Problems related to Big O notation:

Question 1:

Problem:
Consider the following pseudocode function:

```
FUNCTION checkEven(number):
   IF number MOD 2 == 0 THEN
      RETURN True
   ELSE
      RETURN False
   ENDIF
END FUNCTION
```

What is the time complexity of the **checkEven** function?

Question 2:

Problem:
Write a function in pseudocode that sums up all the numbers in an array and analyze its complexity.

Question 3:

Problem:
Write a pseudocode function that finds the number of duplicate pairs in an array and determine its time complexity.

Question 4:

Problem:
Consider the following pseudocode:

```
FUNCTION mixedOperations(arr):
   maxNumber <- arr[0]                    // O(1)

   FOR i <- 1 TO LENGTH(arr) - 1 DO        // O(n)
      IF arr[i] > maxNumber THEN
         maxNumber <- arr[i]
      END IF
```

```
    END FOR

    FOR i <- 0 TO LENGTH(arr) - 1 DO        // O(n)
        FOR j <- 0 TO LENGTH(arr) - 1 DO    // O(n^2)
            PRINT arr[i] * arr[j]
        END FOR
    END FOR
END FUNCTION
```

What is the overall time complexity of the **mixedOperations** function?

Bonus Question:

Two solutions are provided elsewhere on canvas for remove_dupes. What is the big O complexity of each?

Solution A: Assuming in total there are N nodes, the remove_dupes will have to iterate through all N nodes. At each iteration of the loop, current_data value is checked if it is in the non_duplicates list, which in the worst case takes O(N) time. Since this is repeated for all N nodes, the big O complexity is O(N^2)

Solution B: Assuming in total there are N nodes, the worst case would be all nodes are unique. At each iteration of the loop, there are only constant time operations. Since this loop repeats N times, the big O complexity is O(N).

Solutions:

P1.
The **checkEven** function performs a constant number of operations regardless of the size of the input **number**. It only performs one operation, which is checking if the number is even. Therefore, the time complexity of this function is **O(1)**.

P2.
Here's the pseudocode for the function:

```
FUNCTION sumArray(arr):
   total <- 0
   FOR EACH number IN arr DO
      total <- total + number
   END FOR
   RETURN total
END FUNCTION
```

Analysis:
The function **sumArray** iterates over each element in the array exactly once. If **n** is the number of elements in the array, then the function performs **n** additions. Hence, the time complexity is **O(n)**.

P3.
Here's the pseudocode for the function:

```
FUNCTION countDuplicatePairs(arr):
   duplicateCount <- 0
   FOR i <- 0 TO LENGTH(arr) - 2 DO
      FOR j <- i + 1 TO LENGTH(arr) - 1 DO
         IF arr[i] == arr[j] THEN
            duplicateCount <- duplicateCount + 1
         END IF
      END FOR
   END FOR
   RETURN duplicateCount
END FUNCTION
```

Analysis:
The function **countDuplicatePairs** uses two nested loops to compare each pair of elements in the array. For an array of size **n**, the outer loop runs **n** times and the inner loop runs up to **n-1** times in the worst case, leading to a worst-case scenario of approximately **n * (n-1) / 2** comparisons. This simplifies to **O(n^2)**.

P4.
The first part of the function finds the maximum number in an array, which takes **O(n)** time. The second part of the function prints the product of each pair of numbers in the array, which is an **O(n^2)** operation since it involves two nested loops over the entire array.

To find the overall time complexity, we sum the complexities of each part: **O(1) + O(n) + O(n^2)**. Since Big O notation describes the **upper bound**, we take the term with the highest growth rate, which is **O(n^2)**. Therefore, the overall time complexity of the **mixedOperations** function is **O(n^2)**.