// then increment count

if (curr\_count[str[j]] == 1)

count++;

// if all the characters are matched

if (count == dist\_count) {

// Try to minimize the window i.e., check if

// any character is occurring more no. of times

// than its occurrence in pattern, if yes

// then remove it from starting and also remove

// the useless characters.

while (curr\_count[str[start]] > 1) {

if (curr\_count[str[start]] > 1)

curr\_count[str[start]]--;

start++;

}

// Update window size

int len\_window = j - start + 1;

if (min\_len > len\_window) {

min\_len = len\_window;

start\_index = start;

}

}

}

// Return substring starting from start\_index

// and length min\_len

return str.substr(start\_index, min\_len);

}

// Driver code

int main()

{

string str = "aabcbcdbca";

cout << "Smallest window containing all distinct"

" characters is: "

<< findSubString(str);

return 0;

}

**Output**

Smallest window containing all distinct characters is: dbca

* **Complexity Analysis:**
  + **Time Complexity:** O(N).   
    As the string is traversed using two pointers only once.
  + **Space Complexity:** O(N).   
    As a **hash\_map** is used of size N

# Rearrange characters in a string such that no two adjacent are same

**Rearrange characters**

Given a string S with repeated characters (only lowercase). The task is to rearrange characters in a string such that no two adjacent characters are same.

Note: It may be assumed that the string has only lowercase English alphabets.

**Input:**

The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. Each test case contains a single line containing a string of lowercase english alphabets.

**Output:**

For each test case in a new line print "1" (without quotes) if the generated string doesn't contains any same adjacent characters, else if no such string is possible to be made print "0" (without quotes).

**Constraints:**

1 <= T <= 100

1 <= length of string <= 104

**Example:**

**Input:**

3

geeksforgeeks

bbbabaaacd

bbbbb

**Output:**

1

1

0

**Explanation:**

Testcase 1: All the repeated characters of the given string can be rearranged so that no adiacent characters in the string is equal

Testcase 3: Repeated characters in the string cannot be rearranged such that there should not be any adjacent repeated character.

## Solution:

#include <bits/stdc++.h>

using namespace std;

int main() {

//code

int t;

cin>>t;

while(t--){

string s;

cin>>s;

unordered\_map<int,int> mp;

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

mp[s[i]]++;

auto i=mp.begin();

for(; i!=mp.end(); i++){

if(i->second>(s.size()+1)/2){

cout<<"0"<<endl;

break;

}

}

if(i==mp.end())

cout<<"1"<<endl;

}

return 0;

}

**Time Complexity :** O(n)

**Space Complexity:** O(n)

# Minimum characters to be added at front to make string palindrome

Given a string str we need to tell minimum characters to be added at front of string to make string palindrome.  
**Examples:** 

Input : str = "ABC"

Output : 2

We can make above string palindrome as "CBABC"

by adding 'B' and 'C' at front.

Input : str = "AACECAAAA";

Output : 2

We can make above string palindrome as AAAACECAAAA

by adding two A's at front of string.

## Solution:

**Naive approach:** Start checking the string each time if it is a palindrome and if not, then delete the last character and check again. When the string gets reduced to wither a palindrome or empty then the number of characters deleted from the end till now will be the answer as those characters could have been inserted at the beginning of the original string in the order which will will make the string a palindrome.  
Below is the implementation of the above approach:

// C++ program for getting minimum character to be

// added at front to make string palindrome

#include<bits/stdc++.h>

using namespace std;

// function for checking string is palindrome or not

bool ispalindrome(string s)

{

int l = s.length();

int j;

for(int i = 0, j = l - 1; i <= j; i++, j--)

{

if(s[i] != s[j])

return false;

}

return true;

}

// Driver code

int main()

{

string s = "BABABAA";

int cnt = 0;

int flag = 0;

while(s.length()>0)

{

// if string becomes palindrome then break

if(ispalindrome(s))

{

flag = 1;

break;

}

else

{

cnt++;

// erase the last element of the string

s.erase(s.begin() + s.length() - 1);

}

}

// print the number of insertion at front

if(flag)

cout << cnt;

}

**Output:**

2

**Efficient approach:** We can solve this problem **efficiently in O(n) time** using [lps array of KMP algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/).   
First we concat string by concatenating given string, a special character and reverse of given string then we will get lps array for this concatenated string, recall that each index of lps array represent longest proper prefix which is also suffix. We can use this lps array for solving the problem. 

For string = AACECAAAA

Concatenated String = AACECAAAA$AAAACECAA

LPS array will be {0, 1, 0, 0, 0, 1, 2, 2, 2,

0, 1, 2, 2, 2, 3, 4, 5, 6, 7}

Here we are only interested in the last value of this lps array because it shows us the largest suffix of the reversed string that matches the prefix of the original string i.e these many characters already satisfy the palindrome property. Finally minimum number of character needed to make the string a palindrome is length of the input string minus last entry of our lps array. Pease see below code for better understanding

// C++ program for getting minimum character to be

// added at front to make string palindrome

#include <bits/stdc++.h>

using namespace std;

// returns vector lps for given string str

vector<int> computeLPSArray(string str)

{

int M = str.length();

vector<int> lps(M);

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 (str[i] == str[len])

{

len++;

lps[i] = len;

i++;

}

else // (str[i] != str[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++;

}

}

}

return lps;

}

// Method returns minimum character to be added at

// front to make string palindrome

int getMinCharToAddedToMakeStringPalin(string str)

{

string revStr = str;

reverse(revStr.begin(), revStr.end());

// Get concatenation of string, special character

// and reverse string

string concat = str + "$" + revStr;

// Get LPS array of this concatenated string

vector<int> lps = computeLPSArray(concat);

// By subtracting last entry of lps vector from

// string length, we will get our result

return (str.length() - lps.back());

}

// Driver program to test above functions

int main()

{

string str = "AACECAAAA";

cout << getMinCharToAddedToMakeStringPalin(str);

return 0;

}

**Output:**

2

# Given a sequence of words, print all anagrams together

Given an array of strings, return all groups of strings that are anagrams. The groups must be created in order of their appearance in the original array. Look at the sample case for clarification.

**Example 1:**

**Input:**

N = 5

words[] = {act,god,cat,dog,tac}

**Output:**

god dog

act cat tac

**Explanation:**

There are 2 groups of

anagrams "god", "dog" make group 1.

"act", "cat", "tac" make group 2.

**Example 2:**

**Input:**

N = 3

words[] = {no,on,is}

**Output:**

no on

is

**Explanation:**

There are 2 groups of

anagrams "no", "on" make group 1.

"is" makes group 2.

**Your Task:**  
The task is to complete the function **Anagrams()** that takes a list of strings as input and returns a list of groups such that each group consists of all the strings that are anagrams.

**Expected Time Complexity:** O(N\*|S|\*log|S|), where |S| is the length of the strings.  
**Expected Auxiliary Space:** O(N\*|S|), where |S| is the length of the strings.

**Constraints:**  
1<=N<=100

## Solution:

A **simple method** is to create a Hash Table. Calculate the hash value of each word in such a way that all anagrams have the same hash value. Populate the Hash Table with these hash values. Finally, print those words together with same hash values. A simple hashing mechanism can be modulo sum of all characters. With modulo sum, two non-anagram words may have same hash value. This can be handled by matching individual characters.

Following is **another method** to print all anagrams together. Take two auxiliary arrays, index array and word array. Populate the word array with the given sequence of words. Sort each individual word of the word array. Finally, sort the word array and keep track of the corresponding indices. After sorting, all the anagrams cluster together. Use the index array to print the strings from the original array of strings.

Let us understand the steps with following input Sequence of Words:

"cat", "dog", "tac", "god", "act"

**1)** Create two auxiliary arrays index[] and words[]. Copy all given words to words[] and store the original indexes in index[]

index[]: 0 1 2 3 4

words[]: cat dog tac god act

**2)** Sort individual words in words[]. Index array doesn’t change.

index[]: 0 1 2 3 4

words[]: act dgo act dgo act

**3)** Sort the words array. Compare individual words using strcmp() to sort

index: 0 2 4 1 3

words[]: act act act dgo dgo

**4)** All anagrams come together. But words are changed in words array. To print the original words, take index from the index array and use it in the original array. We get

"cat tac act dog god"

Following are the implementations of the above algorithm. In the following program, an array of structure “Word” is used to store both index and word arrays. DupArray is another structure that stores array of structure “Word”.

// A C++ program to print all anagrams together

#include <bits/stdc++.h>

#include <string.h>

using namespace std;

// structure for each word of duplicate array

class Word {

public:

char\* str; // to store word itself

int index; // index of the word in the original array

};

// structure to represent duplicate array.

class DupArray {

public:

Word\* array; // Array of words

int size; // Size of array

};

// Create a DupArray object that contains an array of Words

DupArray\* createDupArray(char\* str[], int size)

{

// Allocate memory for dupArray and all members of it

DupArray\* dupArray = new DupArray();

dupArray->size = size;

dupArray->array = new Word[(dupArray->size \* sizeof(Word))];

// One by one copy words from the given wordArray to dupArray

int i;

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

dupArray->array[i].index = i;

dupArray->array[i].str = new char[(strlen(str[i]) + 1)];

strcpy(dupArray->array[i].str, str[i]);

}

return dupArray;

}

// Compare two characters. Used in qsort() for

// sorting an array of characters (Word)

int compChar(const void\* a, const void\* b)

{

return \*(char\*)a - \*(char\*)b;

}

// Compare two words. Used in qsort()

// for sorting an array of words

int compStr(const void\* a, const void\* b)

{

Word\* a1 = (Word\*)a;

Word\* b1 = (Word\*)b;

return strcmp(a1->str, b1->str);

}

// Given a list of words in wordArr[],

void printAnagramsTogether(char\* wordArr[], int size)

{

// Step 1: Create a copy of all words present in given wordArr.

// The copy will also have original indexes of words

DupArray\* dupArray = createDupArray(wordArr, size);

// Step 2: Iterate through all words in dupArray and sort

// individual words.

int i;

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

qsort(dupArray->array[i].str,

strlen(dupArray->array[i].str), sizeof(char), compChar);

// Step 3: Now sort the array of words in dupArray

qsort(dupArray->array, size, sizeof(dupArray->array[0]), compStr);

// Step 4: Now all words in dupArray are together, but these

// words are changed. Use the index member of word struct to

// get the corresponding original word

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

cout << wordArr[dupArray->array[i].index] << " ";

}

// Driver program to test above functions

int main()

{

char\* wordArr[] = { "cat", "dog", "tac", "god", "act" };

int size = sizeof(wordArr) / sizeof(wordArr[0]);

printAnagramsTogether(wordArr, size);

return 0;

}

**Output:**

cat tac act dog god

***Time Complexity:*** Let there be N words and each word has maximum M characters. The upper bound is O(NMLogM + MNLogN).   
Step 2 takes O(NMLogM) time. Sorting a word takes maximum O(MLogM) time. So sorting N words takes O(NMLogM) time. step 3 takes O(MNLogN) Sorting array of words takes NLogN comparisons. A comparison may take maximum O(M) time. So time to sort array of words will be O(MNLogN).

**Approach Using hashmap:** This is a HashMap solution using C++ Standard Template Library which stores the Key-Value Pair. In the hashmap, the key will be the sorted set of characters and value will be the output string. Two anagrams will be similar when their characters are sorted. Now,

1. Store the vector elements in HashMap with key as the sorted string.
2. If the key is same, then add the string to the value of HashMap(string vector).
3. Traverse the HashMap and print the anagram strings.

// C++ program for finding all anagram

// pairs in the given array

#include <algorithm>

#include <iostream>

#include <unordered\_map>

#include <vector>

using namespace std;

// Utility function for

// printing anagram list

void printAnagram(unordered\_map<string,vector<string> >& store)

{

for (auto it:store)

{

vector<string> temp\_vec(it.second);

int size = temp\_vec.size();

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

cout << temp\_vec[i] << " ";

cout << "\n";

}

}

// Utility function for storing

// the vector of strings into HashMap

void storeInMap(vector<string>& vec)

{

unordered\_map<string,vector<string> > store;

for (int i = 0; i < vec.size(); i++)

{

string tempString(vec[i]);

// sort the string

sort(tempString.begin(),tempString.end());

// make hash of a sorted string

store[tempString].push\_back(vec[i]);

}

// print utility function for printing

// all the anagrams

printAnagram(store);

}

// Driver code

int main()

{

// initialize vector of strings

vector<string> arr;

arr.push\_back("geeksquiz");

arr.push\_back("geeksforgeeks");

arr.push\_back("abcd");

arr.push\_back("forgeeksgeeks");

arr.push\_back("zuiqkeegs");

arr.push\_back("cat");

arr.push\_back("act");

arr.push\_back("tca");

// utility function for storing

// strings into hashmap

storeInMap(arr);

return 0;

}

***Note:****Compile above program with -std=c++11 flag in g++*

**Output:**

cat act tca

geeksforgeeks forgeeksgeeks

geeksquiz zuiqkeegs

**Complexity Analysis:**

* **Time Complexity:** O(n \* m(log m)),  where m is the length of a word.  
  A single traversal through the array is needed.
* **Space Complexity:** O(n).   
  There are n words in a string. The map requires O(n) space to store the strings.

**HashMap with O(N\*M + N\*logN) Solution**  
In the previous approach, we were sorting every string in order to maintain a similar key, but that cost extra time in this approach will take the advantage of another hashmap to maintain the frequency of the characters which will generate the same hash function for different string having same frequency of characters.  
Here, we will take HashMap<HashMap, ArrayList>, the inner hashmap will count the frequency of the characters of each string and the outer HashMap will check whether that hashmap is present or not if present then it will add that string to the corresponding list.

// C++ code to print all anagrams together

#include <bits/stdc++.h>

using namespace std;

void solver(vector<string> my\_list)

{

// Inner hashmap counts frequency

// of characters in a string.

// Outer hashmap for if same

// frequency characters are present in

// in a string then it will add it to

// the vector.

map<map<char, int>, vector<string>> my\_map;

// Loop over all words

for(string str : my\_list)

{

// Counting the frequency of the

// characters present in a string

map<char, int> temp\_map;

vector<string> temp\_my\_list;

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

{

++temp\_map[str[i]];

}

// If the same frequency of characters

// are already present then add that

// string into that arraylist otherwise

// created a new arraylist and add that

// string

auto it = my\_map.find(temp\_map);

if (it != my\_map.end())

{

it->second.push\_back(str);

}

else

{

temp\_my\_list.push\_back(str);

my\_map.insert({ temp\_map, temp\_my\_list });

}

}

// Stores the result in a vector

vector<vector<string>> result;

for(auto it = my\_map.begin();

it != my\_map.end(); ++it)

{

result.push\_back(it->second);

}

for(int i = 0; i < result.size(); ++i)

{

cout << "[";

for(int j = 0; j < result[i].size(); ++j)

{

cout << result[i][j] << ", ";

}

cout << "]";

}

}

// Driver code

int main()

{

vector<string> my\_list = { "cat", "dog", "ogd",

"god", "atc" };

solver(my\_list);

return 0;

}

**Output:**

[[cat, atc], [dog, ogd, god]]

***Time Complexity:*** Let there be N words and each word has maximum M characters. The upper bound is O(NM+NlogN) in worst case since insert function takes O(logN) time. While in best case when all elements are anagram of each other, it takes O(NM) time.  
***Space Complexity:*** Let there be N words and each word has maximum M characters, therefore max. storage space for each word with at max. M characters will be O(M), therefore for max N words, it will be O(N\*M). Therefore, the upper bound is O(NM).

# Find the smallest window in a string containing all characters of another string

Given two strings, string1 and string2, the task is to find the smallest substring in string1 containing all characters of string2 efficiently.

**Examples:**

***Input:****string = “this is a test string”, pattern = “tist”****Output:****Minimum window is “t stri”****Explanation:****“t stri” contains all the characters of pattern.*

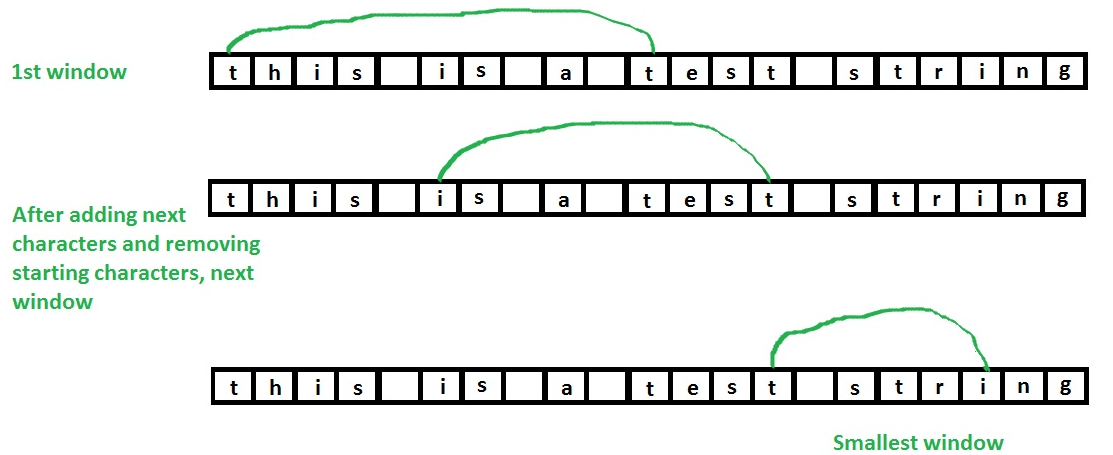
***Input:****string = “geeksforgeeks”, pattern = “ork”****Output:****Minimum window is “ksfor”*

## Solution:

**Method 1 ( Brute force solution )**   
1- Generate all substrings of string1 (“this is a test string”)   
2- For each substring, check whether the substring contains all characters of string2 (“tist”)   
3- Finally, print the smallest substring containing all characters of string2.  
    
**Method 2 ( Efficient Solution )**

1. First check if the length of the string is less than the length of the given pattern, if yes then “**no such window can exist** “.
2. Store the occurrence of characters of the given pattern in a hash\_pat[].
3. we will be using two pointer technique basically
4. Start matching the characters of pattern with the characters of string i.e. increment count if a character matches.
5. Check if (count == length of pattern ) this means a window is found.
6. If such a window found, try to minimize it by removing extra characters from the beginning of the current window.
7. delete one character from first and again find this deleted key at right, once found apply step 5 .
8. Update min\_length.
9. Print the minimum length window.

**A diagram to explain the stated algorithm:**



*after the second image(array) our left pointer should be at s and then find I at right and then apply step5 (basically one step has not been shown)*

Below is the program to implement the above algorithm:

// C++ program to find

// smallest window containing

// all characters of a pattern.

#include <bits/stdc++.h>

using namespace std;

const int no\_of\_chars = 256;

// Function to find smallest

// window containing

// all characters of 'pat'

string findSubString(string str, string pat)

{

int len1 = str.length();

int len2 = pat.length();

// Check if string's length

// is less than pattern's

// length. If yes then no such

// window can exist

if (len1 < len2) {

cout << "No such window exists";

return "";

}

int hash\_pat[no\_of\_chars] = { 0 };

int hash\_str[no\_of\_chars] = { 0 };

// Store occurrence ofs characters

// of pattern

for (int i = 0; i < len2; i++)

hash\_pat[pat[i]]++;

int start = 0, start\_index = -1, min\_len = INT\_MAX;

// Start traversing the string

// Count of characters

int count = 0;

for (int j = 0; j < len1; j++) {

// Count occurrence of characters

// of string

hash\_str[str[j]]++;

// If string's char matches with

// pattern's char

// then increment count

if (hash\_str[str[j]] <= hash\_pat[str[j]])

count++;

// if all the characters are matched

if (count == len2) {

// Try to minimize the window

while (hash\_str[str[start]]

> hash\_pat[str[start]]

|| hash\_pat[str[start]] == 0) {

if (hash\_str[str[start]]

> hash\_pat[str[start]])

hash\_str[str[start]]--;

start++;

}

// update window size

int len\_window = j - start + 1;

if (min\_len > len\_window) {

min\_len = len\_window;

start\_index = start;

}

}

}

// If no window found

if (start\_index == -1) {

cout << "No such window exists";

return "";

}

// Return substring starting from start\_index

// and length min\_len

return str.substr(start\_index, min\_len);

}

// Driver code

int main()

{

string str = "this is a test string";

string pat = "tist";

cout << "Smallest window is : \n"

<< findSubString(str, pat);

return 0;

}

**Output**

Smallest window is :

t stri

**Method 3 (Efficient Solution)**

(Using sliding window  Technique )

#include <bits/stdc++.h>

using namespace std;

// Function

string Minimum\_Window(string s, string t)

{

int m[256] = { 0 };

// Answer

int ans = INT\_MAX; // length of ans

int start = 0; // starting index of ans

int count = 0;

// creating map

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

if (m[t[i]] == 0)

count++;

m[t[i]]++;

}

// References of Window

int i = 0;

int j = 0;

// Traversing the window

while (j < s.length()) {

// Calculations

m[s[j]]--;

if (m[s[j]] == 0)

count--;

// Condition matching

if (count == 0) {

while (count == 0) {

// Sorting ans

if (ans > j - i + 1) {

ans = min(ans, j - i + 1);

start = i;

}

// Sliding I

// Calculation for removing I

m[s[i]]++;

if (m[s[i]] > 0)

count++;

i++;

}

}

j++;

}

if (ans != INT\_MAX)

return s.substr(start, ans);

else

return "-1";

}

main()

{

string s = "ADOBECODEBANC";

string t = "ABC";

cout<<"-->Smallest window that contain all character : "<<endl;

cout << Minimum\_Window(s, t);

}

**Time Complexity  :**O(|s|) , where |s| is the length of string s.

**Space Complexity** : O(1)

Explanation –   Array m  of length 256 is used ,which is constant space, so the Space Complexity is O(1).

# Recursively remove all adjacent duplicates

Given a string**S**delete the characters which are appearing more than once consecutively.

**Example 1:**

**Input:**

S = aabb

**Output:** ab

**Explanation:** 'a' at 2nd position is

appearing 2nd time consecutively.

Similiar explanation for b at

4th position.

**Example 2:**

**Input:**

S = aabaa

**Output:** aba

**Explanation:** 'a' at 2nd position is

appearing 2nd time consecutively.

'a' at fifth position is appearing

2nd time consecutively.

**Your Task:**  
You dont need to read input or print anything. Complete the function **removeConsecutiveCharacter()** which accepts a string as input parameter and **returns** modified string.

**Expected Time Complexity:**O(|S|).  
**Expected Auxiliary Space:**O(|S|).

**Constraints:**  
1<=|S|<=105  
All characters are lowercase alphabets.

## Solution:

Traverse the string and append char to result if it is different from the previous char.

class Solution{

public:

string removeConsecutiveCharacter(string S)

{

// code here.

if(S.size()==0)

return "";

string res = "";

res += S[0];

for(int i=1;i<S.size();i++){

if(S[i]!=S[i-1])

res += S[i];

}

return res;

}

};

**Time Complexity:** O(n), where n is the size of the input string.

**Space Complexity:** O(n), where n is the size of the input string.

# String matching where one string contains wildcard characters

Given two strings **wild** and **pattern** where wild string may contain wild card characters and pattern string is a normal string. Determine if the two strings match. The following are the 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.

**Example 1:**

**Input:** wild = ge\*ks

pattern = geeks

**Output:** Yes

**Explanation:** Replace the '\*' with 'e' to obtain

the string.

**Example 2:**

**Input:** wild =ge?ks\*

pattern = geeksforgeeks

**Output:** Yes

**Explanation:** Replace '?' with 'e' and '\*' with

'forgeeks' and it will be same as pattern.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **match()**which takesthe string wild and pattern as input parameters and returns true if the string wild can be made equal to the string pattern, otherwise, returns false.

**Expected Time Complexity:** O(length of wild string + length of pattern string)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1<=length of the two string<=10^3

## Solution:

// 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;

}

**Output:**

Yes

Yes

No

No

Yes

No

Yes

Yes

**Approach using DP:**

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 the above Dynamic Programming approach.

// 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;

}

**Output**

Yes

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

**DP Memoization solution:-**

// C++ program to implement wildcard

// pattern matching algorithm

#include <bits/stdc++.h>

using namespace std;

// Function that matches input str with

// given wildcard pattern

vector<vector<int> > dp;

int finding(string& s, string& p, int n, int m)

{

// return 1 if n and m are negative

if (n < 0 && m < 0)

return 1;

// return 0 if m is negative

if (m < 0)

return 0;

// return n if n is negative

if (n < 0)

{

// while m is positive

while (m >= 0)

{

if (p[m] != '\*')

return 0;

m--;

}

return 1;

}

// if dp state is not visited

if (dp[n][m] == -1)

{

if (p[m] == '\*')

{

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

|| finding(s, p, n, m - 1);

}

else

{

if (p[m] != s[n] && p[m] != '?')

return dp[n][m] = 0;

else

return dp[n][m]

= finding(s, p, n - 1, m - 1);

}

}

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

return dp[n][m];

}

bool isMatch(string s, string p)

{

dp.clear();

// resize the dp array

dp.resize(s.size() + 1, vector<int>(p.size() + 1, -1));

return dp[s.size()][p.size()]

= finding(s, p, s.size() - 1, p.size() - 1);

}

// Driver code

int main()

{

string str = "baaabab";

string 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 (isMatch(str, pattern))

cout << "Yes" << endl;

else

cout << "No" << endl;

return 0;

}

**Output**

Yes

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

**Further Improvements:**   
We can improve space complexity by making use of the fact that we only uses the result from last row.   
One more improvement is you can merge consecutive ‘\*’ in the pattern to single ‘\*’ as they mean the same thing. For example for pattern “\*\*\*\*\*ba\*\*\*\*\*ab”, if we merge consecutive stars, the resultant string will be “\*ba\*ab”. So, value of m is reduced from 14 to 6.

**Linear Time and Constant Space solution:**

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

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

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

Below is the implementation of the above approach:

// 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 (j < m && 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

**Complexity Analysis:**

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

# Function to find Number of customers who could not get a computer

Write a function “runCustomerSimulation” that takes following two inputs   
a) An integer ‘n’: total number of computers in a cafe and a string:   
b) A sequence of uppercase letters ‘seq’: Letters in the sequence occur in pairs. The first occurrence indicates the arrival of a customer; the second indicates the departure of that same customer.   
A customer will be serviced if there is an unoccupied computer. No letter will occur more than two times.   
Customers who leave without using a computer always depart before customers who are currently using the computers. There are at most 20 computers per cafe.  
For each set of input the function should output a number telling how many customers, if any walked away without using a computer. Return 0 if all the customers were able to use a computer.  
runCustomerSimulation (2, “ABBAJJKZKZ”) should return 0  
runCustomerSimulation (3, “GACCBDDBAGEE”) should return 1 as ‘D’ was not able to get any computer  
runCustomerSimulation (3, “GACCBGDDBAEE”) should return 0  
runCustomerSimulation (1, “ABCBCA”) should return 2 as ‘B’ and ‘C’ were not able to get any computer.  
runCustomerSimulation(1, “ABCBCADEED”) should return 3 as ‘B’, ‘C’ and ‘E’ were not able to get any computer.

## Solution:

Below are simple steps to find number of customers who could not get any computer.  
1) Initialize result as 0.  
2) Traverse the given sequence. While traversing, keep track of occupied computers (this can be done by keeping track of characters which have appeared only once and a computer was available when they appeared). At any point, if count of occupied computers is equal to ‘n’, and there is a new customer, increment result by 1.  
The important thing is to keep track of existing customers in cafe in a way that can indicate whether the customer has got a computer or not. Note that in sequence “ABCBCADEED”, customer ‘B’ did not get a seat, but still in cafe as a new customer ‘C’ is next in sequence.  
Below are implementations of above idea.

// C++ program to find number of customers who couldn't get a resource.

#include<iostream>

#include<cstring>

using namespace std;

#define MAX\_CHAR 26

// n is number of computers in cafe.

// 'seq' is given sequence of customer entry, exit events

int runCustomerSimulation(int n, const char \*seq)

{

// seen[i] = 0, indicates that customer 'i' is not in cafe

// seen[1] = 1, indicates that customer 'i' is in cafe but

// computer is not assigned yet.

// seen[2] = 2, indicates that customer 'i' is in cafe and

// has occupied a computer.

char seen[MAX\_CHAR] = {0};

// Initialize result which is number of customers who could

// not get any computer.

int res = 0;

int occupied = 0; // To keep track of occupied computers

// Traverse the input sequence

for (int i=0; seq[i]; i++)

{

// Find index of current character in seen[0...25]

int ind = seq[i] - 'A';

// If First occurrence of 'seq[i]'

if (seen[ind] == 0)

{

// set the current character as seen

seen[ind] = 1;

// If number of occupied computers is less than

// n, then assign a computer to new customer

if (occupied < n)

{

occupied++;

// Set the current character as occupying a computer

seen[ind] = 2;

}

// Else this customer cannot get a computer,

// increment result

else

res++;

}

// If this is second occurrence of 'seq[i]'

else

{

// Decrement occupied only if this customer

// was using a computer

if (seen[ind] == 2)

occupied--;

seen[ind] = 0;

}

}

return res;

}

// Driver program

int main()

{

cout << runCustomerSimulation(2, "ABBAJJKZKZ") << endl;

cout << runCustomerSimulation(3, "GACCBDDBAGEE") << endl;

cout << runCustomerSimulation(3, "GACCBGDDBAEE") << endl;

cout << runCustomerSimulation(1, "ABCBCA") << endl;

cout << runCustomerSimulation(1, "ABCBCADEED") << endl;

return 0;

}

**Output:**

0

1

0

2

3

Time complexity of above solution is O(n) and extra space required is O(CHAR\_MAX) where CHAR\_MAX is total number of possible characters in given sequence.

**My approach:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cin>>n;

string str;

cin>>str;

int res=0;

unordered\_set<char> set;

for(int i=0;i<str.size();i++){

if(set.find(str[i])!=set.end())

set.erase(str[i]);

else{

set.insert(str[i]);

if(set.size()>n)

res++;

}

}

cout<<res;

return 0;

}

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

**Space Complexity:** O(n), where n is the size of the string.

[Transform One String to Another using Minimum Number of Given Operation](https://www.geeksforgeeks.org/transform-one-string-to-another-using-minimum-number-of-given-operation/)

# Transform One String to Another using Minimum Number of Given Operation

Given two strings A and B, the task is to convert A to B if possible. The only operation allowed is to put any character from A and insert it at front. Find if it’s possible to convert the string. If yes, then output minimum no. of operations required for transformation.

**Examples:**

Input: A = "ABD", B = "BAD"

Output: 1

Explanation: Pick B and insert it at front.

Input: A = "EACBD", B = "EABCD"

Output: 3

Explanation: Pick B and insert at front, EACBD => BEACD

Pick A and insert at front, BEACD => ABECD

Pick E and insert at front, ABECD => EABCD

## Solution:

Checking whether a string can be transformed to another is simple. We need to check whether both strings have same number of characters and same set of characters. This can be easily done by creating a count array for first string and checking if second string has same count of every character.   
How to find minimum number of operations when we are sure that we can transform A to B? The idea is to start matching from last characters of both strings. If last characters match, then our task reduces to n-1 characters. If last characters don’t match, then find the position of B’s mismatching character in A. The difference between two positions indicates that these many characters of A must be moved before current character of A.

Below is complete algorithm.   
1) Find if A can be transformed to B or not by first creating a count array for all characters of A, then checking with B if B has same count for every character.   
2) Initialize result as 0.   
3) Start traversing from end of both strings.   
……a) If current characters of A and B match, i.e., A[i] == B[j]   
………then do i = i-1 and j = j-1   
    b) If current characters don’t match, then search B[j] in remaining   
………A. While searching, keep incrementing result as these characters   
………must be moved ahead for A to B transformation.

Below are the implementations based on this idea.

// C++ program to find minimum number of

// operations required to transform one string to other

#include<bits/stdc++.h>

using namespace std;

// Function to find minimum number of operations required to transform

// A to B.

int minOps(string& A, string& B)

{

int m = A.length(), n = B.length();

// This parts checks whether conversion is

// possible or not

if (n != m)

return -1;

int count[256];

memset(count, 0, sizeof(count));

for (int i=0; i<n; i++) // count characters in A

count[B[i]]++;

for (int i=0; i<n; i++) // subtract count for

count[A[i]]--; // every character in B

for (int i=0; i<256; i++) // Check if all counts become 0

if (count[i])

return -1;

// This part calculates the number of operations required

int res = 0;

for (int i=n-1, j=n-1; i>=0; )

{

// If there is a mismatch, then keep incrementing

// result 'res' until B[j] is not found in A[0..i]

while (i>=0 && A[i] != B[j])

{

i--;

res++;

}

// If A[i] and B[j] match

if (i >= 0)

{

i--;

j--;

}

}

return res;

}

// Driver program

int main()

{

string A = "EACBD";

string B = "EABCD";

cout << "Minimum number of operations "

"required is " << minOps(A, B);

return 0;

}

**Output:**

Minimum number of operations required is 3

**Time Complexity:** O(n), please note that i is always decremented (in while loop and in if), and the for loop starts from n-1 and runs while i >= 0.

# Check if two given strings are isomorphic to each other

Given two strings '**str1**' and '**str2**', check if these two strings are isomorphic to each other.  
Two strings str1 and str2 are called isomorphic if there is a one to one mapping possible for every character of str1 to every character of str2 while **preserving the order**.  
Note: All occurrences of every character in ‘str1’ should map to the same character in ‘str2’

**Example 1:**

**Input:**

str1 = aab

str2 = xxy

**Output:** 1

**Explanation:** There are two different

charactersin aab and xxy, i.e a and b

with frequency 2and 1 respectively.

**Example 2:**

**Input:**

str1 = aab

str2 = xyz

**Output:** 0

**Explanation:** There are two different

charactersin aab but there are three

different charactersin xyz. So there

won't be one to one mapping between

str1 and str2.

**Your Task:**  
You don't need to read input or print anything.Your task is to complete the function **areIsomorphic**() which takes the string **str1** and string **str2** as input parameter and  check if two strings are isomorphic. The function returns **true**if strings are isomorphic else it returns **false**.

**Expected Time Complexity:**O(|str1|+|str2|).  
**Expected Auxiliary Space:**O(Number of different characters).  
**Note:** |s| represents the length of string s.

**Constraints:**  
1 <= |str1|, |str2| <= 2\*104

## Solution:

A **Simple Solution** is to consider every character of ‘str1’ and check if all occurrences of it map to the same character in ‘str2’. The time complexity of this solution is O(n\*n).

An**Efficient Solution** can solve this problem in O(n) time. The idea is to create an array to store mappings of processed characters.

1) If lengths of str1 and str2 are not same, return false.

2) Do following for every character in str1 and str2

a) If this character is seen first time in str1,

then current of str2 must have not appeared before.

(i) If current character of str2 is seen, return false.

Mark current character of str2 as visited.

(ii) Store mapping of current characters.

b) Else check if previous occurrence of str1[i] mapped

to same character.

Below is the implementation of above idea :

// C++ program to check if two strings are isomorphic

#include <bits/stdc++.h>

using namespace std;

#define MAX\_CHARS 256

// This function returns true if str1 and str2 are ismorphic

bool areIsomorphic(string str1, string str2)

{

int m = str1.length(), n = str2.length();

// Length of both strings must be same for one to one

// corresponance

if (m != n)

return false;

// To mark visited characters in str2

bool marked[MAX\_CHARS] = { false };

// To store mapping of every character from str1 to

// that of str2. Initialize all entries of map as -1.

int map[MAX\_CHARS];

memset(map, -1, sizeof(map));

// Process all characters one by on

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

// If current character of str1 is seen first

// time in it.

if (map[str1[i]] == -1) {

// If current character of str2 is already

// seen, one to one mapping not possible

if (marked[str2[i]] == true)

return false;

// Mark current character of str2 as visited

marked[str2[i]] = true;

// Store mapping of current characters

map[str1[i]] = str2[i];

}

// If this is not first appearance of current

// character in str1, then check if previous

// appearance mapped to same character of str2

else if (map[str1[i]] != str2[i])

return false;

}

return true;

}

// Driver program

int main()

{

cout << areIsomorphic("aab", "xxy") << endl;

cout << areIsomorphic("aab", "xyz") << endl;

return 0;

}

**Output:**

1

0

**Another Approach:**

1. In this approach, we will count the number of occurrences of a particular character in both the string using two arrays, while we will compare the two arrays if at any moment with the loop the count of the current character in both strings becomes different we return false, else after the loop ends we return true.
2. Follow the code given below you will understand everything.

Note: There is no need to create here array of 256 characters. We can reduce it to only 26 characters

by storing count of **ch-'a'** (ch is the ith character of the string)in count array .This gives the same

result as string consists of only small case characters.

Below is the implementation of the above idea :

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

#define MAX\_CHARS 26

// This function returns true if

// str1 and str2 are ismorphic

bool areIsomorphic(string str1, string str2)

{

int n = str1.length(), m = str2.length();

// Length of both strings must be

// same for one to one

// correspondence

if (n != m)

return false;

// For counting the previous appearances of character in

// both the strings

int count[MAX\_CHARS] = { 0 };

int dcount[MAX\_CHARS] = { 0 };

// Process all characters one by one

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

count[str1[i] - 'a']++;

dcount[str2[i] - 'a']++;

// For string to be isomorphic the previous counts

// of appearances of

// current character in both string must be same if

// it is not same we return false.

if (count[str1[i] - 'a'] != dcount[str2[i] - 'a'])

return false;

}

return true;

}

// Driver Code

int main()

{

cout << areIsomorphic("aab", "xxy") << endl;

cout << areIsomorphic("aab", "xyz") << endl;

return 0;

}

**Output**

1

0

**Time Complexity:**O(n)

**Another Approach:**

1)  In this approach we will use just a single dictionary for a Key value pair for the respective characters in the first and second string.

2) If the Key repeats we check if the value matches in the respective index.

// C# program to check if two strings

// areIsIsomorphic

using System;

using System.Collections.Generic;

public class GFG {

static bool areIsomorphic(char[] str1, char[] str2)

{

Dictionary<char, char> charCount

= new Dictionary<char, char>();

char c = 'a';

for (int i = 0; i < str1.Length; i++) {

if (charCount.ContainsKey(str1[i])

&& charCount.TryGetValue(str1[i], out c)) {

if (c != str2[i])

return false;

}

else if (!charCount.ContainsValue(str2[i])) {

charCount.Add(str1[i], str2[i]);

}

else {

return false;

}

}

return true;

}

/\* Driver code\*/

public static void Main()

{

string str1 = "aac";

string str2 = "xxy";

// Function Call

if (str1.Length == str2.Length

&& areIsomorphic(str1.ToCharArray(),

str2.ToCharArray()))

Console.WriteLine(1);

else

Console.WriteLine(0);

Console.ReadLine();

}

}

**Output**

1

**My Implementaion:**

bool areIsomorphic(string str1, string str2)

{

int m=str1.size(), n=str2.size();

if(m!=n)

return false;

char mp1[256], mp2[256];

memset(mp1,-1,sizeof(mp1));

memset(mp2,-1,sizeof(mp2));

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

if(mp1[str1[i]]==-1 && mp2[str2[i]]==-1){

mp1[str1[i]] = str2[i];

mp2[str2[i]] = str1[i];

}

else if(str2[i]==mp1[str1[i]] && str1[i]==mp2[str2[i]])

continue;

else

return 0;

}

return 1;

}

**Time Complexity:** O(n), where n is the size of the strings.

**Space Complexity:** O(MAX\_CHARS) ie., O(256) = O(1)

# Recursively print all sentences that can be formed from list of word lists

Given a list of word lists How to print all sentences possible taking one word from a list at a time via recursion?   
**Example:**

Input: {{"you", "we"},

{"have", "are"},

{"sleep", "eat", "drink"}}

Output:

you have sleep

you have eat

you have drink

you are sleep

you are eat

you are drink

we have sleep

we have eat

we have drink

we are sleep

we are eat

we are drink

## Solution:

The idea is based on a simple depth-first traversal. We start from every word of the first list as the first word of an output sentence, then recur for the remaining lists.  
Below is the C++ implementation of the above idea. In the below implementation, the input list of the list is considered as a 2D array. If we take a closer look, we can observe that the code is very close to the [DFS of graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/).

// C++ program to print all possible sentences from a list of word list

#include <iostream>

#include <string>

#define R 3

#define C 3

using namespace std;

// A recursive function to print all possible sentences that can be formed

// from a list of word list

void printUtil(string arr[R][C], int m, int n, string output[R])

{

// Add current word to output array

output[m] = arr[m][n];

// If this is last word of current output sentence, then print

// the output sentence

if (m==R-1)

{

for (int i=0; i<R; i++)

cout << output[i] << " ";

cout << endl;

return;

}

// Recur for next row

for (int i=0; i<C; i++)

if (arr[m+1][i] != "")

printUtil(arr, m+1, i, output);

}

// A wrapper over printUtil()

void print(string arr[R][C])

{

// Create an array to store sentence

string output[R];

// Consider all words for first row as starting points and

// print all sentences

for (int i=0; i<C; i++)

if (arr[0][i] != "")

printUtil(arr, 0, i, output);

}

// Driver program to test above functions

int main()

{

string arr[R][C] = {{"you", "we"},

{"have", "are"},

{"sleep", "eat", "drink"}};

print(arr);

return 0;

}

Output:

you have sleep

you have eat

you have drink

you are sleep

you are eat

you are drink

we have sleep

we have eat

we have drink

we are sleep

we are eat

we are drink

**Time Complexity:** O(n1\*n2\*n3\*…..), where n1,n2,n3…… are the no. of words in each list.

n1\*n2\*n3…… is the number of sentences possible.

**Space Complexity:** O(m\*n1\*n2\*n3\*…..) bcz there are n1\*n2\*n3……. recursive calls. In each call, we have to maintain an array of word of current sentence which is of length m.