Tony Kovanen

Design and Analysis of Algorithms

3.
This algorithm has a running time of O(n log n), since it uses merge-sort. This is the component with the highest running time, since the while loop present in the algorithm is linear. This is because either variable i or j will move one step on each iteration if the sum x is not found. When the variables “meet” the algorithm will break the while loop and return false. This means that each element in the collection is visited at most once, making the loop O(n). Everthing inside and outside the loop except merge-sort function call are O(1).

The algorithm is based on the principle that when we look at the first and the last element, we can figure out wether the sum we are looking for is larger or smaller than the on of the current elements. Based on this information we can increase or decrease our function to the next possible value by incrementing or decrementing one of the pointer variables (i,j) present in the algorithm, since the collection is sorted prior to these procedures. Since we are always moving from one value to the next, we are bound to find sum x, should it exist.

```
findSum(S, x)
        merge-sort(S) // O(n log n)
        int i = 1
        int j = S.length
        while (i < j)
                if S[i] + S[j] == x
                        return true
                if S[i] + S[j] > x
                        j--
                else
                        i++
        return false
```

4.
a)
(2,1), (3,1), (8,6), (8,1), (6,1)

b)
An array that is in a reverse sorted order has the most inversions, where the number of inversions is n-1 + n-2 + ... + 1 = Σ^n-1_1, since each element can be swapped with all its previous elements.

c)
The less inversions a collection has, the less is the running time of insertion sort. This is because every time there is an inversion in the collection, the latter component of the inversion must be swapped all the way to it's correct place.

d)
Modifying merge-sort so that each time an element is merged to the merged array from the right subarray before the left subarray is empty, we increment a counter. The running time of the algorithm will not be affected.

6.

```
multiplyComplex(a, b, c, d)
        realComponent = a * c – b * d
        imaginaryComponent = (a+b) * (c+d) – a * c – b * d
        return realComponent, imaginaryComponent
```

Here the running time is always constant.