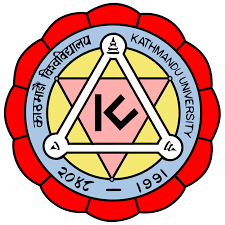
# KATHMANDU UNIVERSITY

DHULIKHEL, KAVREPALANCHOWK, NEPAL

****

**SUBJECT CODE: COMP 314**

*In the partial fulfilment of “Introduction to Sorting Algorithm-Insertion and Merge Sort”*

**Lab Report #2**

**Submitted To:**

Rajani Chulyadyo, PhD

Department of Computer Science and Engineering (DoCSE)

**Submitted By:**

Kamal Shrestha

Roll No.:49

6th Semester, 3nd Year

B.E. Computer Engineering

**Submission Date**: 13th May 2019

Link to GitHub: <https://github.com/KamalShrest/CE_III_49_Lab2>

**Lab 2: Sorting (Insertion Sort and Merge Sort)**

**Objectives:**

* Implementation the following sorting algorithms:  
  (a) Insertion sort  
  (b) Merge sort
* Implementation of some test cases in the algorithms.
* Generation of random inputs for the program and applying both Insertion

Sort and Merge Sort algorithms to sort the generated sequence of data.

* Record the execution times of both algorithms for best and worst cases

on inputs of different size.

* Plot an input-size vs execution-time graph.
* Explanation for the observations

**Introduction: Sorting Algorithms**

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.  
Some of the common sorting algorithms include Selection sort, Bubble sort, Recursive Bubble sort, **Insertion Sort, Merge sort**, Quick Sort, Heap Sort (among the comparison sorting algorithms), similarly Counting Sort, Radix Sort, Bucket Sort, Shell Sort (among the non-comparison sorting algorithms).

It is important if you need an algorithm to be consistent with your performance requirements. For example, if your algorithm has a best-case of O(n lg n) but a worst-case of (n^2) performance, and you might expect the amount of data input to increase ten-fold over the next year, then favouring worst-case running time will provide you with assurance that your algorithm will not lose its quality later.

**Stable sort vs. Unstable sort**

A stable sort is a sort which guarantees that when two items are compared and determined to be the same, their positions in the set will not be compromised. Whereas for unstable sort the positions of the items may change given the nature of the sort.

**Insertion Sort Algorithm:**

**Features:**

* It is efficient for smaller data sets, but very inefficient for larger lists.   
  Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
* It is better than Selection Sort and Bubble Sort algorithms.
* Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
* It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.

**Algorithm:**

**Step 1** − If it is the first element, it is already sorted. Return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than the

Value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

**Pseudo Code:**

procedure insertionSort( A : array of items )

int holePosition

int valueToInsert

for i = 1 to length(A) inclusive do:

/\* select value to be inserted \*/

valueToInsert = A[i]

holePosition = i

/\*locate hole position for the element to be inserted \*/

while holePosition > 0 and A[holePosition-1] > valueToInsert do:

A[holePosition] = A[holePosition-1]

holePosition = holePosition -1

end while

/\* insert the number at hole position \*/

A[holePosition] = valueToInsert

end for

end procedure

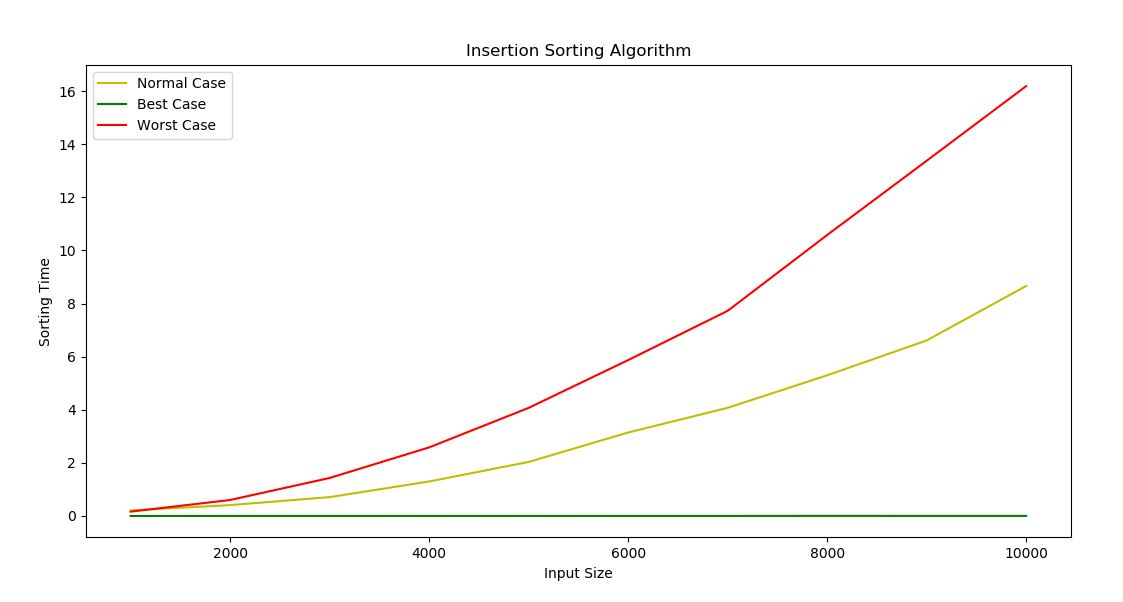
**Python Code:**





**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Size** | **Normal Case** | **Best Case** | **Worst Case** |
| 1000 | 0.08395099639892578 | 0.0010006427764892578 | 0.1599104404449463 |
| 2000 | 0.2788405418395996 | 0.0019996166229248047 | 0.625640869140625 |
| 3000 | 0.6806111335754395 | 0.0019981861114501953 | 1.335254192352295 |
| 4000 | 1.147357702255249 | 0.0029969215393066406 | 3.163175344467163 |
| 5000 | 1.9129068851470947 | 0.003998756408691406 | 4.023701429367065 |
| 6000 | 2.683483600616455 | 0.001997232437133789 | 5.981588363647461 |
| 7000 | 4.000710964202881 | 0.0019989013671875 | 8.226300716400146 |
| 8000 | 5.708739519119263 | 0.0029990673065185547 | 10.973678588867188 |
| 9000 | 6.714166164398193 | 0.0049970149993896484 | 13.931021451950073 |
| 10000 | 8.52612566947937 | 0.0029964447021484375 | 17.432392835617065 |

**Graph:**

**Interpretation of Observations:**

Hence the time complexities for insertion sorting algorithm are: O (n2) in Average and Worst case where as O(n) in best case scenario.

**Merge Sort Algorithm:**

**Divide and Conquer in Merge Sort:**

* **Divide** by finding the number q of the position midway between p and r Do this step the same way we found the midpoint in binary search: add p and r divide by 2, and round down.
* **Conquer** by recursively sorting the subarrays in each of the two sub problems created by the divide step. That is, recursively sort the subarray array[p..q] and recursively sort the subarray array[q+1..r].
* **Combine** by merging the two sorted subarrays back into the single sorted subarray array[p..r].

**Algorithm:**

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

**Pseudo Code:**

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

while ( b has elements )

add b[0] to the end of c

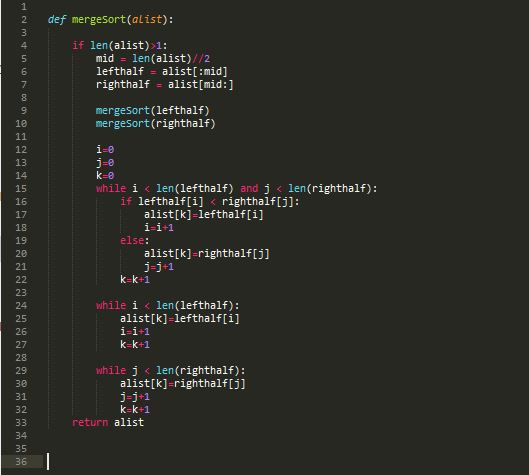
remove b[0] from b

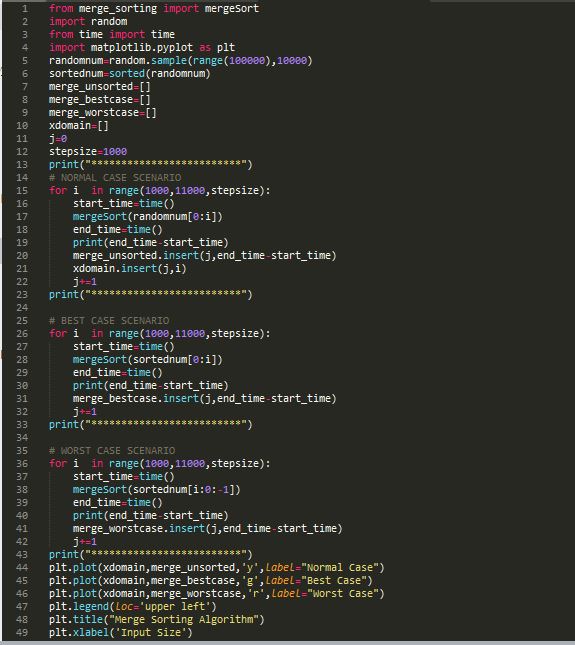
end while

return c

end procedure

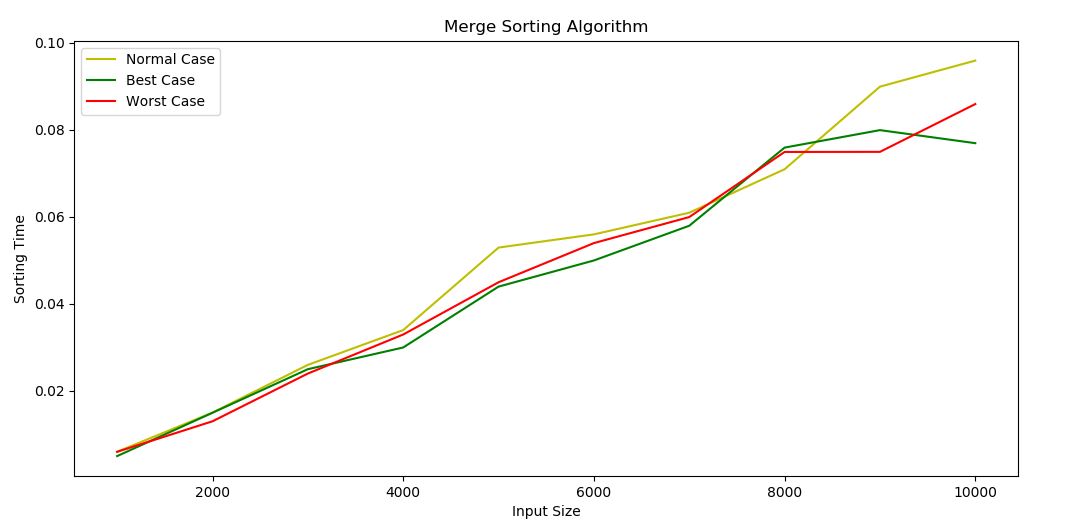
**Python Code:**

****



**Output:**

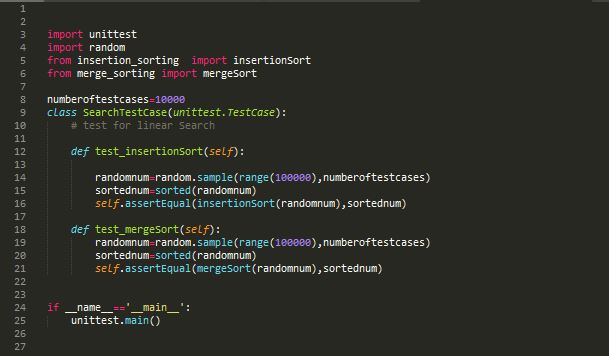
|  |  |  |  |
| --- | --- | --- | --- |
| **Input Size** | **Average Case** | **Best Case** | **Worst Case** |
| 1000 | 0.008983850479125977 | 0.007961511611938477 | 0.008978843688964844 |
| 2000 | 0.01898789405822754 | 0.019970417022705078 | 0.018988847732543945 |
| 3000 | 0.031975507736206055 | 0.020008563995361328 | 0.021990537643432617 |
| 4000 | 0.034993648529052734 | 0.029983043670654297 | 0.035979270935058594 |
| 5000 | 0.044974327087402344 | 0.036978960037231445 | 0.04398822784423828 |
| 6000 | 0.052987098693847656 | 0.0479741096496582 | 0.05098772048950195 |
| 7000 | 0.06094765663146973 | 0.05097341537475586 | 0.0579831600189209 |
| 8000 | 0.08095383644104004 | 0.08698058128356934 | 0.06696200370788574 |
| 9000 | 0.0919656753540039 | 0.08093881607055664 | 0.08095359802246094 |
| 10000 | 0.09094762802124023 | 0.08295106887817383 | 0.08393406867980957 |

**Graph:**

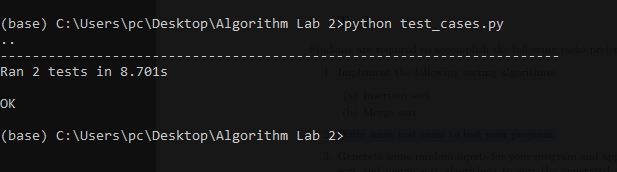
**Interpretation of Observations:**

Hence the time complexities for insertion sorting algorithm are: O (nlogn) in Average and Worst case and best case scenario.

**Test Cases:**

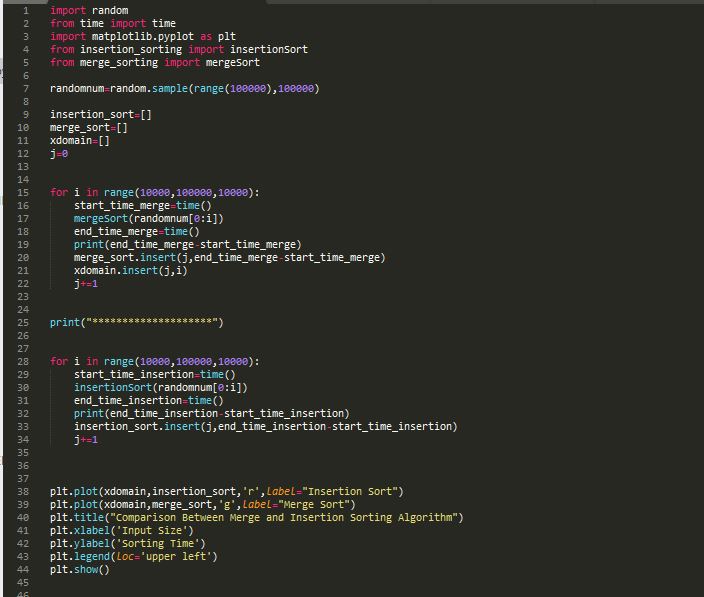
****

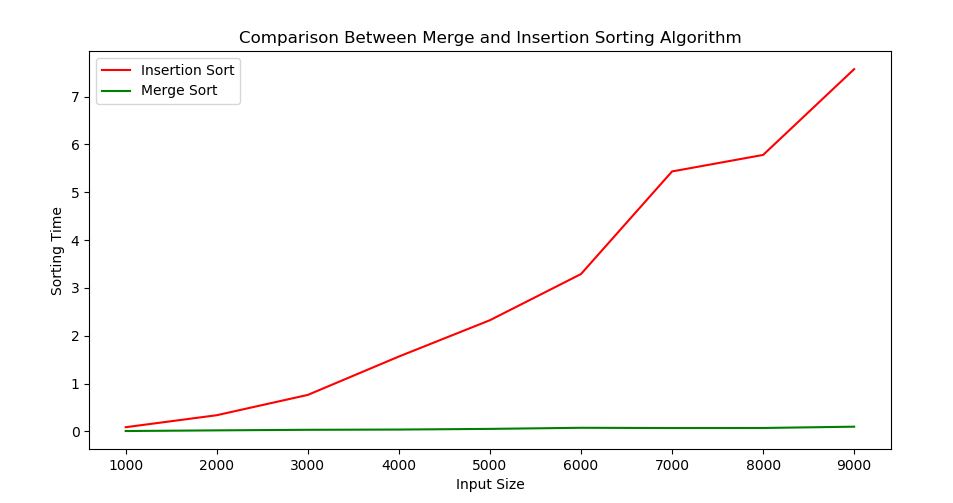
**Output:**

****

**Comparison between the sorting algorithms:**

Comparison between the sorting algorithms were done on the same range and same elements for average case scenario:



**Output Graph**