



**October University for Modern Sciences and Arts**

**Faculty of Computer Science**

**Advanced Algorithms Project**

**“longest common substring”**

**Project Documentation**

**Description** :

* the chosen problem :

the longest common substring:

the problem is to find the longest common substring of two or more strings

* the selected algorithms:
  + hashing tables
* Define a function longest\_common\_substring(s1, s2) that takes in two strings s1 and s2 as input.  
    
  Initialize an empty list substrings to store the common substrings between s1 and s2.  
    
  Iterate over s1 and generate all possible substrings of s1. For each substring, calculate its hash value using a suitable hash function.  
    
  Iterate over s2 and generate all possible substrings of s2. For each substring, calculate its hash value using the same hash function.  
    
  Compare the hash values of the substrings of s1 and s2. If the hash values are equal, add the common substring to the list substrings.  
    
  Sort the list substrings in descending order of length. The longest common substring will be the first element in the sorted list.  
    
  Return the longest common substring.

Dynamic programming algorithm

#This code defines a function longest\_common\_substring that takes two strings as input and returns the longest common substring of the two strings.  
  
#The function first initializes a 2D table dp of size (m+1) x (n+1), where m and n are the lengths of the two strings. It then iterates over the characters of the two strings and fills the table as follows: if the characters at the current positions in the two strings are equal, then it sets dp[i][j] to dp[i-1][j-1] + 1. This means that the current character is part of a common substring that extends the previous common substring by one character. If the characters are not equal, it sets dp[i][j] to 0, which means that the current character is not part of a common substring.  
  
#After the table is filled, the function finds the maximum value in the table, which is the length of the longest common substring. It then backtracks through the table to find the actual substring. It does this by starting at the cell with the maximum value and following the path of maximum values until it reaches a cell with value 0. This gives the function the longest common substring of the two input strings.  
  
#Finally, the function returns the longest common substring. The code also includes some test cases to demonstrate how the function works.

* + the designed algorithm(s): simple linear search
* # Define a function longest\_common\_substring that takes in two strings as arguments.  
  # Initialize an empty list substrings to store all the common substrings.  
  # Iterate through both strings and for each index, check if the characters at that index are the same. If they are, add the character to a temporary string temp.  
  # If the characters are not the same, add temp to the substrings list and reset temp to an empty string.  
  # Sort the substrings list in descending order based on the length of each string.  
  # Return the first element of the sorted list as the longest common substring.  
  # Here is the code implementation:  
    
    
  # The algorithm used in this solution is a simple linear search through the two strings,  
  # where it compares the characters at each index and adds them to a temporary string

B) Implementation Details:

-Hash tables

def longest\_common\_substring\_Hash(s1, s2):

  substrings = []

  for i in range(len(s1)):

    for j in range(i+1, len(s1)+1):

      # Calculate the hash value of the substring of s1

      hash\_value = hash(s1[i:j])

      # Check if the substring with the same hash value is present in s2

      if s2.find(s1[i:j]) != -1:

        # If the substring is present, add it to the list

        substrings.append(s1[i:j])

  # Sort the list in descending order of length

  substrings.sort(key=len, reverse=True)

  # Return the longest common substring

  return substrings[0]

def analyze\_longest\_common\_substring\_Hash():

    # Test cases with small input sizes

    start, end = 0, 0

    start = time.time()

    print(longest\_common\_substring\_Hash("abcdef", "abcdefg"))  # Output: "abcdef"

    print(longest\_common\_substring\_Hash("abcdefg", "hijklmg"))  # Output: "g"    end = time.time()

    end = time.time()

    print(f"Hash table Emprical Anlaysis: {end - start} seconds")

analyze\_longest\_common\_substring\_Hash()

-Dynamic programming algorithm

#Dynamic Algorithm

def longest\_common\_substring\_Dynamic(s1, s2):

    m, n = len(s1), len(s2)

    dp = [[0 for \_ in range(n+1)] for \_ in range(m+1)]

    longest, end = 0, 0

    for i in range(1, m+1):

        for j in range(1, n+1):

            if s1[i-1] == s2[j-1]:

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

                if dp[i][j] > longest:

                    longest = dp[i][j]

                    end = i

            else:

                dp[i][j] = 0

    return s1[end-longest:end]

def analyze\_longest\_common\_substring\_Dynamic():

    # Test cases with small input sizes

    start2 = time.time()

    print(longest\_common\_substring\_Dynamic("abcde", "abxde"))  # Output: "ab"

    print(longest\_common\_substring\_Dynamic("abcdefg", "ghijklmno"))  # Output: "g"

    print(longest\_common\_substring\_Dynamic("abcdefg", "cde"))  # Output: "cde"d"

    end2 = time.time()

    print(f"Dynamic Emprical Anlaysis: {end2 - start2} seconds")

analyze\_longest\_common\_substring\_Dynamic()

-the designed algorithm(s): simple linear search

#Designed Algoritm

def longest\_common\_substring\_Designed(s1, s2):

  substrings = []

  temp = ""

  for i in range(len(s1)):

    if s1[i] == s2[i]:

      temp += s1[i]

    else:

      substrings.append(temp)

      temp = ""

  substrings.append(temp)

  substrings.sort(key=len, reverse=True)

  return substrings[0]

def analyze\_longest\_common\_substring\_Designed():

    # Test cases with small input sizes

    start2 = time.time()

    print(longest\_common\_substring\_Designed("abcdefg", "abcdegf"))  # should output "abcd"

    end2 = time.time()

    print(f"Designed Emprical Anlaysis: {end2 - start2} seconds")

analyze\_longest\_common\_substring\_Designed()

C) Analysis of the algorithms:

-Hash tables

.Analytical ->O(n^2\*m+n\*logn(n))

.Empriical-> 0.0012688636779785156 seconds

-Dynamic Programming

.Analytical ->O(n)

(n)∑(i=0)

=n-0-1= 0(n)

.Empriical->0.00115203857421875 seconds

-Designed Algorithm

.Analytical ->O(n)

(n)∑(i=0)

=n-0-1= 0(n)

.Empriical-> 0.00039577484130859375 seconds

D) Discussion of the results

-Hash tables

Input->”abcdef”,”abcdefg”

Output->”abcdef”

-Dynamic Programming

.Input ->”abcdefg”,”abcdegf”

.Output->”abcde”

-Designed Algorithm

.Input ->”abcdefg”,”ghijklm”

.Output->”g”