Jacob Alspaw
jaa134
EECS 340 - Algorithms
Assignment 1

2.3-4

```
//a recursive function to sort the first n items in array A using insertion
RecursiveInsertionSort(A, n)
    if n > 1                                          CONSTANT
        RecursiveInsertionSort(A, n - 1)              T(n − 1)
        Insert(A,n)                                   Θ(n)

//a function that will insert item j from array A into the correct location
Insert(A, j)
    key ← A[j]
    //insert A[j] into the sorted sequence A[1...j - 1]
    i ← j - 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i - 1
    A[i + 1] ← key
```

If we first analyze the method Insert(A, j), we can note the method simplifies to a worst-case linear run-time, $\Theta(n)$ as the while loop and its constituents dominates. If $T(n)$ is the total time to insertion sort an array with n elements and we recursively call insertion sort until the length of the inserted array is 1, then each recursive call has a run-time of $T(n-1)$. This means RecursiveInsertionSort(A, n) has a total simplified run-time of $T(n) = T(n-1) + \Theta(n)$ when the length of the list is greater than 1. If the length of the list is equal to one in size, then insertion sort runs in constant time.

2-2A

The remaining thing that we need to prove is that ALL elements from A are included in A' such that A' is a permutation of A where each original list item is not reused.

2-2B

Loop Invariant: Array A has n items. At the start of each iteration of the for loop of lines 2-4, A[j] is the minimum value in the subarray A[j...n] and the subarray A[j...n] is and will be a permutation (as described in part A) of the values along A[j...n] at the time the loop started.

Initialization: Subarray A[j...n] is initially just item A[j], which concludes that it is the minimum value of the subarray and a permutation of the initial subarray. The loop invariant holds true.

Maintenance: At the start of the iteration we shall denote j = j_0. By the loop invariant, A[j_0] is a minimum of A[j_0...n]. The next two lines, 3 and 4, will exchange item A[j_0] and A[j_0 - 1] under the condition that A[j_0] is less than A[j_0 - 1]. Under either success or failure of this condition, A[j_0 - 1] will now be the smallest number in the new subarrayA[j_0 - 1...n]. Because the values were swapped in their locations or weren't, the requirement set forth in part 2-2A is held true because no items are lost or duplicated. When j is decreased again, the next iteration will produce the same results.

Termination: The for-loop will terminate when j reaches the same value as i. A[i] will now be the minimum value is array A and A[i...n] will be a permutation of input array A to the standards set forth in part 2-2A.

2-2C

Loop Invariant: Array A has n items. At the start of each iteration of line 1's for-loop, the subarray A[1...i - 1] has, in sorted order, the i - 1 smallest values in input array A. The subarray A[i...n] will contain the remaining values of the input array.

Initialization: At the start of the first iteration when i = 1, the subarray A[1...i - 1] will have length zero, so the loop invariant is true.

Maintenance: At the start of the iteration we shall denote i = i_0. Our loop invariant holds that A[1...i_0 - 1] with i_0 of the smallest values in sorted order from input array A. From part B we know that the inner for-loop will push the smallest element from A[i_0...n] to A[i_0]. A[1...i_0] will now be the i_0 smallest values from A[1...n], in sorted order. Increasing i to i_0 + 1 after each loop iteration will make the loop invariant true until termination.

Termination: The for-loop on line one will terminate when i = n. The loop invariant will hold that A[1...i - 1] will contain the i - 1 smallest elements. The last element will be the largest element and will be at the end of the list in subarray A[i...n]. These two subarrays will make up the entire input array A by the loop invariant and will be in sorted order.

2-2D

```
BUBBLESORT(A)
    for i = 1 to A.length - 1                          n + 1
        for j = A.length downto i + 1                  n/2
            if A[j] ¡ A[j - 1]                         CONSTANT
                exchange A[j] with A[j - 1]            CONSTANT
```

From the above analysis of each line, we can see that the total run time is about $(n+1)*(n/2)*$(CONSTANT)$*$(CONSTANT) which simplifies down to $\Theta(n^2)$.