# HOMEWORK 1

COMP3121 - ALGORITHM DESIGN

# QUESTION 1

**PUNEETH KAMBHAMPATI**

*z5164647*

# PART A:

## **Pseudo-Code**

```
DEF squareSums(nums):

    squares= [ ]

    # The time complexity for this Loop will be O(n)
    FOR num in nums:
        Append num^2 to squares

    allSums = [ ]

    # This FOR loop will iterate through n, n-1, n-2, ... 1 elements
    # So in total, it will run n(n-1)/2 times
    # The time complexity for this Loop will be O(n^2)
    FOR i in  (  0 <=  i  < length of squares array  ):
        FOR j in  (  i+1 <=  j  < length of squares array  ) :
            Add ( squares[i] + squares[j] ) to the allSums array

    # we apply merge sort for the shortest worst case time complexity of
    # O(n^2*logn) for allSums array which is technically n^2 for length.
    sortedAllSums = MergeSort(allSums)


    # The time complexity for this Loop will be O(n^2)
    FOR element in sortedAllSums:
        IF element == next element:
            A number exists such that it can be written as a sum of
            two distinct numbers in our array in two different ways
            RETURN True

    RETURN False
```

## <u>EXPLANATION:</u>

The simplest way to solve this question to achieve a **WORST CASE O(n^2*logn)** time complexity was,

1. Square all the distinct numbers in array
2. Find sum of all possible pairs of squares, without reuse of numbers
3. Sort the sums
4. Check for two consecutive sums being equal

This way we can check if there were two instances of the same square sum from different numbers.

While all the time complexities are rather straightforward, the derived complexity for the merge sort algorithm wasn't trivial, so here is how I derived it,

Merge sort complexity on input size of n:

O(n*logn)

Input size for allSums array = n(n-1)/2

Merge sort complexity:

⇒ (n(n-1)/2) * log(n(n-1)/2)

⇒ O(n^2 * (logn^2))

⇒ **O(n^2 * logn)**

We can compute from the pseudo that the complexity is,

**O(n) + O(n^2) + O(n^2*logn) + O(n^2) = O(n^2*logn)**

# PART B:

## **Pseudo-Code**

```
DEF squareSums(nums):

    # Store the squared values of given array in SQUARES
    squares= [ ]

    # This FOR loop will iterate through all the elements in nums
    # The time complexity for this Loop will be O(n)
    FOR num in nums:
        Append (num^2) to squares array


    # Store the count of occurrence of each sum in a hashmap
    # sumCounts MAPS Sum -> Count
    sumCounts = { }

    # This FOR loop will iterate through n, n-1, n-2, ... 1 elements
    # So in total, it will run n(n-1)/2 times
    # The time complexity for this Loop will be O(n^2)
    FOR i in  ( 0 <=  i  < length of squares array  ):
        FOR j in  (  i+1 <=  j  < length of squares array  ) :

            # Insert, Update and Get operations are O(1) in average case
            Increment the count of sum in sumCounts
            IF count of sum in sumCounts > 1:
                RETURN True

    Return False
```

## **<u>EXPLANATION:</u>**

The new algorithm, after modifications from PART A, can run in EXPECTED time complexities of **O(n^2),**

1. Square all the distinct numbers in array
2. Find sum of all possible pairs of squares, without reuse of numbers
3. Maintain a count of the occurrence of each sum
4. Check for count of any sum reaching 2

The main reason we achieved a speedup over the previous algorithm is due to the use of the hashmap. Insertion, Updates or Getting the count of a given sum from the hashmap is O( 1 ) in average cases. This gives us the **EXPECTED** time complexity of,

$$O(n) + O(n^2)*O(1) = O(n^2)$$

However, in worst cases when we get hash collisions, we can end up with O(n) time complexity for each hashmap operation which could push our algorithm time Complexity for **WORST CASE** as,

$$O(n) + O(n^2)*O(n) = O(n^3)$$