Horizontal Crossing Problem

Reasoning

We can use a divide and conquer algorithm similar to the one we used in class. However instead of each inversion counting for 1, each inversion counts for the sum of the two elements involved. So we must add $(Bs[k] + Cs[j])$ for k from i to the end which is also $(Bs[k])$ for k from i to the end $+ (Bs.\text{len} - i + 1).\text{Cs}[j]$ We can then precompute the sums of Bs to save time.

Psudocode

```
function count(A[1 ... n]):
    if n <= 1: return 0
    m = n # 2
    B = A[1:m]
    C = A[m+1:n]
    leftcount = count(B)
    rightcount = count(C)
    Bs = sort(B)
    Cs = sort(C)
    midcount = countMidInv(Bs, Cs)
    return midcount + leftcount + rightcount

function countMidInv(Bs, Cs):
    i = 1
    j = 1
    count = 0

    # Precompute suffix sums of Bs
    suffixSum = array of length Bs.len+1
    suffixSum[Bs.len+1] = 0
    from k = Bs.len down to 1:
        suffixSum[k] = suffixSum[k+1] + Bs[k]

    while i <= Bs.len and j <= Cs.len:
        if Bs[i] <= Cs[j]:
            i += 1
        else:
            # All Bs[i..end] > Cs[j], so add weighted contribution
            num = Bs.len - i + 1
            sumBs = suffixSum[i]
            count += sumBs + num * Cs[j]
            j += 1
    return count
```

Proof

The recursive nature confirms we count all inversions: those entirely in the left half, those entirely in the right half, and those crossing between halves. Sorting each half allows us to detect inversions across in linear time. When $Bs[i] > Cs[j]$, all later elements in Bs are also greater than $Cs[j]$. Thus, we can add their contributions in bulk using suffix sums, rather than checking each pair individually. The suffix sum guarantees correctness of the weighted contribution, since it captures the total of all Bs[k] for k ≥ i.

Running Time

Estimate

$O(n \log^2 n)$

Reasoning

```
function countWeighted(A[1 ... n]) { # T(n)
    if n <= 1: return 0 # O(1)
    m = n % 2 # O(1)
    B = A[1:m] # O(n)
    C = A[m+1:n] # O(n)
    leftcount = countWeighted(B) # T(n/2)
    rightcount = countWeighted(C) # T(n/2)
    Bs = sort(B) # O(n log n)
    Cs = sort(C) # O(n log n)
    midcount = countMidInvWeighted(Bs, Cs) # U(n) = O(n)
    return midcount + leftcount + rightcount # O(1)
}

function countMidInvWeighted(Bs, Cs) { # U(n) => O(n)
    i = 1 # O(1)
    j = 1 # O(1)
    count = 0 # O(1)

    # Precompute suffix sums of Bs
    suffixSum[Bs.len+1] = 0 # O(1)
    for k = Bs.len downto 1: # O(n)
        suffixSum[k] = suffixSum[k+1] + Bs[k] # O(1)

    while i <= Bs.len and j <= Cs.len {  # O(n)
        if Bs[i] <= Cs[j]: # O(1)
            i += 1 # O(1)
        else: # O(1)
            num = Bs.len - i + 1 # O(1)
            sumBs = suffixSum[i] # O(1)
            count += sumBs + num * Cs[j] # O(1)
            j += 1 # O(1)
    }
    return count # O(1)
}
```

$T(n) = 2T(\frac{n}{2}) + O(n \log n)$ Using master theorem $a = 2, b = 2, f(n) = n \log n \; n^{\log_2 2} = n$

$f(n) = O(n \log n) =$ Case 2 So it runs in $T(n) = \Theta(n \log^2 n)$