Jordan Allard
CSCI 264-01
Homework 1

HW1 Problem 4 Writeup

a) Verbal Description

This algorithm determines the number that can be found the most by forming pairs by calculating the sum of every possible pair of numbers in the array and keeping track of how often they occur.

b) Pseudocode

```
Let A be an array of size n that contains the numbers
Let S be an array of size (n(n+1))/2      // This will contain the sums
Let sIndex = 0
For every int a = 0 to a = n:
      For every int i = a + 1 to i = n:
            !! Handle duplicates !!
            S[sIndex] = A[a] + A[i]
            sIndex += 1
Sort S from least to greatest using MergeSort
Let mostCommonAmt = 1
Let mostCommonSum = S[1]
Let currentDupe = S[1]
Let currentAmt = 1
For every element with index i = 2 to i = (n(n-1))/2 in S:
      If S[i] == currentDupe:
            currentAmt += 1
            If currentAmt > mostCommonAmt:
                  mostCommonAmt = currentAmt
                  mostCommonSum = s[i]
      Else:
            currentDupe = s[i]
            currentAmt = 1
Print mostCommonAmt
Print mostCommonSum
```

c) Proof of Correctness

NOTE: THIS IMPLEMENTATION OF THE ALGORITHM IS NOT CORRECT. In order for this algorithm to truly produce a correct result with any input, it needs to account for the possibility of duplicate values. I started to implement this, but I ran out of time to get it working before the deadline and decided to submit the incomplete algorithm.

If the input does not include duplicates, this algorithm is always correct because it checks the sum of every unique pair in the array without checking duplicates (since the nested for loop starts from a + 1). Once the proper amount of possible pairs is determined, because the array is then sorted, all duplicate sums are adjacent to each other. Thus, if the next sum in the array is not equal to the current sum, it is guaranteed that there are no more copies of that sum elsewhere in the array. So, finding the largest string of adjacent duplicates will produce the correct result.

d) Running Time Estimate

O(n^2 log n)

e) Running Time Estimate Reasoning

The longest parts of this algorithm are the for loops and the sorting algorithm. In the worst case scenario, both for loops will run for the length of S, which is (n * (n - 1)) / 2, or (n^2)/2. However, the true longest part of the algorithm is sorting S from least to greatest. Since MergeSort has a time complexity of $O(n \log n)$ and the length of S is $(n^2 - n)/2$, sorting S has a time complexity of $O(n^2 \log n)$.