Problem 1

a)

The Goofy Algorithm will work if he is lucky enough, the best case scenario will happen if Goofy runs the Algo when it's already sorted.

The worse case might never happen which will result in a stack overflow bse the base case might never be achieved

The average case will be achieved at some point if Goofy runs the Algo and he gets the result after so many trials

b) Best case scenario will happen if Goofy gets an already sorted array as his input

c) Best case runtime is O(N) because it will loop through to check if the array is already sorted

d) The worst case scenario of this Algorithm is Infinite. This will might cause a stack overflow even before the algorithm gets you a result

e) Assuming you input an array of 9 elements

Average no of inversions = n(n-1)/4

= 9(9 - 1)/4 = 18 inversions on average

Assuming you got results on the second shuffle. This means that the algorithm went through the array atleast twice.

Total number of Comparisons = 8 comparisons atleast

Since for an algorithm to be inversion bound the number of inversions should be equal or greater than the number of comparisons. Meaning that this Goofy's Algorithm is not inversion bound

Problem 2

```
public int[] sortAlgo(int[] array){
        int countOnes = 0, countZeros = 0;
        for(int num : array){
            if(num == 0){
                countZeros++;
            }else if(num == 1){
                countOnes++;
            }
        }
        for(int i = 0; i <array.length; i++){
            if(i < countZeros) array[i] = 0;
            else if(i >= countOnes && i < (countZeros + countOnes))  array[i] = 1;
            else array[i] = 2;
        }
```

```
        return array;
    }
```

The above algorithm sorts the array in O(N) because the runtime is O(N + N) which becomes O(N)

Problem 3

a) After the array is sorted the exterior while loop will not execute. Hence no extra trips will be taken given the array is already sorted

```
        public int[] sort(int[] arr) {
            int len = arr.length;
            //for(int i = 0; i < len; i++){
            boolean isSorted = false;
            while(!isSorted){
                isSorted = true;
                for (int j = 0; j < len - 1; j++){
                    if(arr[j] > arr[j+1]){
                        swap(arr,j,j+1);
                        isSorted = false;
                    }
                }
            }
            return arr;
        }
```

b) Refactored the above algorithm not to consider the sorted part after every trip.

```
public int[] sort(int[] arr) {
    int unsortedLength = arr.length - 1;

    boolean isSorted = false;
    while(!isSorted){
        isSorted = true;
        for (int j = 0; j < unsortedLength; j++){
            if(arr[j] > arr[j+1]){
                swap(arr,j,j+1);
                isSorted = false;
            }
        }
        unsortedLength--;
    }
    return arr;
}
```

c)

The running time turned out to be smaller for the second option because after every trip the largest element was found to be at the end of the loop. This led to a better time than the previous which only had an improvement for the loop to break once the entire array was sorted.

Problem 4

```
    public int[] countOfZerosAndOnes(int[] arr){
        int index = binSearch(arr,0,arr.length-1);
        int countZeros = index + 1;
        int countOnes = arr.length - countZeros;
        return new int[]{countZeros,countOnes};
    }

    private int binSearch(int[] arr, int lower,int upper){
        int mid = (lower + upper)/2;
        if(arr[mid]==0 && arr[mid+1]==1) return mid;
        else if(arr[mid]==1 && arr[mid-1]==0) return mid-1;
        else if(arr[mid]==1 && arr[mid-1]==1) return binSearch(arr,lower, mid-1);
        else return binSearch(arr,mid+1,upper);
    }
```