Exercise 4

Explanation:

This function merge_and_fuse sorts an array while also counting the number of inversions—a measure of how far the array is from being sorted. An inversion is a pair (i, j) where i < j but A[i] > A[j].

It uses a modified merge sort approach:

1. Base Case: If the array has fewer than 2 elements, return it as-is with zero inversions.
2. Divide: The array is split into two halves.
3. Recursive Sort and Count: The function is called on both halves separately.
4. Merge and Count Split Inversions:
   - This is handled by fuse_count, which merges two sorted halves while counting how many elements from the right half precede elements from the left half (split inversions).
   - Every time an element from C is placed before elements from B, it contributes to the inversion count.
5. Return the sorted array and total inversions.

Code:

```
def merge_and_fuse(A):
    if len(A) < 2:
        return A, 0

    mid = len(A) // 2
    B, left_inv = merge_and_fuse(A[:mid])
    C, right_inv = merge_and_fuse(A[mid:])
    merged, split_inv = fuse_count(B, C)

    return merged, left_inv + right_inv + split_inv

def fuse_count(B, C):
    i, j = 0, 0
    merged = []
    inv_count = 0
    while i < len(B) and j < len(C):
        if B[i] <= C[j]:
            merged.append(B[i])
            i += 1
        else:
            merged.append(C[j])
            j += 1
            inv_count += 1

    merged.extend(B[i:])
    merged.extend(C[j:])

    return merged, inv_count

A = [1,2,7,3,4,5,6]
sorted_A, inversions = merge_and_fuse(A)
print(f"Number of inversions: {inversions}")
```

Test Case:

Input: [1,2,7,3,4,5,6]

Output: Number of inversions: 1

Exercise 6

Explanation:

This algorithm efficiently finds a single fake coin in a collection, assuming it is either lighter or heavier than real coins. It works using a divide-and-conquer approach:

1. Base Cases:

    - If there is only one coin, it must be fake, but its weight difference is unknown.
    - If there are two coins, we can't determine which is fake without a reference.

   - If there are three or four coins, we can compare groups and determine the fake one.

2. Divide and Weigh:

   - If there are more than four coins, we divide them into two groups and those into two subgroups and weigh them against each other.
   - If the two subgroups are equal, the fake coin is in the remaining part.
   - If they are unequal, we recursively search the lighter or heavier group.

Code:

```
def find_fake_coin(coins, start_index=0):
    n = len(coins)
    ##### BASE CASES #####
    if n == 1:
        return start_index, "unknown"
    elif n == 2:
        return "We can't know, add real coin to find the fake coin"
    elif n == 3:
        if coins[0] == coins[1]:
            return start_index + 2, "more" if coins[2] > coins[0] else "less"
        elif coins[0] == coins[2]:
            return start_index + 1, "more" if coins[1] > coins[0] else "less"
        else:
            return start_index, "more" if coins[0] > coins[1] else "less"
    elif n == 4:
        if coins[0] == coins[1] == coins[2]:
            return start_index + 3, "more" if coins[3] > coins[0] else "less"
        elif coins[0] == coins[1] == coins[3]:
            return start_index + 2, "more" if coins[2] > coins[0] else "less"
        elif coins[0] == coins[2] == coins[3]:
            return start_index + 1, "more" if coins[1] > coins[0] else "less"
        elif coins[1] == coins[2] == coins[3]:
            return start_index, "more" if coins[0] > coins[1] else "less"
```

```
# More than four coins, we can divide and conquer
    else:
        mid = n // 2
        left_coins = coins[:mid + 1]
        right_coins = coins[mid:]
        left_mid = len(left_coins) // 2

        if sum(left_coins[:left_mid]) == sum(left_coins[left_mid:]):
            return find_fake_coin(right_coins, start_index + mid)
        else:
            return find_fake_coin(left_coins, start_index)

# Example usage
coins = [1, 1, 1, 1, 1, 0.33, 1, 1, 1, 1]  # One fake coin (0.33)
position, weight_comparison = find_fake_coin(coins)
print(f"The fake coin is at position {position} and it weighs {weight_comparison}")
```

Test Case:

Input: [1, 1, 1, 1, 1, 0.33, 1, 1, 1, 1]

Output: The fake coin is at position 5 and it weighs less