Arai Leyva
Andrew Conti
Eliot Park
Quan Truong

Project 2

Algorithm 1: Target terms or Substrings

a. **Pseudocode**

```
Algorithm1(inputA, inputB)
{
   // Step 1: Initialization
   full_string = inputA[0]
   output_order = []       // Will hold the starting index of each found word
   output_array = []       // Will hold the actual matched words

   // Step 2: Search for each word in inputB within full_string
   for each word in inputB {
      index = full_string.find(word)    // Find the starting index of the word

      if index ≠ -1 {                   // If the word is found
         output_order.append(index)    // Add the index to output_order
         output_array.append(word)     // Add the word to output_array
      }
   }

   // Step 3: Combine and sort results by index
   combined = zip(output_order, output_array)   // Create list of (index, word) pairs
   sort(combined)                               // Sort pairs by index (ascending)

   // Step 4: Unzip sorted results into separate arrays
   if combined is not empty {
      output_order, output_array = unzip(combined)
      return output_order, output_array
   }
   else {
      return empty list, empty list
   }
}
```

b. **Analysis and Efficiency class**

The algorithm runs efficiently with a time complexity of O(m × n + m log m), where m is the number of target words and n is the length of the input string. Each word is searched using Python's find() method in linear time O(1) , and the results are sorted using Timsort O(n log n) , which is highly efficient. Space complexity is O(m),

c. **Output**

/usr/bin/python3 "/Users/quantruong/CSU Fullerton Dropbox/Hung Truong/CPSC 335 Algo/CPSC335_Project2/target_terms.py"
--- Result for array 1 ---
Output_order: [34, 38, 44, 51]
Output_array: ['brea', 'corona', 'modesto', 'clovis']
--- Result for array 2 ---
Output_order: [12, 25, 43, 49]
Output_array: ['fremont', 'fullerton', 'fresno', 'chino']
--- Result for array 3 ---
Output_order: [21, 26, 43, 59]
Output_array: ['marco', 'oxnard', 'irvine', 'orange']

Algorithm 2: Run Encoding Problem

a. **Pseudocode**:

```
Algorithm string_run_encoding(string)
  If S is empty Then
    Return an empty string

  result ← empty list
  count ← 1
  For i from 1 to Length(S) - 1 Do
    If S[i] == S[i - 1] Then
      Increment count
    Else
      If count > 1 Then
        Append count and S[i - 1] to result
      Else
        Append S[i - 1] to result
      End If
      count ← 1
    End If
  End for
```

```
    # Process last run
    If count > 1 Then
        Append count and S[length(S) - 1] to result
    Else
        Append S[length(S) - 1] to result
    End If

    Return concatenation of all elements in result
End
```

b. **Analysis and Efficiency Class**
   1. Checking if the string is empty → O(1)
   2. Initialize variables → O(1)
   3. For loop iterating over string (n-1 times where n is the length of S) → O(n)
      Inside the loop:
         a. Comparison and increment → O(1)
         b. Else, comparison and append → O(1)
         c. Reset count → O(1)
   4. Last run, if comparison and append → O(1)
   5. Return statement → O(n)

The dominant term in the analysis is O(n) from the main loop and the final string concatenation, and since we ignore the constants, the final efficiency class is O(n).

c. **Output**

quantruong@Quans-MacBook-Air CPSC335_Project2 % python3 string_run_encoding.py
"ddd" becomes "3d"
"heloooooooo there" becomes "hel8o there"
"choosemeeky and tuition-free" becomes "ch2osem2eky and tuition-fr2e"

Algorithm 3: Merging Techniques

a. **Pseudocode**

```
Algorithm merging_array(list_of_arrays)
        min_heap = [] // Create a min-heap to store the current smallest elements
        merged_list = [] // Create an empty list to hold the final sorted output
        // Step 1: Push the first element of each list into the min_heap
        For i from 0 to length of list_of_arrays - 1:
            current_list = list_of_arrays[i]
```

```
        If current_list is not empty:
            value = current_list[0]
            Push (value, i, 0) into min_heap
            // (value, list_index, element_index)
// Step 2: Process the min_heap
 While min_heap is not empty:
        Pop (value, list_index, element_index) from min_heap
        Append value to merged_list

        // If there's a next element in the same list
        If element_index + 1 < length of list_of_arrays[list_index]:
            next_value = list_of_arrays[list_index][element_index + 1]
            Push (next_value, list_index, element_index + 1) into min_heap
 Return merged_list
```

b. Analysis and Efficiency class

The merge_sorted_list algorithm uses a min-heap to merge k sorted lists with N elements. It first inserts the first element of each list into the heap in O(k log k) time. Then, for each of the N elements, it performs a heap pop and possibly a push, each taking O(log k) time. This results in a merging phase that runs in O(N log k) time. Overall, the algorithm runs in O(N log k) time, making it efficient for merging multiple sorted lists, especially when k is much smaller than N.

c. Output

quantruong@Quans-MacBook-Air CPSC335_Project2 % python3 merging_techniques.py
Array_1: [-10, -1, 0, 2, 2, 4, 5, 6, 9, 12, 20, 21, 81, 121, 150]
Array_2: [-3, 0, 3, 7, 8, 9, 10, 11, 11, 12, 17, 18, 19, 21, 29, 29, 81, 88, 121, 131]
Array_3: [-4, -2, 0, 2, 4, 5, 6, 6, 7, 10, 10, 12, 14, 15, 20, 24, 25]