DAA Lab - Session 7 - Insertion Sort and comparison of sorting algorithms

Decrease-and-Conquer: Implementation of **Insertion Sort** algorithm

**Problem Definition:**

Sort an array of student records (a record is a structure with a “serial number” and a “score” field) using Insertion Sort in nondecreasing order on the “serial number” field of the records. Compare the running time and number of element-to-element comparisons with the implementations of Bubble Sort, Selection Sort, Merge Sort and Quick Sort algorithms.

**Input:** Input begins with n (1 ≤ n ≤ 220) of number of records indicating the size of the input array. The following n lines has a record per line with a 9-digit “serial number” field and a 4-digit integer “score” field separated by a space.

**Output:** Output to have five lines, one line for each of Insertion Sort, Bubble Sort, Selection Sort, Merge Sort and Quick Sort algorithms. A line to have the name of the algorithm, element-to-element comparison count and running time (in seconds upto 6 decimal places) separated by spaces as as shown in the sample output. For the set of 8 test-cases of random numbers (of input sizes 32k, 64k, 96k, 128k, 160k, 192k, 224, and 256k), plot two curves of an algorithm per graph sheet; one curve for the comparison counts and the other for the running times. So five graph sheets for the set of 8 test-cases to be used. Let the x-axis to have input sizes 32k per unit, and two different units for the y-axis; one being comparison count and the other being time in seconds. Mention about the correlation between comparison counts and running times in your conclusion.

**Sample Input:**

5

2123456 100

1234 92

1 1

123123123 9999

9 99

**Sample Output:**

Insertion Sort: 6 0.000000

Bubble Sort: 7 0.000000

Selection Sort: 10 0.000000

Merge Sort: 6 0.000002

Quick Sort: 7 0.000001

**Algorithms:**

**Algorithm InsertionSort(A[0..n-1])**

for i ← 1 to n-1

temp ← A[i]

j ← i-1

while(j ≥ 0 and **A[j] > temp**)

A[j+1] ← A[j]

j ← j-1

A[j+1] ← temp

**Algorithm BubbleSort(A[0..n-1])**

for *i* ← 0 to n - 2

noSwaps ← TRUE

for j ← 0 to *n - 2 - i*

*if(****A[j] > A[j+1]****)*

*swap A[j] and A[j+1]*

noSwaps ← FALSE

if (noSwaps = TRUE) return

**Algorithm SelectionSort(A[0..n-1])**

for i ← 0 to n-2

min ← i

for j ← i+1 to n-1

if(**A[j] < A[min]**) min ← j

Swap A[i] with A[min]

**Algorithm MergeSort(A[0..n-1])**

if(n ≤ 1)return

m = ⌊n/2⌋

MergeSort(A[0..m-1])

MergeSort(A[m..n-1])

Merge(A[0..n-1], m)

**Algorithm Merge(A[0..n-1], m)**

i ← 0, j ← m, k ← 0

while(i < m and j < n) do

if(**A[i] ≤ A[j]**) B[k] ← A[i]; i ← i+1

else B[k] ← A[j]; j ← j+1

k ← k+1

if(j = n) Copy A[i..m-1] to B[k..n-1]

else Copy A[j..n-1] to B[k..n-1]

Copy B[0..n-1] to A[0..n-1]

**Algorithm QuickSort(A[0..n-1])**

if(n ≤ 1) return

s ← Partition(A[0..n-1])

QuickSort(A[0..s-1])

QuickSort(A[s+1..n-1])

return

**Algorithm Partition(A[0..n-1])**

if(A[n/2] is the median of A[0], A[n/2] and A[n-1]) swap A[0], A[n/2]

if(A[n-1] is the median of A[0], A[n/2] and A[n-1]) swap A[0], A[n-1]

//Don’t count the element-to-element comparison above

i ← 1, j ← n-1

while(i ≤ j)

while(i ≤ j and **A[i] < A[0]**) i ← i + 1

while(i ≤ j and **A[j] > A[0]**) j ← j - 1

if(i < j)

swap A[i], A[j]

i ← i + 1

j ← j - 1

swap A[j], A[0]

return j

**References:** A video of the intro session is available at <https://youtu.be/jN30GM61MBQ> under the title “Introductory session on the lab in Design and Analysis of Algorithms”.

**Practice-Problems:**

1. Just like the set of 8 test-cases of random numbers, generate sets of test-cases with sorted numbers, sorted in reverse order, almost sorted, a lot of repeated entries, etc. Some algorithms are expected to perform better than the others on specific kinds of inputs.