1. Given an array of size 16, what is the maximum possible *mixed-up-ness* score? Explain why you think this (e.g., give a logical argument or provide an example)

The maximum possible mixed-up-ness score for an array of size 16 is 120. This is because the mixed-up-ness score is calculated as the sum of inversions in the array, which is the number of pairs of elements that are out of order with respect to each other. In an array of size n, the maximum number of inversions is n*(n-1)/2.

For an array of size 16, the maximum number of inversions is 16*(16-1)/2 = 120. This occurs when the array is in reverse sorted order, i.e., the largest element is at index 0 and the smallest element is at index 15. In this case, every pair of elements is out of order, resulting in the maximum possible mixed-up-ness score of 120.

For example, the following array is in reverse sorted order and has a mixed-up-ness score of 120:

{15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}

2. What is the worst-case runtime of the brute-force algorithm that you implemented? Give a proof (a convincing argument) of this.

The brute-force algorithm for calculating the mixed-up-ness score is to check every pair of elements in the array and count the number of inversions. This requires a nested loop, iterating over each element in the array and comparing it to every element that comes after it. The worst-case runtime of this algorithm is O(n^2), where n is the size of the array. This is because for an array of size n, the number of comparisons needed is proportional to the sum of the first n - 1 integers, which is equal to (n - 1) * n / 2. This means that the number of comparisons grows quadratically with the size of the input array, and hence the worst-case runtime is O(n^2).

3. State the recurrence that results from the divide-and-conquer algorithm you implemented in Part 3.

The recurrence that results from the divide-and-conquer algorithm for calculating the mixed-up-ness score is:

$T(n) = 2T(n/2) + nlog(n)$

where T(n) is the time taken to calculate the mixed-up-ness score for an array of size n. The recurrence arises from the fact that the algorithm splits the array into two halves and recursively computes the mixed-up-ness score for each half, and then combines the two halves by computing the cross inversions, which takes O(nlog(n)) time.

4. Solve the recurrence for the divide-and-conquer algorithm using the *substitution method*. For full credit, show your work.

To solve the recurrence relation using the substitution method, we first need to guess a solution for the recurrence.

Guess: T(n) <= cnlog(n)

Where c is a constant to be determined.

Now, we substitute our guess into the recurrence relation:

T(n) = 2T(n/2) + O(nlogn)

T(n) = 2[c(n/2)log(n/2)] + O(nlogn)

T(n) = cnlog(n) - cnlog(2) + O(nlogn)

T(n) = cnlog(n) - cn + O(nlogn)

Next, we need to prove that our guess is correct. That is, we need to show that T(n) <= cnlog(n) for some constant c.

Assume that T(k) <= cklog(k) for all k < n. We will use this assumption to prove that T(n) <= cnlog(n).

T(n) = 2T(n/2) + O(nlogn)

T(n) <= 2c(n/2)log(n/2) + O(nlogn)

T(n) <= cnlog(n) - cn + O(nlogn) (substituting our guess)

T(n) <= cnlog(n) (if we can show that -cn + O(nlogn) <= 0)

-cn + O(nlogn) <= 0

O(nlogn) <= cn

Thus, we have shown that T(n) <= cnlog(n) for some constant c. Therefore, our guess is correct and the solution to the recurrence relation T(n) = 2T(n/2) + O(nlogn) is T(n) = O(nlogn).

5. Confirm that your solution to #4 is correct by solving the recurrence for the divide-and-conquer algorithm using the *master theorem*. For full credit, clearly define the values of *a*, *b*, and *d*.

To use the master theorem, we need to first identify the values of a, b, and d for the recurrence relation that we obtained in problem #4. From our recurrence relation, we have:

T(n) = 2T(n/2) + nlog(n)

Here, a = 2 since we are dividing the problem into two subproblems of equal size. The value of b = 2 because we are dividing the input size by 2 in each step. Finally, d = 1 since the time complexity of the merging step is nlog(n).

Now, we can apply the master theorem:

If f(n) = nlog(n) and logb(a) = log2(2) = 1, then we have:

Case 2: If f(n) = Θ(nlogb(a) log(n)), then T(n) = Θ(nlogb(a) log2(n))

Since f(n) = nlog(n) = Θ(nlog2(2) log(n)), we can conclude that the solution to the recurrence is:

T(n) = Θ(nlog2(n))

Therefore, the time complexity of the divide-and-conquer algorithm is Θ(nlog2(n)).