1. 10 points. Let's take a look at this loop in insertion sort

Algorithm 1 Code snippet.

```
while i > 0 and A[i] > key do
    A[i + 1] ← A[i]
    i ← i − 1
end while
```

This loop does a linear search to scan through the sorted sub-array in backwards direction to find the proper position for key and then shifts the elements that are greater than the key towards the end to perform the correct insertion.

Even if binary search finds the position in logarithmic time, it still needs to shift all elements after it to the right, which is linear in the worst case. It will perform the same number of swaps, although it would reduce the number of comparisons.

2. 10 points. No the algorithm does not correctly sort the elements. Firstly, it tries to access a memory location that does not exist, i.e., by using $A[k+1]$ which is why it will throw a memory error. Secondly, swapping takes place between k^{th} index and j^{th} index when the outer loop is iterating over i.

   Counter Example: If $A = 1, 3, 100, 4$ then the output will be $1, 100, 4, 3$ which is clearly not sorted.

3. 10 points.

   (a) Assume that our integers are stored in array $A[1 \cdots n]$ that stores n integers. The range of integers lies in between 0 to k.

   (b) We need two additional arrays $B[1 \cdots k]$ and $C[1 \cdots k]$.

   (c) Our algorithm will first initialize all elements of array B to 0. This step requires $O(k)$ time.

   (d) Next, for each element of array A (i.e., $A[i]$ for $i = 1, 2, \cdots, n$), we will increment $B[A[i]]$. This step requires $O(n)$ time.

   (e) $B[j]$ for $j = 1, 2, \cdots, k$, now contains the number of elements of A having value j.

   (f) Make $C[1] = B[1]$. Then for each element l of array C $where$ $l = 2, 3, \cdots, k$ we will $computeC[l] = B[l] + B[l-1]$. This step takes $O(k)$ time.

   (g) To answer any query about how many of n integers fall into a range $[a \cdots b]$, we simply compute $C[b] - C[a] + B[a]$. This computation requires $O(1)$ time after the $O(n + k)$ pre-processing time.

4. 10 points.
   Short version: Use Merge Sort to sort the array in $\Theta(n \log n)$ steps (worst case). Then go through the array and for each element $A[i]$ use binary search to determine if $x - A[i]$ is in the array. Sorting takes time $\Theta(n \log n)$. Binary search takes time $O(\log n)$ and is executed n times. The total time is thus $\Theta(n \log n)$.

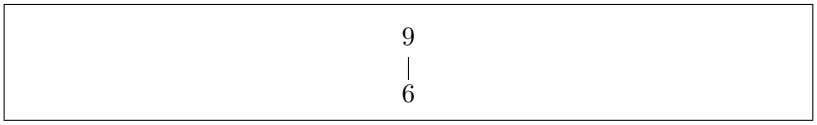
   Complete Algorithm:
   Input: Array S sorted in ascending order using merge sort, and x
   Output: *true* if there exist two elements in S whose sum is exactly x, $false$ otherwise.

Algorithm 2

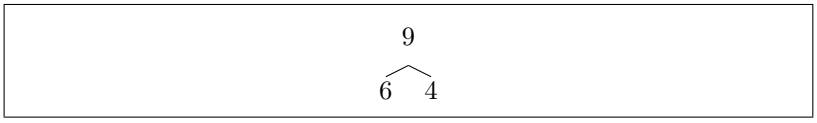
```
i ← 1
j ← n
while i ≤ j do
    if S[i] + S[j] = x then
        return true
    end if
    if S[i] + S[j] < x then
        i ← i + 1
    else
        j ← j − 1
    end if
end while
return false
```

5. 10 points.

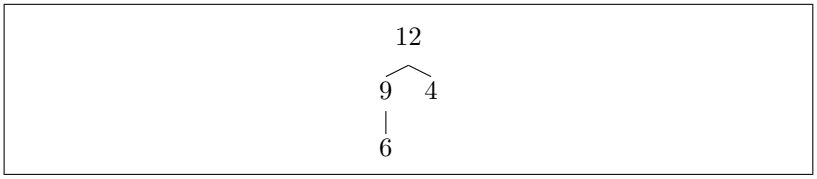
   (a) Inserting 6
   Inserting 9



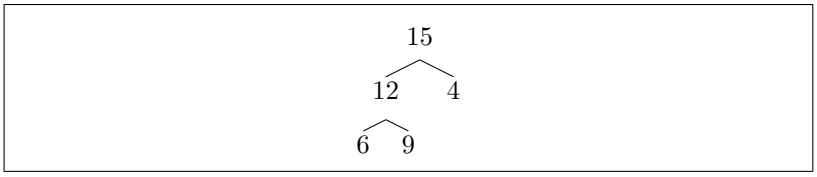
   Inserting 4



   Inserting 12



Inserting 15



(b) Removing 15 and then reordering