Exercises 2.3-6

Let an array of size *n*, which is in descending order. Consider each time we include an element *x* into the sorted part. Although we can use a modified binary search algorithm to find the correct position in $O(\log i)$ time (assume i is number of elements in the sorted part), the algorithm still needs to push all the elements in the sorted part one-position backward and then insert *x* into the first position of the array. So, when we include *x* we need to:

$O(\log i)$ to find the correct position

$O(i)$ to put it into the correct position

which is $O(i)$-time

In order to finish the insertion sort, all the elements are needed to be included into the sorted part:

$\Sigma_{i=2 \text{ to } n} O(i) = O(n^2)$

To conclude, even we use binary search instead of linear search, the overall worst-case running time of insertion sort is still $O(n^2)$ but not $\Theta(n \log n)$.

Exercises 2.3-7

The algorithm is as follows:

Assumptions:

S is in an array started from index 0

sizeof(S) = n

Algorithm:

```
1.  CheckExist(S, x)
2.      MergeSort(S, 0, sizeof(S) –1);
3.      for i = 0 to sizeof(S) – 1
4.          part_of_x = x – S[i]
5.          position = BinarySearch(S, 0, sizeof(S)-1, part_of_x)
6.          if position≠-1 and position ≠ i then
7.              return true
8.          end if
9.      end for
10.     return false
11. end CheckExist
```

Analysis:

line 2 takes Θ(n log n)-times (from analysis of merge sort)

Within for-loop 3:

line 4 takes Θ(1)-times

line 5 takes Θ(log n)-times (from analysis of binary search)

line 6-8 takes Θ(1)-time

line 4-8 takes Θ(1) + Θ(log n) + Θ(1) = Θ(log n)-time

line 3 takes Θ(n)-time (it iterates n-1 times)

line 3-9 takes Θ(n) * Θ(log n) = Θ(n log n)-time

line 10 takes Θ(1)-time

line 2-10 takes Θ(n log n) + Θ(n log n) + Θ(1) = Θ(n log n)-time

Exercises 2-4

a) (1, 5), (2, 5), (3, 5), (4, 5), (3, 4)

b) Array in the form {n, n-1, …, 1} has the most inversions. It has $\Sigma_{k=1 \text{ to } n-1} k = n(n-1)/2$ number of inversions.

c) Each pair of inversion (i, j) implies if we want to restore the position of the j-th elements the i-th element must be moved once. Insertion sort needs to restore the position of the 2^{nd} element to the n-th element. Combine this with our observation, the running time of insertion sort is directly proportion to the number of inversions.

d) *Observation:*

The inversion pair (i, j) should belongs to one of these three:

for $m = \lfloor (i + j) / 2 \rfloor$

1. both i and j are less then or equal to m
2. both i and j are greater then m
3. i is less then or equal to m and j is greater then m

Base on this observation, we design a recursive algorithm which will

1. count the number of inversion pair on the left half recursively
2. count the number of inversion pair on the right half recursively
3. count the number of inversion pair across the middle

Algorithm:

```
1.  Inversion(A, left, right)
2.      if (left < right) then
3.          middle = ⌊ (left + right) / 2 ⌋
4.          no_inver = 0
5.          no_inver = no_inver + Inversion(A, left, middle)
6.          no_inver = no_inver + Inversion(A, middle+1, right)
7.          no_inver = no_inver + InverAcrossMiddle(A, left, middle+1, right)
8.          return no_inver
9.      else
10.         return 0
11.     end if
12. end Inversion
```

```
1.  InverAcrossMiddle(A, p, q, r)
2.      n1 = q – p
3.      n2 = r – q + 1
4.      create array L[0..n1], R[0..n2]
5.      for i = 0 to n1-1 do L[i] = A[p+i]
6.      for j = 0 to n2-1 do R[j] = A[q+j]
7.      L[n1] = R[n2] = ∞
8.      i = j = 0
9.      no_inver = 0
10.     for k = p to r
11.         if L[i] > R[j] then
12.             if i ≠ n1 then
13.                 no_inver = no_inver + (n1-i)
14.             end if
15.             A[k] = R[j]
16.             j = j + 1
17.         else
18.             A[k] = L[i]
19.             i = i + 1
20.         end if
21.     end for k
22.     return no_inver
23. end InverAcrossMiddle
```

Proof of correctness:

1) **Proof of InvertAcrossMiddle**

Observe that because L and R are in sorted order, when L[i] > R[j] it implies that for L[k] > R[j] where i<=k<n1. So there are (n1-i) inversion pairs of (k, j). When L[i] <= R[j], it implies that there is no such inversion pair (i, j). The line 12 to 14 in InvertAcrossMiddle is to add the number of inversion pairs found when L[i] > R[j] is encountered. As a result this algorithm is used to count the number of inversion pairs in array A that the inversion pairs (i, j) that are in the form of

i is from p to q-1

j is from q to r

2) **Proof of Inversion**

let the size of input = right-left +1 = n

When n <= 1

by the algorithm it returns 0, which is correct because there is no inversion pair if the number of element in the array is less then or equal to 1

Assume the algorithm is correct for n <= k

When n = k + 1

1) call itself recursively on $\lceil (k+1)/2 \rceil$, the left half of the array, which by the assumption will give the number of inversion pairs on the left-half correctly

2) call itself recursively on $\lfloor (k+1)/2 \rfloor$, the right half of the array, which by the assumption will give the number of inversion pairs on the right-half correctly

3) call the InverAcrossMiddle on k+1 elements, by the result above, we know that it will give the number of inversion pairs across the middle of array.

4) by adding up these three numbers, and according to our observation before, we know that the algorithm will give us the total number of inversion pairs in the array A of size k+1

The algorithm is also correct for n = k + 1

By Mathematical induction the algorithm works for all n

Analysis:

1) InverAcrossMiddle

The only different with merge is line 12-14, which gives no effect on the running time analysis (Prove it by yourself). So InverAcrossMiddle takes $\Theta(n)$- times which n equals to the number of elements from p to r.

2) Inversion

let T(n) be the running time of Inversion on the array with n elements.

$T(1) = \Theta(1)$

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

By solving the above two equations, $T(n) = \Theta(n \log n)$