Algorithms and Data Structures

CSE 311

Fall 2020

Group 1 - Section 1

Assignment 2

Ahmed Moustafa AbdElfatah Mohamed Ramadan

Personal Email: geekahmed1@gmail.com

Problem 1: Insertion Sort with contents

Algorithm Pseudo-Code:

```
Insertion-Sort(L): // L is the list of integers which is zero-indexed
    for i = 1 to L.size:
        key = L[i]
        j = i - 1
        while j >= 0 and L[i] >= key:
            L[j+1] = L[j]
            j = j - 1
        L[j-1] = key
```

Implementation in JAVA:

Here you could find the implementation using JAVA:

https://drive.google.com/file/d/1vMuF3b1FdF4PW4YXpPgaJJGWuV92mben/view?usp=sharing

Problem 2: Insertion Sort for descending order

Algorithm Pseudo-Code:

```
Insertion-Sort(L): // L is the list of integers which is zero-indexed
    for i = 1 to L.size:
        key = L[i]
        j = i - 1
        while j >= 0 and L[i] <= key:
            L[j+1] = L[j]
            j = j - 1
        L[j-1] = key
```

Java Implementation

Here you could find the implementation using JAVA:

https://drive.google.com/file/d/1WDJp_0LmoErzkaJoabKNFPyfh9W3SLdb/view?usp=sharing

Problem 3: Searching Problem

Linear Search:

Algorithm Pseudo-Code:

```
Linear-Search(L,V):
    for element in L:
        if element == V:
            return element index
    return Null
```

Analysis:

In this searching algorithm we check each element with our value so it is called linear search. The Best case of that searching algorithm is to find the element in the first index. The Worst case is to find the element in the last index or didn't find it anymore.

So, the running time is in order of n. O(n)

Binary Search:

Algorithm Pseudo-Code:

```
Binary-Search-Iterative(L,V):
    left = 0
    right = L.size - 1
    while left <= right:
        mid = left + right / 2
        if L[mid] == V : return mid
        if L[mid] < V : left = mid + 1
        if L[mid] > V : right = mid - 1
    return Null
```

```
Binary-Search-Recursive(L,V, left, right):
    if left <= right:
        mid = left + right / 2
        if L[mid] == V : return mid
        if L[mid] > V : return Binary-Search-Recursive(L,V, left, mid - 1)
        return Binary-Search-Recursive(L,V, mid + 1, right)
    return Null
```

Analysis:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

We basically ignore half of the elements just after one comparison.

1. Compare v with the middle element.
2. If v matches with the middle element, we return the mid index.
3. Else If v is greater than the mid element, then v can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (v is smaller) recur for the left half.

The time complexity of Binary Search can be written as: $T(n) = T(n/2) + c$

Using Master Theorem or Recurrence tree, we find that the running time is $O(\lg n)$

Java Implementation

Here you could find the implementation using JAVA for both algorithms:

https://drive.google.com/file/d/1-dYdpdSgeHhXq8QI1A3eBevqH9r0WjFF/view?usp=sharing

Problem 4: Selection Sort

Pseudo-Code:

```
Selection-Sort(L):
    for i = 0 to L.size - 1:
        minIndex = i
        for j = i + 1 to L.size:
            if L[j] < L[minIndex]: minIndex = j
        temp = L[minIndex]
        L[minIndex] = L[i]
        L[i] = temp
```

Analysis:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Time complexity: **the worst case** is that the array is in reverse order so the running time is in order of n^2. The time complexity is: O(n^2).

The average case is also O(n^2) as the algorithm is independent of the data distribution.

Java Implementation:

Here you could find the implementation using JAVA:

https://drive.google.com/file/d/1SHqQB04AnilHD_BOQBCscV8AnMdT_4Wz/view?usp=sharing

Problem 5: Merge Sort

Pseudo-Code:

```
Merge-Sort(L, left, right):
        if left < right:
                mid = (left + right) / 2
                Merge-Sort(L, left, mid)
                Merge-Sort(L, mid + 1, right)
                Merge(L, left, mid ,right)
Merge(L, left, mid, right):
        n1 = mid - left + 1
        n2 = right - mid
        Create LeftArray of size n1 and rightArray of size n2
        for i = 0 to n1: leftArray[i] = L[left + i]
        for j = 0 to  n2: rightArray[j] = L[mid + 1 + j]
        i = 0
        j = 0
        k = left
       while (i < n1 && j < n2):
          if (leftArray[i] <= rightArray[j]):
              L[k] = leftArray[i]
              i++
          else:
              L[k] = rightArray[j]
              j++
              k++
```

```
while (i < n1):
    L[k] = leftArray[i]
    i++
    k++
while (j < n2):
    L[k] = rightArray[j];
    j++
    k++
```

Analysis:

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is merge step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

Have we reached the end of any of the arrays?

No:

Compare current elements of both arrays

Copy smaller element into sorted array

Move pointer of element containing smaller element

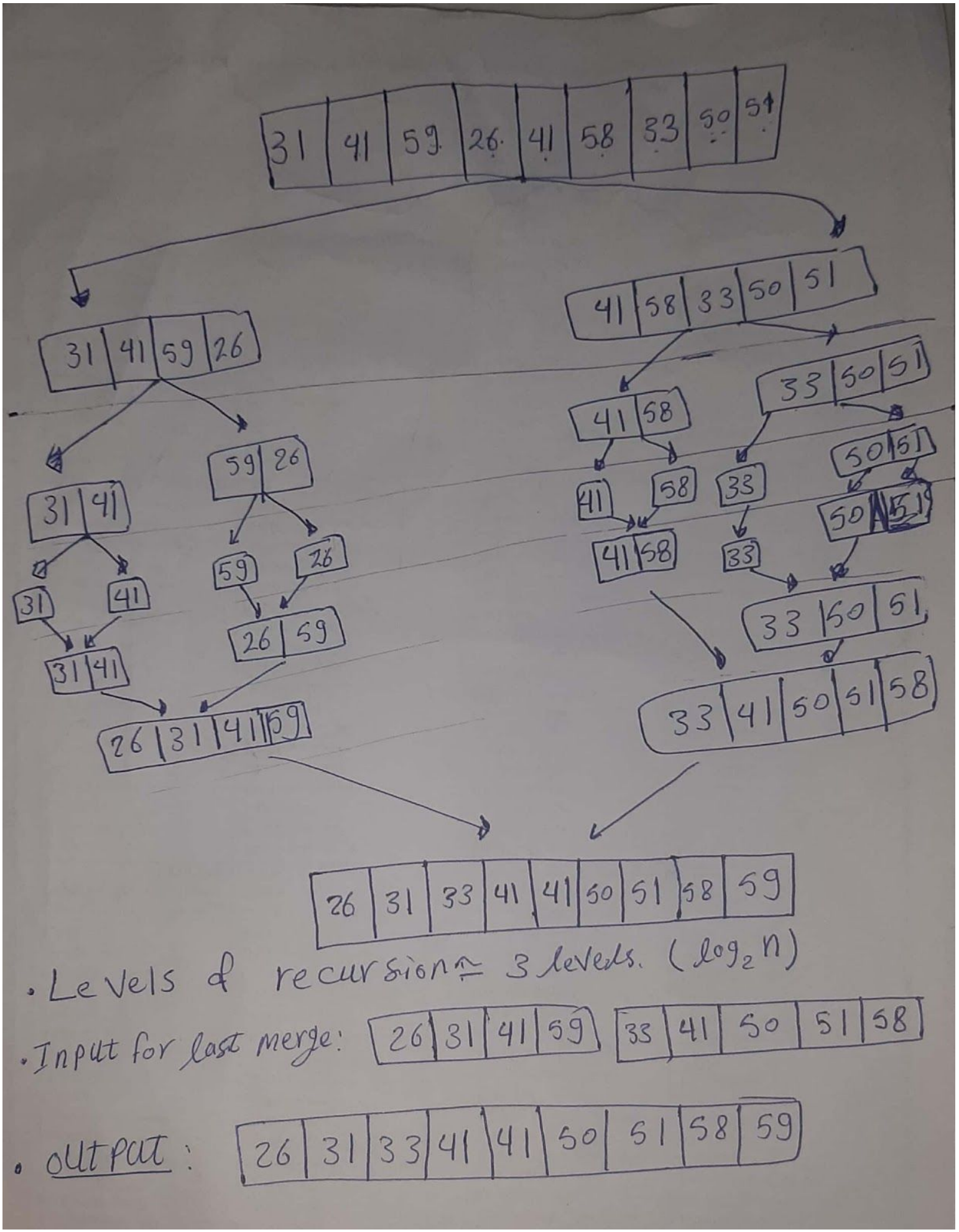
Yes:

Copy all remaining elements of non-empty array

Time Complexity: Merge Sort is a recursive algorithm and time complexity can be expressed as the following recurrence relation: $T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. The solution of the recurrence is $\theta(nLogn)$. Time complexity of Merge Sort is $\theta(nLogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Example Array [31,41,59,26,41,58,33,50,51]:



Java Implementation:

Here you could find the implementation using JAVA:

https://drive.google.com/file/d/1zf1OqJuhINVEGo443vMzr4dDzBQbZIx4/view?usp=sharing

Problem 6: Merge Sort and Insertion Sort

Java Implementation:

Here you could find the implementation using JAVA:

https://drive.google.com/file/d/1TkHXJn_17XaptxHO84EeDHepehEDy9fm/view?usp=sharing

My Comment:

Merge sort has less time complexity than insertion sort and it is more efficient. It has a very small number of comparisons compared to insertion sort.