Find Max Pairs Double

Reasoning

Iterating over the array and all future elements

We could go through each element of the array and find the sum of it and each element ahead of it in the array, adding the sum of each to an array. Then sort the array, so all elements of the same size are next to each other. Then we could count the maximum number of occurences of one sum from the array, and return that and the number of times it appeared.

Psudocode

```
function findCommonSum (integer array input) {
    sums is an array of the sums
    for i from 0 to input length {
        for j from i to input length {
            append input[j] + input[i] to sums
        }
    }

    mergeSort(sums)

    new integer max_streak_len = 0
    new integer max_streak_holder
    new integer current_streak_len = 0
    for i from 0 to sums' length {
        if previous sum is different from this sum:
            current_streak_len = 0
        current_streak_len ++
        if current_streak_len > maximum_streak_len {
            maximum_streak_len = current_streak_len
            max_streak_holder = sums[i]
        }
    }
    return max_streak_len, max_streak_holder
}
```

Proof

This algorithm first lists every possible sum, for every possible pair of numbers in the array. Then it sorts that list, meaning each section of the same numbers is adjacent, after it goes through and finds the largest contigous block, and returns that, and how long the block was.

Running Estimate

```
function findCommonSum (integer array input) {
    sums is an array of the sums
    for i from 0 to input length { # O(n^2)
        for j from i to input length {
            append input[j] + input[i] to sums
        }
    }

    mergeSort(sums) # merge sort on n^2 elements will take O(n^2 log n^2)
or O(n^2 log n)

    new integer max_streak_len = 0
    new integer max_streak_holder
    new integer current_streak_len = 0
    for i from 0 to sums' length { # O(n^2)
        if previous sum is different from this sum:
            current_streak_len = 0
        current_streak_len ++
        if current_streak_len > maximum_streak_len {
            maximum_streak_len = current_streak_len
            max_streak_holder = sums[i]
        }
    }
    return max_streak_len, max_streak_holder
}
```

Since the longest part of this will be the merge sort, which takes $O(n^2 \log n)$, the algorithm will work in $O(n^2 \log n)$ time.