PART B

(a)

Minimum: 0
Maximum: nC2 = [n(n-1)/2]

(b) Pseudocode:

```
showBadPairs(arr, length_of_arr):
    for i in 0 to (length_of_arr-1):
        for j in (i + 1) to (length_of_arr-1):
            if arr[i] > arr[j] + 2:
                print((arr[i], arr[j]))
```

For each element arr[i] we check if there exists a j>i such that a[i] > a[j]+2, if true then it is a bad pair and hence we print it.

It is a simple brute force algorithm
The composite loop runs (n-1)+(n-2)+(n-3)+…+1 = [{n(n+1)/2} - n] times
Time complexity: O(n^2)

(c) Pseudocode:

```
countBadPairsInSpecialArray(arr, lowest_index, highest_index):
    rightHalf = (highest_index - lowest_index + 1) / 2
    i = (highest_index + lowest_index) / 2
    j = highest_index
    count = 0
    while i >= lowest_index and rightHalf > 0:
        if arr[i] > arr[j] + 2:
            count = count + rightHalf
            i = i - 1
        else:
            j = j - 1
            rightHalf = rightHalf - 1

    return count
```

i points to the index of the greatest unvisited element in the left half
j points to the index of the greatest unvisited element in the right half
count hold the number of bad pairs
rightHalf keeps track of the number of elements between arr[j] and the smallest element in the right half(both inclusive).

If arr[i] > arr[j]+2,
then since arr[i] is currently the greatest unvisited element in the left half and arr[j] is currently the greatest unvisited element in the right half, therefore arr[i] forms bad pairs with arr[j] and also with all the elements lesser than arr[j] and hence the number of bad pairs, denoted by *rightHalf*, is added to the current count and we move to the next greatest element in the left half.

Otherwise,
Since the current greatest element of the left half does not form a bad pair with current greatest element of the right half, no other element of the left half can form a bad pair with the current arr[j] and so we move to the next greatest element in the right half and also reduce *rightHalf*

At each step of the loop we reduce one of i or j and hence the loop will run for at most n-1 times (n=length_of_array)
Time Complexity: O(n)

(d) Pseudocode:

```
count = 0 //free variable used for counting

countBadPairs(arr, lowest_index, highest_index):
    if lowest_index < highest_index:
        mid = (lowest_index + highest_index) / 2
        countBadPairs(arr, lowest_index, mid)
        countBadPairs(arr, mid + 1, highest_index)
        count = count + mergeAndCount(arr, lowest_index, mid, 
highest_index)


mergeAndCount(arr, low_index1, high_index1, high_index2):
    badPairCount = countBadPairsInSpecialArray(arr, low_index1, 
high_index2)

    mergedArr = [None] * (high_index2 - low_index1 + 1)
    i = low_index1
    j = high_index1 + 1
    k = 0

    while i <= high_index1 and j <= high_index2:
        if arr[i] > arr[j] + 2:
            mergedArr[k] = arr[j]
            j = j + 1
        else:
```

```
            mergedArr[k] = arr[i]
            i = i + 1

        k = k + 1

    while i <= high_index1:
        mergedArr[k] = arr[i]
        i = i + 1
        k = k + 1

    while j <= high_index2:
        mergedArr[k] = arr[j]
        j = j + 1
        k = k + 1

    j = low_index1
    for i in range(0, k):
        arr[j] = mergedArr[i]
        j = j + 1

    return badPairCount
```

We build this algorithm over simple merge sort algorithm.
We take advantage of the fact that on each merging step (done by mergeAndCount) we logically merge 2 sorted arrays, and we logically pass the 2 sorted arrays as a single array with a sorted left half and a sorted right half, and then countBadPairsInSpecialArray counts and returns the number of bad pairs in the passed array.

The merging operation takes $\Theta(n)$ time and countBadPairsInSpecialArray takes $O(n)$ and
$\Theta(n)+O(n) = \Theta(n)$
So the time complexity of this algorithm remains the same as that of merge sort.
Time complexity: $\Theta(n\log n)$

Note:

We can skip the sorting of the two halves in the final step of merging since we get the *badPairCount* before sorting begins, but even then we will only reduce our cost by $O(n)$, while the dominating term in nlogn and so the time complexity of our algorithm still remains $\Theta(n \log n)$.

(e)

Yes, Alice's argument is correct.

We need every sub-array that we pass into *countBadPairsInSpecialArray* to at least be sorted in the left half and the right half.

And that is why we are using merge sort, which always merges two sorted sub-arrays.

And any comparison based sorting algorithm like merge sort always has a lower bound of $\Omega(n \log n)$ and so the above algorithm (d) also has a lower bound of $\Omega(n \log n)$.

(f)

Recurrence relation for algorithm (d) is:

$T(n) = 2T(n/2) + \Theta(n) + O(n)$

$\Rightarrow T(n) = 2T(n/2) + \Theta(n)$

Solving the above relation using master theorem,

$a = 2$, $b = 2$, $f(n) = \Theta(n)$ —> (i)

$n^{(\log_b a)} = n^{(\log_2 2)} = n^1 = n$

$\Theta(n^{(\log_b a)}) = \Theta(n)$ —> (ii)

Therefore, $f(n) = \Theta(n)$ using (i) and (ii)

And hence as per master theorem,

$T(n) = \Theta([n^{(\log^b a)}][\log n]) = \Theta(n \log n)$

And $T(n) = \Theta(n \log n)$ implies both

$T(n) = O(n \log n)$ and $T(n) = \Omega(n \log n)$ [Hence Proved]