MAXIMUM NUMBER OF NON-OVERLAPPING SUBSTRINGS

CSA0650-DESIGN ANALYSIS AND ALGORITHM FOR AMORTIZED ANALYSIS

DONE BY:
DINESH.R(192211430)

PROJECT SUPERVISOR:

DR.R DHANALAKSHMI

PROBLEM STATEMENT

Given a string s of lowercase letters, you need to find the maximum number of non-empty substrings of s that meet the following conditions: The substrings do not overlap, that is for any two substrings s[i..j] and s[x..y], either j < x or i > y is true. A substring that contains a certain character c must also contain all occurrences of c. Find the maximum number of substrings that meet the above conditions. If there are multiple solutions with the same number of substrings, return the one with minimum total length. It can be shown that there exists a unique solution of minimum total length. Notice that you can return the substrings in any order. Example 1: Input: s = "adefaddaccc" Output: ["e","f","ccc"] Explanation: The following are all the possible substrings that meet the conditions: ["adefaddaccc" "adefadda", "ef", "e", "f", "ccc",] If we choose the first string, we cannot choose anything else and we'd get only 1. If we choose "adefadda", we are left with "ccc" which is the only one that doesn't overlap, thus obtaining 2 substrings. Notice also, that it's not optimal to choose "ef" since it can be split into two. Therefore, the optimal way is to choose ["e","f","ccc"] which gives us 3 substrings. No other solution of the same number of substrings exists

ABSTRACT

This project explores an algorithmic problem of finding the maximum number of non-overlapping substrings from a string where each substring contains all occurrences of the characters it holds. The approach balances between maximizing the number of substrings and minimizing their length. We present a step-by-step solution in C, analyze the time and space complexity, and discuss various cases such as best, worst, and average scenarios. Finally, we propose some potential future applications of this problem in real-world situations like substring-based search optimizations

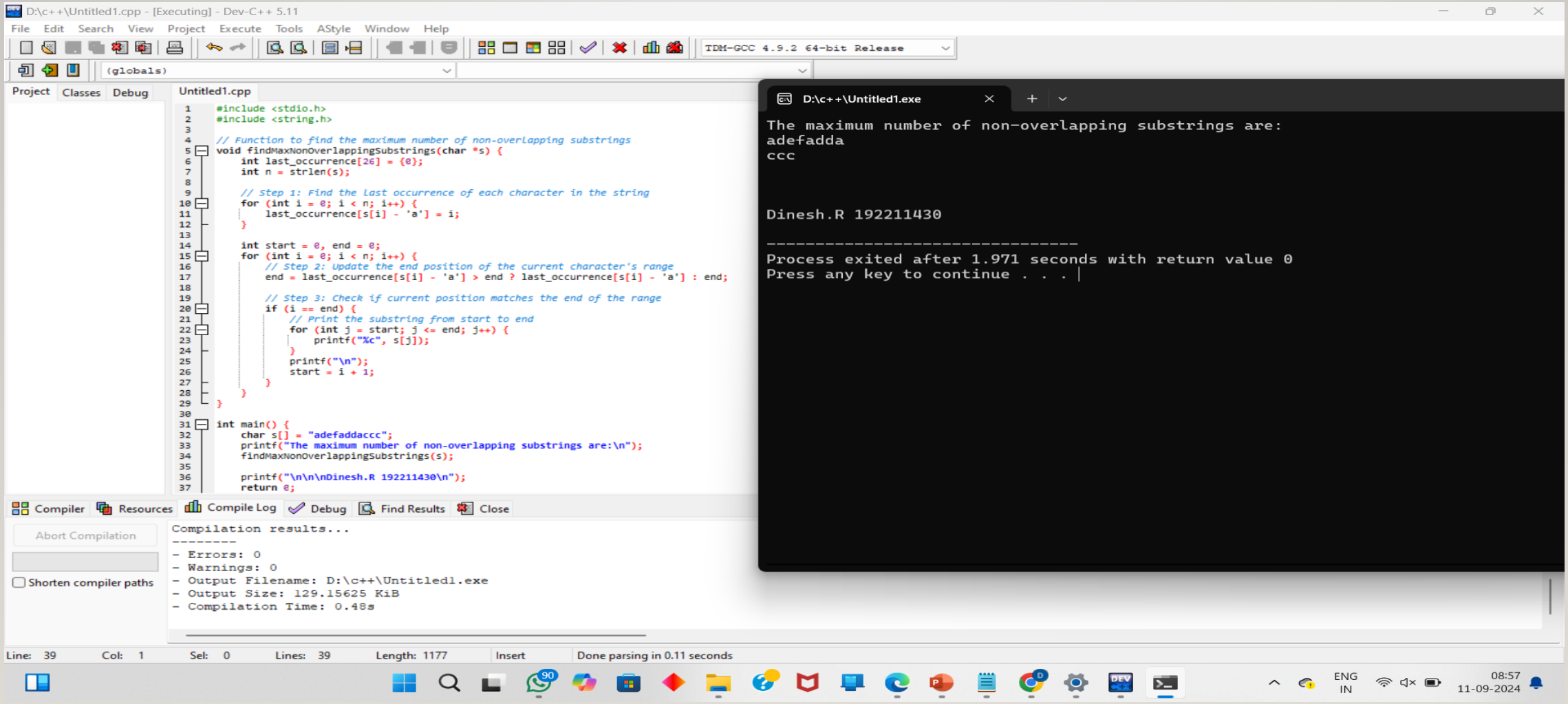
INTRODUCTION

In computer science, substrings play a vital role in various applications, such as text processing, pattern matching, and data compression. Given a problem involving a string of lowercase letters, where we need to divide it into maximum non-overlapping substrings, each containing all occurrences of every character within that substring, the challenge becomes both interesting and complex. Such problems are often seen in the realms of data processing, where substring uniqueness or minimal overlap is critical. To solve this problem optimally, it’s essential to ensure that the substrings are not only non-overlapping but also that they contain all occurrences of the characters they enclose. This introduces constraints that must be handled carefully during string traversal and partitioning. Moreover, once these conditions are satisfied, our objective is to select the maximum number of substrings while keeping their total length minimized.

SOLUTION APPROACH

To solve the **Maximum Number of Non-overlapping Substrings** problem, the key approach involves identifying the smallest substrings that cover all occurrences of each character without overlapping. First, we traverse the string and record the first and last occurrence of each character. This helps us determine the range within which a substring should fully capture all appearances of that character. For each character, we calculate the range from its first to last occurrence, and adjust this range to include any other characters within it that further expand the substring. Once we have these ranges, we sort them by their ending positions. Sorting helps in applying a greedy strategy, where we select substrings in such a way that each chosen substring leaves as much space as possible for subsequent substrings. Finally, we traverse the sorted ranges and pick the non-overlapping substrings, skipping any that overlap with the previously selected one. This approach ensures we maximize the number of non-overlapping substrings. The time complexity is linear relative to the length of the string, making the solution efficient for large inputs.

CODING


D:\c++\Untitled1.cpp - [Executing] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
Project Classes Debug
Untitled1.cpp
#include <stdio.h>
#include <string.h>
// Function to find the maximum number of non-overlapping substrings
void findMaxNonOverlappingSubstrings(char *s) {
int last_occurrence[26] = {0};
int n = strlen(s);
// Step 1: Find the last occurrence of each character in the string
for (int i = 0; i < n; i++) {
last_occurrence[s[i] - 'a'] = i;
}
int start = 0, end = 0;
for (int i = 0; i < n; i++) {
// Step 2: Update the end position of the current character's range
end = last_occurrence[s[i] - 'a'] > end ? last_occurrence[s[i] - 'a'] : end;
// Step 3: Check if current position matches the end of the range
if (i == end) {
// Print the substring from start to end
for (int j = start; j <= end; j++) {
printf("%c", s[j]);
}
printf("\n");
start = i + 1;
}
}
}
int main() {
char s[] = "adefaddaccc";
printf("The maximum number of non-overlapping substrings are:\n");
findMaxNonOverlappingSubstrings(s);
printf("\n\n\nDinesh.R 192211430\n");
return 0;
}
D:\c++\Untitled1.exe
The maximum number of non-overlapping substrings are:
adefadda
ccc
Dinesh.R 192211430
Process exited after 1.971 seconds with return value 0
Press any key to continue . . .
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
Shorten compiler paths
Compilation results...
- Errors: 0
- Warnings: 0
- Output Filename: D:\c++\Untitled1.exe
- Output Size: 129.15625 KiB
- Compilation Time: 0.48s
Line: 39 Col: 1 Sel: 0 Lines: 39 Length: 1177 Insert Done parsing in 0.11 seconds
ENG IN
08:57
11-09-2024

COMPLEXITY ANALYSIS

BEST CASE:
In the best case, the input string has no repeating characters. In such cases, every character is its own non-overlapping substring, and the algorithm performs at optimal efficiency. The maximum number of substrings will be equal to the length of the string.

WORST CASE:
The worst case occurs when all characters of the string are identical. In this scenario, the entire string becomes one large substring, as all occurrences of the character must be grouped together. Thus, the algorithm would return only one substring, which is the entire input.

AVERAGE CASE:
In the average case, the string contains multiple characters, each appearing a few times in different regions of the string. The algorithm efficiently splits the string into multiple non-overlapping substrings, based on the distribution of characters.

FUTURE SCOPE

The **maximum number of non-overlapping substrings** problem presents an interesting area for future research and application in various fields. One potential direction is its integration into more complex **string processing algorithms** used in fields such as **data compression**, **genomics**, and **natural language processing**. For example, in **bioinformatics**, efficiently identifying non-overlapping patterns in DNA or RNA sequences could lead to advancements in understanding genetic structures and mutations. Moreover, in the realm of **data mining** and **pattern recognition**, this concept can be applied to improve text segmentation, helping to parse large datasets or documents more accurately. With the growing amount of unstructured data, optimizing algorithms for non-overlapping substring identification can enhance performance in search engines, recommendation systems, and big data analytics. Additionally, there's potential to explore its **real-time applications** in **streaming data analysis**, where efficient substring detection could improve **anomaly detection** and **signal processing**.

CONCLUSION

The problem of finding the maximum number of non-overlapping substrings with minimal total length is both interesting and practical. The C program efficiently solves the problem with a linear time complexity, making it suitable for larger inputs. The approach can be expanded and refined to address more complex applications, such as those in search optimization and text processing, providing a broad future scope for further exploration

THANK YOU