Assignment 3

Course: COMPSCI 2C03

Date: 10/24/2025

Author: Sulagna Nandi

MacID: nandis

Student Number: 400575118

Professor: Dr. Vincent Maccio

Problem 1.1: Sort Algorithm Invariants and Bound Function

This sorting algorithm is correct and is essentially a reverse bubble sort (the smallest values "bubble" down to the start of the list instead of the largest values bubbling up to the end of the list). The invariant for the outer loop can be $\forall k, 0 \leqslant k < i,\ L[k] \leqslant L[k+1]$. The invariant for the inner loop can be $\forall k, j \leqslant k < N,\ L[k-1] \leqslant L[k]$. The bound function for the outer loop, which represents the number of iterations left for the outer loop and is guaranteed to reach 0, would be $f(i) = N - i$. Similarly, the bound function for the inner loop would be $f(j) = j - i$.

Problem 1.2: Stability

This sorting algorithm is essentially reverse bubble sort, which is stable. This is because only adjacent elements are swapped, and they are swapped if and only if the left element is greater than the right element. When the smallest elements start bubbling down, duplicates will stop bubbling when they are next to each other because there is no "equal to" in our condition. This is why it's not possible for a duplicate that was originally at a higher index in the list to end up being at a lower index than one of its copies.

Problem 1.3: WC Runtime, Optimal or Not

The worst-case time complexity of this reverse bubble sort algorithm (which is the same as its average case) is $\Theta(N^2)$, where N is the number of elements in the list. This is because, in the worst case, the number of comparisons will be equal to the number of iterations. The number of iterations should be equal to, based on the i and j ranges, $\sum_{i=0}^{N}(N-1-i) = \frac{N^2-N-2}{2}$. This is on the order of N^2 and, using the definition of Big Theta, can be proved to be an element of $\Theta(N^2)$.

Problem 1.4: WC Memory

The worst case (also best and average case) memory usage of this reverse bubble sort algorithm is $\Theta(1)$. This is because swapping does not require a number of new variables that is dependent on N (no new lists created), and everything is done in place, within the original list.

Problem 2.1: Sort Violations Example

In the list [5,2,3,1], the sort violations can be found by looking at each element one at a time, starting at index 0, and comparing it with each element at a higher index than it. The sort violations (index pairs that satisfy the sort violation definition) are (0,1), (0,2), (0,3), (1,3), and (2,3). These indices respectively correspond to the element pairs [5,2], [5,3], [5,1], [2,1], and [3,1] (the first element is greater than the second element in each pair).

Problem 2.2: WC Sort Violations

The worst case is when the list is sorted in reverse order (i.e. highest to lowest). This is because every single possible pair from the list will be a sort violation. If the list were not in reverse order, there would be at least one pair that is not a sort violation (therefore not the worst case).

The most straightforward way to find all the sort violations is to look at all possible pairs of elements linearly. To do this, each of the L elements must be compared with the (L-i) elements in front of it. Since we are looking at the worst case, the number of sort violations will be equal to the sum of all comparisons:

$$\sum_{i=1}^{L}(L-i) = L\sum_{i=1}^{L}1 - \sum_{i=1}^{L}i = L^2 - \frac{L(L+1)}{2} = \frac{L(L-1)}{2}$$

Problem 2.3: Efficient Sort Violations Counter

The strategy is to go through a Merge Sort algorithm, which has a time complexity of $\Theta(n\log(n))$, and count the sort violations within it.

Algorithm 1 Count Sort Violations Using Merge Sort

```
function CountViolations($arr$)
    if length(arr) $\leqslant$ 1 then
        return $arr, 0$ {sorted array, 0 violations}
    end if
    mid = $\lfloor$ length(arr)/2 $\rfloor$
    leftArray, leftViolations = CountViolations($arr[0 : mid]$)
    rightArray, rightViolations = CountViolations($arr[mid :]$)
    mergedArray, crossViolations = MergeAndCount(leftArray, rightArray)
    totalViolations = leftViolations + rightViolations + crossViolations
    return mergedArray, totalViolations
end function
function MergeAndCount(leftArray, rightArray)
    i = 0, j = 0, count = 0
    mergedArray = []
    while i ¡ length(leftArray) and j ¡ length(rightArray) do
        if leftArray[i] $\leqslant$ rightArray[j] then
            append leftArray[i] to mergedArray
            i = i + 1
        else
            append rightArray[j] to mergedArray
            count = count + (length(leftArray) - i) {all remaining elements in leftArray form violations}
            j = j + 1
        end if
    end while
    append remaining elements of leftArray and rightArray to mergedArray
    return mergedArray, count
end function=0
```