3) We can modify the Merge function in Mergesort algorithm to count the number of inversions in the array. Basically, during the merge operation, we have two sorted arrays A and B, subarrays of the total. If an element b of B is smaller than an element a of A, then all elements in A after a are also larger than b. So, we add that number to our total count. Otherwise, the total remains constant.

Pseudocode:

```
def mergesort(arr):
  n = arr.len()
  if n < 1:
    return (0, arr)
  mid = n/2
  inv1, arr1 = mergesort(arr[0, mid])
  inv2, arr2 = mergesort(arr[mid+1,n-1])
  return merge(inv1+inv2, arr1, arr2)

def merge(inv, arr1, arr2):
  merged = []
  i, j=0
  n_left = arr1.len()
  n_right = arr2.len()
  while (i < n_left and j < n_right):
    if arr1[i] > arr2[j]:
      inv += (n_left-i)
      merged.append(arr2[j]]
      j++
    else:
      merged.append(arr1[i])
      i++
  while (i < n_left):
    merged.append(arr1[i])
    i++
  while (j < n_right):
    merged.append(arr2[j])
    j++
  return (inv, merged)
```

Correctness:
Invariant: After each merge operation, the number of inversions within the original subarray is correct, and the subarray is sorted

Initialisation:
We shall consider two cases:
Case 1: The array is empty. In this case, there are no elements, hence there are no bad pairs of elements. Inversions are zero. And since there are no elements, it is also true that it is sorted.

Case 2: The array has one element. In this case, there are also no pairs possible, so there are no bad pairs of elements. Inversions are zero. And since there is only one element, it is also true that the array is sorted.
In both cases, the subarray is sorted, and captures the correct number of inversions.

Maintenance:
We have two sorted arrays with the right number of inversions. Next, we consider the recursive case, when we merge the two arrays. Let A and B be the left and right half of the array we have split, and both are sorted. Also, we set i and j to be the index pointers of A and B. Next, during the merge operation, if an element b of B is smaller than an element a of A, the all the elements after a in A are also bigger than b, so b makes a bad pair with (len(A)-i) other elements, which is accounted for in the number of inversions. Otherwise, if all elements of A are smaller than all elements of B, this addition won't trigger, so the inversions for this recursive step would be zero (as expected). Therefore, the inversions are accurately represented. Also, since this is a mergesort algorithm, this process also leads to a sorted array.

Termination:
Once we are done merging, it is necessarily true that the final array is sorted, as this is the mergesort algorithm. Also, we don't add to the number of inversions in this final step, so as long as no more merge operations are done, then the number of inversions will remain as it is.

Runtime Complexity:
Since this is essentially a mergesort algorithm, then we have the same runtime complexity as the mergesort algorithm, which is O(nlgn).