Suppose we are given two sets of \( n \) points, one set \( \{p_1, p_2, \ldots, p_n\} \) on the line \( y = 0 \) and the other set \( \{q_1, q_2, \ldots, q_n\} \) on the line \( y = 1 \). Consider the \( n \) line segments connecting each point \( p_i \) to the corresponding point \( q_i \). Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in \( O(n \log n) \) time. See the example below.

Your input consists of two arrays \( P[1..n] \) and \( Q[1..n] \) of \( x \)-coordinates; you may assume that all \( 2n \) of these numbers are distinct. No proof of correctness is necessary, but you should justify the running time.

**Solution:**

We begin by sorting the array \( P[1..n] \) and permuting the array \( Q[1..n] \) to maintain correspondence between endpoints, in \( O(n \log n) \) time. Then for any indices \( i < j \), segments \( i \) and \( j \) intersect if and only if \( Q[i] > Q[j] \). Thus, our goal is to compute the number of pairs of indices \( i < j \) such that \( Q[i] > Q[j] \). Such a pair is called an **inversion**.

We count the number of inversions in \( Q \) using the following extension of mergesort; as a side effect, this algorithm also sorts \( Q \). If \( n < 100 \), we use brute force in \( O(1) \) time. Otherwise:

- Recursively count inversions in (and sort) \( Q[1..[n/2]] \).
- Recursively count inversions in (and sort) \( Q[[n/2]+1..n] \).
- Count inversions \( Q[i] > Q[j] \) where \( i \leq [n/2] \) and \( j > [n/2] \) as follows:
  - Color the elements in the Left half \( Q[1..n/2] \) blue.
  - Color the elements in the Right half \( Q[n/2+1..n] \) red.
  - Merge \( Q[1..n/2] \) and \( Q[n/2+1..n] \), maintaining their colors.
  - For each blue element \( Q[i] \), count the number of smaller red elements \( Q[j] \).

The last substep can be performed in \( O(n) \) time using a simple for-loop:

```makefile
COUNTREDBLUE(A[1..n]):
count ← 0
total ← 0
for i ← 1 to n
    if A[i] is red
        count ← count + 1
    else
        total ← total + count
return total
```
In fact, we can execute the third merge-and-count step directly by modifying the Merge algorithm, without any need for “colors”. Here changes to the standard Merge algorithm are indicated in red.

```
MERGEANDCOUNT(A[1..n], m):
    i ← 1; j ← m + 1; count ← 0; total ← 0
    for k ← 1 to n
        if j > n
            B[k] ← A[i]; i ← i + 1; total ← total + count
        else if i > m
            B[k] ← A[j]; j ← j + 1; count ← count + 1
        else if A[i] < A[j]
            B[k] ← A[i]; i ← i + 1; total ← total + count
        else
            B[k] ← A[j]; j ← j + 1; count ← count + 1
    for k ← 1 to n
        A[k] ← B[k]
    return total
```

We can further optimize this algorithm by observing that count is always equal to \( j - m - 1 \). (Proof: Initially, \( j = m + 1 \) and count = 0, and we always increment \( j \) and count together.)

```
MERGEANDCOUNT2(A[1..n], m):
    i ← 1; j ← m + 1; total ← 0
    for k ← 1 to n
        if j > n
            B[k] ← A[i]; i ← i + 1; total ← total + j - m - 1
        else if i > m
            B[k] ← A[j]; j ← j + 1
        else if A[i] < A[j]
            B[k] ← A[i]; i ← i + 1; total ← total + j - m - 1
        else
            B[k] ← A[j]; j ← j + 1
    for k ← 1 to n
        A[k] ← B[k]
    return total
```

The modified Merge algorithm still runs in \( O(n) \) time, so the running time of the resulting modified mergesort still obeys the recurrence \( T(n) = 2T(n/2) + O(n) \). We conclude that the overall running time is \( O(n \log n) \), as required.

**Rubric:** 10 points = 2 for base case + 3 for divide (split and recurse) + 3 for conquer (merge and count) + 2 for time analysis. Max 3 points for a correct \( O(n^2) \)-time algorithm. This is neither the only way to correctly describe this algorithm nor the only correct \( O(n \log n) \)-time algorithm. No proof of correctness is required.