Table of Contents

Question 1.

First, we pick a random magnet. This algorithm will split the arrays until they are of size one. Then, check every size-one arrays to see if it attracts or repels the random magnet. We add them to the respective global linked lists, and we concatenate them in the end.

Algorithm
Sort_Mangets (`magnets`: [array of north and south magnets])

```
1: random_magnet = magnets[random()*magnets.length - 1]
2: attracts = new linked_list()
3: repels = new linked_list()
4:
5: function split_and_merge (magnets)
6:     if magnets.length == 1 then
7:         if magnets repels random_magnet then
8:             repels.append(magnet)
9:         else
10:            attracts.append(magnet)
11:        return
12:
13:    split_and_merge(magnets[:magnets.length/2])
14:    split_and_merge(magnets[magnets.length/2:])
15:
16: split_and_merge(magnets)
17: result_magnets = attracts + [random_magnet] + repels
18: return result_magnets
```

`Sort_Mangets()` is feasible since we return a list of magnets.

Proof of complexity:
This algorithm consists of three parts

1. Picking a random magnet
2. Divide and Conquer
3. Merging the final two lists and the random magnet

Note: All container elements are lists with head and tail pointers, except the initial given array of magnets. Thus, merging and appending to the end is O(1).

Picking a random magnet is clearly O(1) from the magnets array. Merging the two final linked lists with the extra element is also O(1) since everything is in a linked list.

The divide and conquer only works on the base case, and returns nothing. Therefore, the recurrence relation is T(n) = 2T(n/2). Assuming T(1) is a constant, we can recurse this relation to its base case giving us $T(n) = 2^{\log_2(n)}T(1) = nT(1) \in O(n)$.

$\therefore O(1) + O(1) + O(n) \in O(n)$

Proof of Correctness (by Induction):

Base Case:

Test the single magnet against our indicator magnet:
Case 1: The magnets repel so add it to the repels list
Case 2: The magnets attract so add it to the attracts list

Since the base case is applied over all elements then they will be in the appropriate list. So when all lists are merged all the elements will be sorted.

Question 2.

This algorithm is pretty much the counting inversions algorithm we saw in class, but we also call it with the right array doubled.

Algorithm
Sort_and_Count (list: [array of list])

```
1: function merge_and_count (left_half, right_half)
2:     i = 0, j = 0, count = 0
3:     merged = []
4:
5:     while i ≠ left_half.length and j ≠ right_half.length do
6:         if j == right_half.length then
7:             merged.append(left_half[i])
8:             i++
9:         if i == left_half.length then
10:            merged.append(right_half[j])
11:            j++
12:            count++
13:        if left_half[i] ≥ right_half[j] then
14:            merged.append(left_half[i])
15:        else
16:            merged.append(right_half[j])
17:            count++
18:    return count, merged
19:
20: if list.length == 1 then
21:    return (0, list)
22:
23: left_half = list[:list.length]
24: right_half = list[list.length:]
25:
26: left_inversions, left_half = Sort_and_Count(left_half)
27: right_inversions, right_half = Sort_and_Count(right_half)
28:
29: inversions, _ = merge_and_count(left_half, right_half.map(n => 2*n))
30: _, list = merge_and_count(left_half, right_half)
31: return left_inversions + right_inversions + inversions, list
```

`Sort_and_Count()` is feasible since we return a number of inversions.

Proof of complexity:
This algorithm is similar to `sort-and-count()` which was discussed in class. The only difference is that we use one more `merge-and-count()` call. This adds one more $O(n)$ call to our algorithm so overall the recurrence relation is

$$T(n) = 2T\left(\frac{n}{2}\right) + 2O(n) + O(1) = 2T\left(\frac{n}{2}\right) + O(n) + O(1) \in O(n \log n)$$

as shown in lecture.

Proof of Correctness (by Induction):

Base Case:

Test with one element: return that one element has 0 inversions.

Induction Hypothesis:

Assume that our algorithm works for lists of size $0 \leq i < n$.

Induction Step:

Consider n:

When we call `merge-and-count()` with the right array doubled, we will count inversions where $i < j,\ a_i > 2a_j$ similar to `merge-and-count()` in lecture. Adding this to the counts from the left and right subarrays with element $n/2$ we will get the total amount of inversions from the entire merged array as smaller subsets are correct by our induction hypothesis.