Daily Coding Problem #44

Problem

This problem was asked by Google.

We can determine how "out of order" an array A is by counting the number of inversions it has. Two elements `A[i]` and `A[j]` form an inversion if `A[i] > A[j]` but `i < j`. That is, a smaller element appears after a larger element.

Given an array, count the number of inversions it has. Do this faster than O(N^2) time.

You may assume each element in the array is distinct.

For example, a sorted list has zero inversions. The array [2, 4, 1, 3, 5] has three inversions: (2, 1), (4, 1), and (4, 3). The array [5, 4, 3, 2, 1] has ten inversions: every distinct pair forms an inversion.

Solution

The brute force solution here should come naturally from the definition: we can run a doubly nested for loop over all pairs, and increment a counter whenever we encounter a larger element before a smaller element. That would look like this:

```
def count_inversions(arr):
    count = 0
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            if arr[i] > arr[j]:
                count += 1
    return count
```

However, this would run in O(N^2), and we want something faster. We can use the following recursive, divide-and-conquer algorithm to count the number of inversions in O(N log N) time.

- First, let's separate our input array into two halves A and B
- Count the number of inversions in each list recursively
- Count the inversions *between* A and B
- Return the sum of all three counts

If we are able to count all the inversions between A and B in linear time, then according to the master theorem for divide-and-conquer recurrences), our algorithm will run in O(N log N) time, since we have the same recurrence relationship as merge sort.

How can we count the inversions between A and B in linear time? If we expand our `count_inversions` function to also sort the array as well, we can use a similar technique to merge sort to merge and also count the inversions between A and B. To be specific, assuming A and B are sorted, we can construct a helper function that does the following:

- Scan A and B from left to right, with two pointers `i` and `j`
- Compare `A[i]` and `B[j]`
    - If `A[i]` is smaller than `B[j]`, then `A[i]` is not inverted with anything from `B`, since it's expected that everything in `A` would be smaller than everything in `B` if `A + B` was sorted.
    - If `A[i]` is greater than `B[j]`, then `B[j]` is inverted with everything from `A[i:]`, since `A` is sorted. Increment our counter by the number of elements in `A[i:]`.
- Append the smaller element between `A[i]` and `B[j]` to our sorted list
- Return the sorted list

```python
def count_inversions(arr):
    count, _ = count_inversions_helper(arr)
    return count

def count_inversions_helper(arr):

    if len(arr) <= 1:
        return 0, arr
    mid = len(arr) // 2
    a = arr[:mid]
    b = arr[mid:]
    left_count, left_sorted_arr = count_inversions_helper(a)
    right_count, right_sorted_arr = count_inversions_helper(b)
    between_count, sorted_arr = merge_and_count(left_sorted_arr, right_sorted_arr)
    return left_count + right_count + between_count, sorted_arr

def merge_and_count(a, b):
    count = 0
    sorted_arr = []
    i, j = 0, 0
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            sorted_arr.append(a[i])
            i += 1
        elif a[i] > b[j]:
            sorted_arr.append(b[j])
            count += len(a) - i
            j += 1
    sorted_arr.extend(a[i:])
    sorted_arr.extend(b[j:])
    return count, sorted_arr
```

Press