**📌 Top 10 String & Pattern Matching Questions Asked in Companies**

📌 **1. Reverse Words in a String**  
**Asked in:** Amazon, Google, TCS

* Given a string **S**, reverse the order of words without reversing individual words.

📌 **2. Check if a String is a Palindrome**  
**Asked in:** Flipkart, Wipro, TCS

* Determine if a given string **S** is a palindrome (reads the same forward and backward).

📌 **3. Check if Two Strings are Anagrams**  
**Asked in:** Amazon, Google, Microsoft

* Given two strings **S1** and **S2**, check if they are anagrams (contain the same characters in any order).

📌 **4. Find the Longest Palindromic Substring**  
**Asked in:** Google, Amazon, Flipkart

* Given a string **S**, find the longest substring that is a palindrome.

📌 **5. Implement the KMP Algorithm for Pattern Matching**  
**Asked in:** Amazon, Google, Microsoft

* Given a string **S** and a pattern **P**, implement **KMP (Knuth-Morris-Pratt)** algorithm to find all occurrences of **P** in **S** efficiently.

📌 **6. Implement the Rabin-Karp Algorithm for String Matching**  
**Asked in:** Adobe, Google, TCS

* Given a string **S** and a pattern **P**, implement **Rabin-Karp** algorithm to search for **P** in **S** using hashing.

📌 **7. Find the Longest Common Prefix (LCP) Among Strings**  
**Asked in:** Microsoft, Google, Amazon

* Given an array of strings, find the longest common prefix among them.

📌 **8. Find All Occurrences of a Pattern in a Text Using Z-Algorithm**  
**Asked in:** Flipkart, Amazon, TCS

* Given a text **S** and pattern **P**, use **Z-Algorithm** to efficiently find all occurrences of **P** in **S**.

📌 **9. Build a Suffix Array for a Given String**  
**Asked in:** Google, Microsoft, Amazon

* Construct a **suffix array** for a given string **S**, which helps in efficient pattern matching.

📌 **10. Find the Longest Repeated Substring Using Suffix Arrays & LCP**  
**Asked in:** Google, Facebook, Microsoft

* Given a string **S**, find the longest repeated substring using **suffix arrays** and **LCP (Longest Common Prefix)**.

📌 1. Reverse Words in a String Asked in: Amazon, Google, TCS Given a string S, reverse the order of words without reversing individual words.

#### **C++ Solution**

#include <bits/stdc++.h>

using namespace std;

string reverseWords(string s) {

stringstream ss(s);

string word, result = "";

vector<string> words;

while (ss >> word)

words.push\_back(word);

reverse(words.begin(), words.end());

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

result += words[i];

if (i != words.size() - 1) result += " ";

}

return result;

}

int main() {

string s = "Hello World from Syntax Error";

cout << reverseWords(s) << endl;

return 0;

}

#### **Java Solution**

import java.util.\*;

public class ReverseWords {

public static String reverseWords(String s) {

String[] words = s.trim().split("\\s+");

Collections.reverse(Arrays.asList(words));

return String.join(" ", words);

}

public static void main(String[] args) {

String s = "Hello World from Syntax Error";

System.out.println(reverseWords(s));

}

}

#### **Python Solution**

def reverse\_words(s):

words = s.split()

return " ".join(reversed(words))

# Example usage

s = "Hello World from Syntax Error"

print(reverse\_words(s))

📌 **2. Check if a String is a Palindrome**  
**Asked in:** Flipkart, Wipro, TCS

* Determine if a given string **S** is a palindrome (reads the same forward and backward).

#### **C++ Solution**

#include <bits/stdc++.h>

using namespace std;

string reverseWords(string s) {

stringstream ss(s);

string word, result = "";

vector<string> words;

while (ss >> word)

words.push\_back(word);

reverse(words.begin(), words.end());

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

result += words[i];

if (i != words.size() - 1) result += " ";

}

return result;

}

int main() {

string s = "Hello World from Syntax Error";

cout << reverseWords(s) << endl;

return 0;

}

#### **Java Solution**

import java.util.\*;

public class ReverseWords {

public static String reverseWords(String s) {

String[] words = s.trim().split("\\s+");

Collections.reverse(Arrays.asList(words));

return String.join(" ", words);

}

public static void main(String[] args) {

String s = "Hello World from Syntax Error";

System.out.println(reverseWords(s));

}

}

#### **Python Solution**

def reverse\_words(s):

words = s.split()

return " ".join(reversed(words))

# Example usage

s = "Hello World from Syntax Error"

print(reverse\_words(s))

📌 **3. Check if Two Strings are Anagrams**  
**Asked in:** Amazon, Google, Microsoft

* Given two strings **S1** and **S2**, check if they are anagrams (contain the same characters in any order).

### ****C++ Solution****

#include <bits/stdc++.h>

using namespace std;

bool areAnagrams(string s1, string s2) {

if (s1.length() != s2.length()) return false;

sort(s1.begin(), s1.end());

sort(s2.begin(), s2.end());

return s1 == s2;

}

int main() {

string s1 = "listen", s2 = "silent";

if (areAnagrams(s1, s2))

cout << "Anagrams" << endl;

else

cout << "Not Anagrams" << endl;

return 0;

}

### ****Java Solution****

import java.util.Arrays;

public class AnagramCheck {

public static boolean areAnagrams(String s1, String s2) {

if (s1.length() != s2.length()) return false;

char[] arr1 = s1.toCharArray();

char[] arr2 = s2.toCharArray();

Arrays.sort(arr1);

Arrays.sort(arr2);

return Arrays.equals(arr1, arr2);

}

public static void main(String[] args) {

String s1 = "listen", s2 = "silent";

if (areAnagrams(s1, s2))

System.out.println("Anagrams");

else

System.out.println("Not Anagrams");

}

}

### ****Python Solution****

def are\_anagrams(s1, s2):

return sorted(s1) == sorted(s2)

# Example usage

s1, s2 = "listen", "silent"

if are\_anagrams(s1, s2):

print("Anagrams")

else:

print("Not Anagrams")

📌 **4. Find the Longest Palindromic Substring**  
**Asked in:** Google, Amazon, Flipkart

* Given a string **S**, find the longest substring that is a palindrome.

### ****C++ Solution (Expand Around Center)****

#include <bits/stdc++.h>

using namespace std;

string expandFromCenter(string s, int left, int right) {

while (left >= 0 && right < s.length() && s[left] == s[right]) {

left--;

right++;

}

return s.substr(left + 1, right - left - 1);

}

string longestPalindrome(string s) {

if (s.empty()) return "";

string longest = "";

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

string odd = expandFromCenter(s, i, i); // Odd-length palindrome

string even = expandFromCenter(s, i, i + 1); // Even-length palindrome

if (odd.length() > longest.length()) longest = odd;

if (even.length() > longest.length()) longest = even;

}

return longest;

}

int main() {

string s = "babad";

cout << "Longest Palindromic Substring: " << longestPalindrome(s) << endl;

return 0;

}

### ****Java Solution (Expand Around Center)****

public class LongestPalindromicSubstring {

public static String expandFromCenter(String s, int left, int right) {

while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {

left--;

right++;

}

return s.substring(left + 1, right);

}

public static String longestPalindrome(String s) {

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

String longest = "";

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

String odd = expandFromCenter(s, i, i); // Odd-length palindrome

String even = expandFromCenter(s, i, i + 1); // Even-length palindrome

if (odd.length() > longest.length()) longest = odd;

if (even.length() > longest.length()) longest = even;

}

return longest;

}

public static void main(String[] args) {

String s = "babad";

System.out.println("Longest Palindromic Substring: " + longestPalindrome(s));

}

}

### ****Python Solution (Expand Around Center)****

def expand\_from\_center(s, left, right):

while left >= 0 and right < len(s) and s[left] == s[right]:

left -= 1

right += 1

return s[left + 1:right]

def longest\_palindrome(s):

longest = ""

for i in range(len(s)):

odd = expand\_from\_center(s, i, i) # Odd-length palindrome

even = expand\_from\_center(s, i, i + 1) # Even-length palindrome

longest = max(longest, odd, even, key=len)

return longest

# Example usage

s = "babad"

print("Longest Palindromic Substring:", longest\_palindrome(s))

📌 **5. Implement the KMP Algorithm for Pattern Matching**  
**Asked in:** Amazon, Google, Microsoft

Given a string **S** and a pattern **P**, implement **KMP (Knuth-Morris-Pratt)** algorithm to find all occurrences of **P** in **S** efficiently

## **C++ Solution**

#include <bits/stdc++.h>

using namespace std;

// Function to compute LPS array

vector<int> computeLPS(string pattern) {

int m = pattern.length();

vector<int> lps(m, 0);

int len = 0, i = 1;

while (i < m) {

if (pattern[i] == pattern[len]) {

lps[i++] = ++len;

} else {

if (len) len = lps[len - 1];

else lps[i++] = 0;

}

}

return lps;

}

// KMP Pattern Matching Algorithm

void KMP(string text, string pattern) {

int n = text.length(), m = pattern.length();

vector<int> lps = computeLPS(pattern);

int i = 0, j = 0; // i -> text, j -> pattern

while (i < n) {

if (text[i] == pattern[j]) {

i++, j++;

}

if (j == m) {

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

j = lps[j - 1];

} else if (i < n && text[i] != pattern[j]) {

j ? j = lps[j - 1] : i++;

}

}

}

int main() {

string text = "ababcababcabc", pattern = "abc";

KMP(text, pattern);

return 0;

}

## **Java Solution**

import java.util.\*;

public class KMPAlgorithm {

public static int[] computeLPS(String pattern) {

int m = pattern.length();

int[] lps = new int[m];

int len = 0, i = 1;

while (i < m) {

if (pattern.charAt(i) == pattern.charAt(len)) {

lps[i++] = ++len;

} else {

len = (len != 0) ? lps[len - 1] : 0;

if (len == 0) lps[i++] = 0;

}

}

return lps;

}

public static void KMP(String text, String pattern) {

int n = text.length(), m = pattern.length();

int[] lps = computeLPS(pattern);

int i = 0, j = 0;

while (i < n) {

if (text.charAt(i) == pattern.charAt(j)) {

i++; j++;

}

if (j == m) {

System.out.println("Pattern found at index " + (i - j));

j = lps[j - 1];

} else if (i < n && text.charAt(i) != pattern.charAt(j)) {

j = (j != 0) ? lps[j - 1] : i++;

}

}

}

public static void main(String[] args) {

String text = "ababcababcabc", pattern = "abc";

KMP(text, pattern);

}

}

## **Python Solution**

def compute\_lps(pattern):

m = len(pattern)

lps = [0] \* m

length, i = 0, 1

while i < m:

if pattern[i] == pattern[length]:

length += 1

lps[i] = length

i += 1

else:

if length != 0:

length = lps[length - 1]

else:

lps[i] = 0

i += 1

return lps

def kmp(text, pattern):

n, m = len(text), len(pattern)

lps = compute\_lps(pattern)

i, j = 0, 0

while i < n:

if text[i] == pattern[j]:

i, j = i + 1, j + 1

if j == m:

print(f"Pattern found at index {i - j}")

j = lps[j - 1]

elif i < n and text[i] != pattern[j]:

j = lps[j - 1] if j != 0 else i + 1

# Example Usage

text, pattern = "ababcababcabc", "abc"

kmp(text, pattern)

📌 **6. Implement the Rabin-Karp Algorithm for String Matching**  
**Asked in:** Adobe, Google, TCS

* Given a string **S** and a pattern **P**, implement **Rabin-Karp** algorithm to search for **P** in **S** using hashing.

## **C++ Solution**

#include <bits/stdc++.h>

using namespace std;

#define PRIME 101 // A prime number for hashing

long long createHash(string str, int end) {

long long hash = 0;

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

hash = hash \* PRIME + str[i];

return hash;

}

long long recalculateHash(string str, int oldIndex, int newIndex, long long oldHash, int patternLength) {

oldHash -= str[oldIndex];

oldHash /= PRIME;

oldHash += str[newIndex] \* pow(PRIME, patternLength - 1);

return oldHash;

}

void RabinKarp(string text, string pattern) {

int n = text.length(), m = pattern.length();

long long patternHash = createHash(pattern, m);

long long textHash = createHash(text, m);

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

if (patternHash == textHash && text.substr(i, m) == pattern)

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

if (i < n - m)

textHash = recalculateHash(text, i, i + m, textHash, m);

}

}

int main() {

string text = "ababcabcabc", pattern = "abc";

RabinKarp(text, pattern);

return 0;

}

## **Java Solution**

import java.util.\*;

public class RabinKarpAlgorithm {

private static final int PRIME = 101; // Prime number for hashing

public static long createHash(String str, int end) {

long hash = 0;

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

hash = hash \* PRIME + str.charAt(i);

return hash;

}

public static long recalculateHash(String str, int oldIndex, int newIndex, long oldHash, int patternLength) {

oldHash -= str.charAt(oldIndex);

oldHash /= PRIME;

oldHash += str.charAt(newIndex) \* Math.pow(PRIME, patternLength - 1);

return oldHash;

}

public static void rabinKarp(String text, String pattern) {

int n = text.length(), m = pattern.length();

long patternHash = createHash(pattern, m);

long textHash = createHash(text, m);

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

if (patternHash == textHash && text.substring(i, i + m).equals(pattern))

System.out.println("Pattern found at index " + i);

if (i < n - m)

textHash = recalculateHash(text, i, i + m, textHash, m);

}

}

public static void main(String[] args) {

String text = "ababcabcabc", pattern = "abc";

rabinKarp(text, pattern);

}

}

## **Python Solution**

PRIME = 101 # Prime number for hashing

def create\_hash(string, end):

hash\_value = 0

for i in range(end):

hash\_value = hash\_value \* PRIME + ord(string[i])

return hash\_value

def recalculate\_hash(string, old\_index, new\_index, old\_hash, pattern\_length):

old\_hash -= ord(string[old\_index])

old\_hash //= PRIME

old\_hash += ord(string[new\_index]) \* (PRIME \*\* (pattern\_length - 1))

return old\_hash

def rabin\_karp(text, pattern):

n, m = len(text), len(pattern)

pattern\_hash = create\_hash(pattern, m)

text\_hash = create\_hash(text, m)

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

if pattern\_hash == text\_hash and text[i:i + m] == pattern:

print(f"Pattern found at index {i}")

if i < n - m:

text\_hash = recalculate\_hash(text, i, i + m, text\_hash, m)

# Example Usage

text, pattern = "ababcabcabc", "abc"

rabin\_karp(text, pattern)

📌 **7. Find the Longest Common Prefix (LCP) Among Strings**  
**Asked in:** Microsoft, Google, Amazon

* Given an array of strings, find the longest common prefix among them.

## **C++ Solution (Horizontal Scanning)**

#include <bits/stdc++.h>

using namespace std;

string longestCommonPrefix(vector<string>& strs) {

if (strs.empty()) return "";

string prefix = strs[0];

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

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

prefix = prefix.substr(0, prefix.size() - 1);

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

}

}

return prefix;

}

int main() {

vector<string> strs = {"flower", "flow", "flight"};

cout << "Longest Common Prefix: " << longestCommonPrefix(strs) << endl;

return 0;

}

## **Java Solution (Horizontal Scanning)**

import java.util.\*;

public class LCP {

public static String longestCommonPrefix(String[] strs) {

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

String prefix = strs[0];

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

while (!strs[i].startsWith(prefix)) {

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

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

}

}

return prefix;

}

public static void main(String[] args) {

String[] strs = {"flower", "flow", "flight"};

System.out.println("Longest Common Prefix: " + longestCommonPrefix(strs));

}

}

## **Python Solution (Horizontal Scanning)**

def longest\_common\_prefix(strs):

if not strs:

return ""

prefix = strs[0]

for s in strs[1:]:

while not s.startswith(prefix):

prefix = prefix[:-1]

if not prefix:

return ""

return prefix

# Example Usage

strs = ["flower", "flow", "flight"]

print("Longest Common Prefix:", longest\_common\_prefix(strs))

📌 **8. Find All Occurrences of a Pattern in a Text Using Z-Algorithm**  
**Asked in:** Flipkart, Amazon, TCS

* Given a text **S** and pattern **P**, use **Z-Algorithm** to efficiently find all occurrences of **P** in **S**.

## **C++ Solution**

#include <bits/stdc++.h>

using namespace std;

// Function to compute the Z-array

vector<int> computeZ(string s) {

int n = s.size();

vector<int> Z(n, 0);

int L = 0, R = 0;

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

if (i <= R)

Z[i] = min(R - i + 1, Z[i - L]);

while (i + Z[i] < n && s[Z[i]] == s[i + Z[i]])

Z[i]++;

if (i + Z[i] - 1 > R) {

L = i;

R = i + Z[i] - 1;

}

}

return Z;

}

// Function to find all occurrences of a pattern in a text using Z-Algorithm

vector<int> findPattern(string text, string pattern) {

string combined = pattern + "$" + text;

vector<int> Z = computeZ(combined);

vector<int> occurrences;

int patternLength = pattern.size();

for (int i = patternLength + 1; i < Z.size(); i++) {

if (Z[i] == patternLength)

occurrences.push\_back(i - patternLength - 1);

}

return occurrences;

}

int main() {

string text = "ababcababcab";

string pattern = "abc";

vector<int> result = findPattern(text, pattern);

for (int index : result)

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

return 0;

}

## **Java Solution**

import java.util.\*;

public class ZAlgorithm {

// Compute the Z-array

private static int[] computeZ(String s) {

int n = s.length();

int[] Z = new int[n];

int L = 0, R = 0;

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

if (i <= R)

Z[i] = Math.min(R - i + 1, Z[i - L]);

while (i + Z[i] < n && s.charAt(Z[i]) == s.charAt(i + Z[i]))

Z[i]++;

if (i + Z[i] - 1 > R) {

L = i;

R = i + Z[i] - 1;

}

}

return Z;

}

// Find all occurrences of a pattern in a text using Z-Algorithm

public static List<Integer> findPattern(String text, String pattern) {

String combined = pattern + "$" + text;

int[] Z = computeZ(combined);

List<Integer> occurrences = new ArrayList<>();

int patternLength = pattern.length();

for (int i = patternLength + 1; i < Z.length; i++) {

if (Z[i] == patternLength)

occurrences.add(i - patternLength - 1);

}

return occurrences;

}

public static void main(String[] args) {

String text = "ababcababcab";

String pattern = "abc";

List<Integer> result = findPattern(text, pattern);

for (int index : result)

System.out.println("Pattern found at index: " + index);

}

}

## **Python Solution**

def compute\_z(s):

n = len(s)

Z = [0] \* n

L, R = 0, 0

for i in range(1, n):

if i <= R:

Z[i] = min(R - i + 1, Z[i - L])

while i + Z[i] < n and s[Z[i]] == s[i + Z[i]]:

Z[i] += 1

if i + Z[i] - 1 > R:

L, R = i, i + Z[i] - 1

return Z

def find\_pattern(text, pattern):

combined = pattern + "$" + text

Z = compute\_z(combined)

pattern\_length = len(pattern)

occurrences = []

for i in range(pattern\_length + 1, len(Z)):

if Z[i] == pattern\_length:

occurrences.append(i - pattern\_length - 1)

return occurrences

# Example usage

text = "ababcababcab"

pattern = "abc"

result = find\_pattern(text, pattern)

for index in result:

print(f"Pattern found at index: {index}")

📌 **9. Build a Suffix Array for a Given String**  
**Asked in:** Google, Microsoft, Amazon

* Construct a **suffix array** for a given string **S**, which helps in efficient pattern matching.

## **C++ Solution**

#include <bits/stdc++.h>

using namespace std;

// Function to build a suffix array

vector<int> buildSuffixArray(const string &s) {

int n = s.size();

vector<int> suffixArray(n), rank(n), tempRank(n);

// Initial ranking based on single characters

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

suffixArray[i] = i;

rank[i] = s[i];

}

// Sort by doubling length (2^k)

for (int length = 1; length < n; length \*= 2) {

auto cmp = [&](int a, int b) {

if (rank[a] != rank[b])

return rank[a] < rank[b];

int ra = (a + length < n) ? rank[a + length] : -1;

int rb = (b + length < n) ? rank[b + length] : -1;

return ra < rb;

};

sort(suffixArray.begin(), suffixArray.end(), cmp);

// Update ranks

tempRank[suffixArray[0]] = 0;

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

tempRank[suffixArray[i]] = tempRank[suffixArray[i - 1]] + cmp(suffixArray[i - 1], suffixArray[i]);

}

rank = tempRank;

}

return suffixArray;

}

// Driver Code

int main() {

string s = "banana";

vector<int> suffixArray = buildSuffixArray(s);

cout << "Suffix Array: ";

for (int i : suffixArray)

cout << i << " ";

cout << endl;

return 0;

}

## **Java Solution**

import java.util.\*;

public class SuffixArray {

// Function to build suffix array

public static int[] buildSuffixArray(String s) {

int n = s.length();

Integer[] suffixArray = new Integer[n];

int[] rank = new int[n];

int[] tempRank = new int[n];

// Initial ranking based on single characters

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

suffixArray[i] = i;

rank[i] = s.charAt(i);

}

// Sort suffixes by 2^k length

for (int length = 1; length < n; length \*= 2) {

final int len = length;

Arrays.sort(suffixArray, (a, b) -> {

if (rank[a] != rank[b])

return Integer.compare(rank[a], rank[b]);

int ra = (a + len < n) ? rank[a + len] : -1;

int rb = (b + len < n) ? rank[b + len] : -1;

return Integer.compare(ra, rb);

});

// Update ranks

tempRank[suffixArray[0]] = 0;

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

tempRank[suffixArray[i]] = tempRank[suffixArray[i - 1]] +

(rank[suffixArray[i - 1]] != rank[suffixArray[i]] ||

((suffixArray[i - 1] + length < n ? rank[suffixArray[i - 1] + length] : -1) !=

(suffixArray[i] + length < n ? rank[suffixArray[i] + length] : -1)) ? 1 : 0);

}

rank = Arrays.copyOf(tempRank, n);

}

return Arrays.stream(suffixArray).mapToInt(Integer::intValue).toArray();

}

// Driver Code

public static void main(String[] args) {

String s = "banana";

int[] suffixArray = buildSuffixArray(s);

System.out.print("Suffix Array: ");

for (int index : suffixArray) {

System.out.print(index + " ");

}

}

}

## **Python Solution**

def build\_suffix\_array(s):

n = len(s)

suffix\_array = list(range(n))

rank = list(map(ord, s))

temp\_rank = [0] \* n

# Sort suffixes by 2^k length

length = 1

while length < n:

suffix\_array.sort(key=lambda i: (rank[i], rank[i + length] if i + length < n else -1))

# Update ranks

temp\_rank[suffix\_array[0]] = 0

for i in range(1, n):

prev = suffix\_array[i - 1]

curr = suffix\_array[i]

temp\_rank[curr] = temp\_rank[prev] + (

rank[prev] != rank[curr] or

(rank[prev + length] if prev + length < n else -1) !=

(rank[curr + length] if curr + length < n else -1)

)

rank = temp\_rank[:]

length \*= 2

return suffix\_array

# Example usage

s = "banana"

suffix\_array = build\_suffix\_array(s)

print("Suffix Array:", suffix\_array)

📌 **10. Find the Longest Repeated Substring Using Suffix Arrays & LCP**  
**Asked in:** Google, Facebook, Microsoft

* Given a string **S**, find the longest repeated substring using **suffix arrays** and **LCP (Longest Common Prefix)**.

## **C++ Solution**

#include <bits/stdc++.h>

using namespace std;

// Function to build a suffix array

vector<int> buildSuffixArray(const string &s) {

int n = s.size();

vector<int> suffixArray(n), rank(n), tempRank(n);

// Initial ranking based on single characters

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

suffixArray[i] = i;

rank[i] = s[i];

}

// Sort suffixes by doubling length (2^k)

for (int length = 1; length < n; length \*= 2) {

auto cmp = [&](int a, int b) {

if (rank[a] != rank[b])

return rank[a] < rank[b];

int ra = (a + length < n) ? rank[a + length] : -1;

int rb = (b + length < n) ? rank[b + length] : -1;

return ra < rb;

};

sort(suffixArray.begin(), suffixArray.end(), cmp);

// Update ranks

tempRank[suffixArray[0]] = 0;

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

tempRank[suffixArray[i]] = tempRank[suffixArray[i - 1]] + cmp(suffixArray[i - 1], suffixArray[i]);

}

rank = tempRank;

}

return suffixArray;

}

// Function to build the LCP (Longest Common Prefix) array

vector<int> buildLCPArray(const string &s, const vector<int> &suffixArray) {

int n = s.size();

vector<int> lcp(n, 0), rank(n, 0);

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

rank[suffixArray[i]] = i;

}

int h = 0;

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

if (rank[i] > 0) {

int j = suffixArray[rank[i] - 1];

while (i + h < n && j + h < n && s[i + h] == s[j + h]) {

h++;

}

lcp[rank[i]] = h;

if (h > 0) h--;

}

}

return lcp;

}

// Function to find the longest repeated substring

string longestRepeatedSubstring(const string &s) {

vector<int> suffixArray = buildSuffixArray(s);

vector<int> lcp = buildLCPArray(s, suffixArray);

int maxLength = 0, index = 0;

for (int i = 1; i < lcp.size(); i++) {

if (lcp[i] > maxLength) {

maxLength = lcp[i];

index = suffixArray[i];

}

}

return s.substr(index, maxLength);

}

// Driver Code

int main() {

string s = "banana";

string result = longestRepeatedSubstring(s);

cout << "Longest Repeated Substring: " << result << endl;

return 0;

}

## **Java Solution**

import java.util.\*;

public class LongestRepeatedSubstring {

// Function to build suffix array

public static int[] buildSuffixArray(String s) {

int n = s.length();

Integer[] suffixArray = new Integer[n];

int[] rank = new int[n];

int[] tempRank = new int[n];

// Initial ranking based on single characters

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

suffixArray[i] = i;

rank[i] = s.charAt(i);

}

// Sort suffixes by 2^k length

for (int length = 1; length < n; length \*= 2) {

final int len = length;

Arrays.sort(suffixArray, (a, b) -> {

if (rank[a] != rank[b])

return Integer.compare(rank[a], rank[b]);

int ra = (a + len < n) ? rank[a + len] : -1;

int rb = (b + len < n) ? rank[b + len] : -1;

return Integer.compare(ra, rb);

});

// Update ranks

tempRank[suffixArray[0]] = 0;

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

tempRank[suffixArray[i]] = tempRank[suffixArray[i - 1]] +

(rank[suffixArray[i - 1]] != rank[suffixArray[i]] ||

((suffixArray[i - 1] + length < n ? rank[suffixArray[i - 1] + length] : -1) !=

(suffixArray[i] + length < n ? rank[suffixArray[i] + length] : -1)) ? 1 : 0);

}

rank = Arrays.copyOf(tempRank, n);

}

return Arrays.stream(suffixArray).mapToInt(Integer::intValue).toArray();

}

// Function to find the longest repeated substring

public static String longestRepeatedSubstring(String s) {

int[] suffixArray = buildSuffixArray(s);

int n = s.length();

int[] lcp = new int[n];

int[] rank = new int[n];

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

rank[suffixArray[i]] = i;

}

int h = 0, maxLength = 0, index = 0;

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

if (rank[i] > 0) {

int j = suffixArray[rank[i] - 1];

while (i + h < n && j + h < n && s.charAt(i + h) == s.charAt(j + h)) {

h++;

}

lcp[rank[i]] = h;

if (h > 0) h--;

if (lcp[rank[i]] > maxLength) {

maxLength = lcp[rank[i]];

index = suffixArray[rank[i]];

}

}

}

return s.substring(index, index + maxLength);

}

// Driver Code

public static void main(String[] args) {

String s = "banana";

String result = longestRepeatedSubstring(s);

System.out.println("Longest Repeated Substring: " + result);

}

}

## **Python Solution**

def build\_suffix\_array(s):

n = len(s)

suffix\_array = sorted(range(n), key=lambda i: s[i:])

return suffix\_array

def build\_lcp\_array(s, suffix\_array):

n = len(s)

rank = [0] \* n

lcp = [0] \* n

for i, suffix in enumerate(suffix\_array):

rank[suffix] = i

h = 0

for i in range(n):

if rank[i] > 0:

j = suffix\_array[rank[i] - 1]

while i + h < n and j + h < n and s[i + h] == s[j + h]:

h += 1

lcp[rank[i]] = h

if h > 0:

h -= 1

return lcp

def longest\_repeated\_substring(s):

suffix\_array = build\_suffix\_array(s)

lcp = build\_lcp\_array(s, suffix\_array)

max\_length = max(lcp)

index = lcp.index(max\_length)

return s[suffix\_array[index]:suffix\_array[index] + max\_length]

# Example usage

s = "banana"

print("Longest Repeated Substring:", longest\_repeated\_substring(s))

**🎉 Congratulations! 🎉**

You've successfully completed this PDF and taken one step closer to mastering coding challenges! Your dedication and hard work will surely pay off. Keep pushing your limits and sharpening your skills! 💻🔥

For more coding content, problem-solving techniques, and valuable insights, follow **@SYNTAX\_ERROR** on Instagram! 🚀📚

**Keep coding, keep growing! 🚀🎯**