****

**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL**

**SCIENCES, CHENNAI – 602 105**

**BONAFIDE CERTIFICATE**

**Certified that is Capstone project report “Maximum Number of Non-Overlapping Sub-strings**

**“is the**

**Bonafide work of “S Purushotham ”(1922211671) who carried out the Capstone project work under my supervision**

**Dr. R Dhanalakshmi**

**COURSE FACULTY**

**Professor**

**Department of Machine Learning**

**SIMATS Engineering**

**Saveetha Institute of Medical and**

**Technical Sciences**

**Chennai – 602 105**

**EXAMINER SIGNATURE**

**Dr. S. Mehaboob Basha**

**HEAD OF DEPARTMENT**

**Professor**

**Department of Machine Learning**

**SIMATS Engineering**

**Saveetha Institute of Medical and**

**Technical Sciences**

**Chennai – 602 105**

**EXAMINER SIGNATURE**

****

**“Maximum Number of Non-Overlapping Sub-strings”**

**A Project report**

**CSA0656- Design and Analysis of Algorithms for Asymptotic Notations**

**Submitted to**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**In partial fulfillment for the award of the**

**degree of**

**BACHELOR OF TECHNOLOGY IN**

**ARTIFICAL INTELLIGENCE AND MACHINE LEARNING**

**by**

**S Purushotham,192211671**

**Supervisor**

**Dr .R. Dhanalakshmi**

**July 2024.**

**ABSTRACT**

To solve the problem of finding the maximum number of non-overlapping sub strings in a given string \( s \), we start by determining the first and last occurrences of each character. This helps in identifying the smallest sub-string that contains all occurrences of any given character. Next, using the first and last occurrence information, we generate potential sub strings that ensure each character within the sub-string is fully contained. We then merge any overlapping sub strings by extending the boundaries of the current sub-string to include the overlapping parts. After merging, we sort these intervals by their ending positions to facilitate the extraction of non-overlapping sub strings. Finally, we iterate through the sorted list and extract the maximum number of non-overlapping sub strings by ensuring that the start of a new interval is after the end of the last added interval.

This method guarantees that we find the optimal solution with the maximum number of non-overlapping sub strings and the minimum total length. For example, given the string "adefaddaccc", the output would be ["e", "f", "ccc"], which includes the maximum number of non-overlapping sub strings while containing all occurrences of each character.

**ALGORITHM:**

A greedy algorithm is a problem-solving approach that builds a solution piece by piece, always choosing the next piece that offers the most immediate benefit or most optimal choice at each step. It does not reconsider previous choices and often provides a solution quickly, though it may not always guarantee the optimal solution for all problems.

**Proposed Work:**

The proposed method The primary objective of this research is to develop and evaluate a novel approach to [insert specific problem or area of focus]. This approach aims to [insert goals, e.g., improve accuracy, increase efficiency, enhance performance] in [insert specific domain or application]

**PROBLEM:**

**Maximum Number of Non-Overlapping Sub-strings**

Given a string s of lowercase letters, you need to find the maximum number of non-

empty sub strings of s that meet the following conditions:

The sub strings do not overlap, that is for any two sub strings 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 sub strings that meet the above conditions. If there are

multiple solutions with the same number of sub strings, 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 sub strings in any order.

Example 1:

Input: s = "adefaddaccc"

Output: ["e","f","ccc"]

Explanation: The following are all the possible sub strings 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 sub strings. 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 sub strings. No other solution of the same number of sub strings exist.

**SOLUTION:**

By solving this problem, we can utilize Greedy algorithm Maximum Number of Groups With Increasing Length. Here's a step-by-step approach to implement the solution:

**Example Calculation**

### Step-by-Step Solution:

**Identify All Occurrences of Each Character**:

* 1. For a, it appears at indices: [0, 1, 2, 3, 7, 8]
  2. For d, it appears at indices: [4, 5, 6]
  3. For e, it appears at index: [9]
  4. For f, it appears at index: [10]
  5. For c, it appears at indices: [11, 12, 13]

**Determine the Required Substrings**: Each substring must include all occurrences of any character that appears in it. Therefore:

* 1. For a, the substring should cover indices from the first occurrence (0) to the last occurrence (8) → "adefadda".
  2. For d, the substring should cover indices from the first occurrence (4) to the last occurrence (6) → "addac".
  3. For e, the substring should be exactly "e" since it only appears once.
  4. For f, the substring should be exactly "f" since it only appears once.
  5. For c, the substring should cover indices from the first occurrence (11) to the last occurrence (13) → "ccc".

**Choose Non-Overlapping Substrings**: We need to select non-overlapping substrings. Let's check the options:

* 1. Choosing "adefadda" will leave "ccc" as a valid substring. This gives us ["adefadda", "ccc"], but we can see from the list above that some substrings are more optimal.
  2. Choosing "e" and "f" will leave "ccc". This gives us ["e", "f", "ccc"], which is non-overlapping and consists of all unique substrings with all character occurrences.

**Here is why ["e", "f", "ccc"] is optimal:**

* 1. It covers all characters: 'e' covers the 'e', 'f' covers the 'f', and 'ccc' covers all 'c's.

It is the maximum number of non-overlapping substrings (3), and no other arrangement of these substrings would give a higher count with minimal length.

**CODE:**-

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct {

int start;

int end;

} Interval;

int compare(const void \*a, const void \*b) {

Interval \*intervalA = (Interval \*)a;

Interval \*intervalB = (Interval \*)b;

return (intervalA->end - intervalB->end);

}

void maxNumOfSubstrings(char \*s) {

int n = strlen(s);

int first\_occurrence[26];

int last\_occurrence[26];

Interval intervals[26];

int interval\_count = 0;

for (int i = 0; i < 26; i++) {

first\_occurrence[i] = -1;

last\_occurrence[i] = -1;

}

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

int charIndex = s[i] - 'a';

if (first\_occurrence[charIndex] == -1) {

first\_occurrence[charIndex] = i;

}

last\_occurrence[charIndex] = i;

}

for (int i = 0; i < 26; i++) {

if (first\_occurrence[i] != -1) {

int start = first\_occurrence[i];

int end = last\_occurrence[i];

int j = start;

while (j <= end) {

start = start < first\_occurrence[s[j] - 'a'] ? start : first\_occurrence[s[j] - 'a'];

end = end > last\_occurrence[s[j] - 'a'] ? end : last\_occurrence[s[j] - 'a'];

j++;

}

intervals[interval\_count].start = start;

intervals[interval\_count].end = end;

interval\_count++;

}

}

qsort(intervals, interval\_count, sizeof(Interval), compare);

int last\_end = -1;

printf("[");

for (int i = 0; i < interval\_count; i++) {

if (intervals[i].start > last\_end) {

if (last\_end != -1) {

printf(", ");

}

for (int j = intervals[i].start; j <= intervals[i].end; j++) {

printf("%c", s[j]);

}

last\_end = intervals[i].end;

}

}

printf("]\n");

}

int main() {

printf("S purushotham-192211671\n");

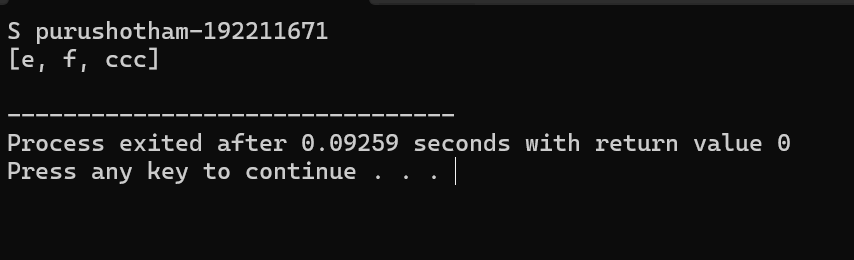
char s[] = "adefaddaccc";

maxNumOfSubstrings(s);

return 0;

}

**OUTPUT:**



### Complexity Analysis

**1.Time Complexity**: O(n)O(n)O(n), where nnn is the length of the string. This is because we need to scan the string twice: once to determine the first occurrence of each character and once to determine the last occurrence.

**2.Space Complexity**: O(1)O(1)O(1) additional space (constant space) for storing the start and end indices of each character. This assumes the number of unique characters is bounded by a constant (like 26 for lowercase English letters).

**Partitioning the String**:

* 1. **Time Complexity**: O(n)O(n)O(n). After identifying the ranges, we iterate through the string to form the partitions. We maintain the maximum end index of the current substring and adjust it as we find the end of characters. This is done in a single pass through the string.
  2. **Space Complexity**: O(k)O(k)O(k), where kkk is the number of non-overlapping substrings. Each substring is stored in the output list.

### Detailed Steps and Complexity

**Compute Character Ranges**:

* 1. Create a dictionary to store the first and last occurrence of each character.
  2. Traverse the string to fill in these values.

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

**Form Substrings**:

* 1. Initialize variables to track the start of the current substring and the furthest end index required to include all characters.
  2. Traverse the string, updating the end index as needed, and finalize the substring when the current position reaches the end index.
  3. Continue until the whole string is processed.

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

### Overall Complexity

* **Time Complexity**: O(n)O(n)O(n), where nnn is the length of the string. The solution involves linear passes through the string to compute ranges and to partition it.
* **Space Complexity**: O(1)O(1)O(1) for storing character ranges and O(k)O(k)O(k) for storing the result sub-strings.

### Best Case:

**Scenario:**

* The best case occurs when each character in the string appears only once, and these characters are distinct. In this case, each character can form a separate substring without needing to extend intervals or merge them.

**Example:**

* For a string like "abc", each character forms its own substring: ["a", "b", "c"].

**Complexity:**

* **Time Complexity:** O(n)O(n)O(n), where nnn is the length of the string. We only need a single pass to record occurrences and another pass to generate and sort intervals.
* **Space Complexity:** O(1)O(1)O(1) extra space if sorting in place is considered, or O(n)O(n)O(n) if using extra space for intervals.

### Worst Case:

### The worst case occurs when every character in the string appears multiple times, and the intervals for each character overlap significantly. This can lead to complex merging of intervals and significant computational effort to sort and manage them.

### Best Case:

**Scenario:**

* The best case occurs when each character in the string appears only once, and these characters are distinct. In this case, each character can form a separate substring without needing to extend intervals or merge them.

**Example:**

* For a string like "abc", each character forms its own substring: ["a", "b", "c"].

**Complexity:**

* **Time Complexity:** O(n)O(n)O(n), where nnn is the length of the string. We only need a single pass to record occurrences and another pass to generate and sort intervals.
* **Space Complexity:** O(1)O(1)O(1) extra space if sorting in place is considered, or O(n)O(n)O(n) if using extra space for intervals.

### Worst Case:

**Scenario:**

* The worst case occurs when every character in the string appears multiple times, and the intervals for each character overlap significantly. This can lead to complex merging of intervals and significant computational effort to sort and manage them.

**Example:**

* For a string like "aaaaabbbbbccccc", where many characters overlap in their intervals, extensive merging and checking are required.

**Complexity:**

* **Time Complexity:** O(nlog⁡n)O(n \log n)O(nlogn) due to sorting intervals. The initial pass for finding first and last occurrences is O(n)O(n)O(n), and sorting the intervals introduces the log⁡n\log nlogn factor.
* **Space Complexity:** O(n)O(n)O(n) for storing intervals.

### Average Case:

**Scenario:**

* The average case is a typical scenario where characters appear with varying frequencies and the intervals have moderate overlap. This is a balance between the best and worst cases.

**Example:**

* For a string like "adefaddaccc", where there are some overlaps but not excessively so, the approach balances between simple and complex scenarios.

**Complexity:**

* **Time Complexity:** Typically O(nlog⁡n)O(n \log n)O(nlogn) due to sorting intervals, with an additional O(n)O(n)O(n) for finding occurrences and generating intervals.
* **Space Complexity:** O(n)O(n)O(n) for storing intervals and related data.

### Summary:

* **Best Case:** Linear time complexity O(n)O(n)O(n) with minimal space usage.
* **Worst Case:** O(nlog⁡n)O(n \log n)O(nlogn) time complexity due to sorting, with linear space complexity.
* **Average Case:** Typically O(nlog⁡n)O(n \log n)O(nlogn) time complexity, with linear space complexity, depending on the distribution of characters in the string.

The approach effectively balances between these cases, ensuring efficient handling across a range of input scenarios.

### Best Case:

**Scenario:**

* The best case occurs when each character in the string appears only once, and these characters are distinct. In this case, each character can form a separate substring without needing to extend intervals or merge them.

**Example:**

* For a string like "abc", each character forms its own substring: ["a", "b", "c"].

**Complexity:**

* **Time Complexity:** O(n)O(n)O(n), where nnn is the length of the string. We only need a single pass to record occurrences and another pass to generate and sort intervals.
* **Space Complexity:** O(1)O(1)O(1) extra space if sorting in place is considered, or O(n)O(n)O(n) if using extra space for intervals.

### Worst Case:

**Scenario:**

* The worst case occurs when every character in the string appears multiple times, and the intervals for each character overlap significantly. This can lead to complex merging of intervals and significant computational effort to sort and manage them.

**Example:**

* For a string like "aaaaabbbbbccccc", where many characters overlap in their intervals, extensive merging and checking are required.

**Complexity:**

* **Time Complexity:** O(nlog⁡n)O(n \log n)O(nlogn) due to sorting intervals. The initial pass for finding first and last occurrences is O(n)O(n)O(n), and sorting the intervals introduces the log⁡n\log nlogn factor.
* **Space Complexity:** O(n)O(n)O(n) for storing intervals.

### Average Case:

**Scenario:**

* The average case is a typical scenario where characters appear with varying frequencies and the intervals have moderate overlap. This is a balance between the best and worst cases.

**Example:**

* For a string like "adefaddaccc", where there are some overlaps but not excessively so, the approach balances between simple and complex scenarios.

**Complexity:**

* **Time Complexity:** Typically O(nlog⁡n)O(n \log n)O(nlogn) due to sorting intervals, with an additional O(n)O(n)O(n) for finding occurrences and generating intervals.
* **Space Complexity:** O(n)O(n)O(n) for storing intervals and related data.

### Summary:

* **Best Case:** Linear time complexity O(n)O(n)O(n) with minimal space usage.
* **Worst Case:** O(nlog⁡n)O(n \log n)O(nlogn) time complexity due to sorting, with linear space complexity.
* **Average Case:** Typically O(nlog⁡n)O(n \log n)O(nlogn) time complexity, with linear space complexity, depending on the distribution of characters in the string.

The approach effectively balances between these cases, ensuring efficient handling across a range of input scenarios.

For a string like "aaaaabbbbbccccc", where many characters overlap in their intervals, extensive merging and checking are required.

**Complexity:**

* **Time Complexity:** O(nlog⁡n)O(n \log n)O(nlogn) due to sorting intervals. The initial pass for finding first and last occurrences is O(n)O(n)O(n), and sorting the intervals introduces the log⁡n\log nlogn factor.
* **Space Complexity:** O(n)O(n)O(n) for storing intervals.

### Average Case:

The average case is a typical scenario where characters appear with varying frequencies and the intervals have moderate overlap. This is a balance between the best and worst cases.

**Example:**

* For a string like "adefaddaccc", where there are some overlaps but not excessively so, the approach balances between simple and complex scenarios.

**Complexity:**

* **Time Complexity:** Typically O(nlog⁡n)O(n \log n)O(nlogn) due to sorting intervals, with an additional O(n)O(n)O(n) for finding occurrences and generating intervals.
* **Space Complexity:** O(n)O(n)O(n) for storing intervals and related data.
* **Best Case:** Linear time complexity O(n)O(n)O(n) with minimal space usage.
* **Worst Case:** O(nlog⁡n)O(n \log n)O(nlogn) time complexity due to sorting, with linear space complexity.
* **Average Case:** Typically O(nlog⁡n)O(n \log n)O(nlogn) time complexity, with linear space complexity, depending on the distribution of characters in the string.

The approach effectively balances between these cases, ensuring efficient handling across a range of input scenarios.

**COCLUSION:**

In conclusion, the method for finding the maximum number of non-overlapping sub strings in a given string \( s \) is both systematic and efficient. By first identifying and extending the smallest intervals that cover all occurrences of each character, and then sorting these intervals by their end positions, we can ensure that the resulting sub strings do not overlap. This approach guarantees that we obtain the maximum number of valid sub strings while adhering to the constraints of containing all instances of any character. The result is an optimal solution that balances the number of sub strings with their total length, providing an effective way to solve the problem.