**Insertion Sort :**

Insertion Sort is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of inputs given.

The algorithm runs in O(n) for the already sorted input.

**Merge Sort :**

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n) where n is the number of inputs given, it is one of the efficient sorting algorithms.

The merge sort takes almost same time in all the three cases of input.

**Quick Sort :**

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the pivotal value based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is efficient for large-sized data sets as its average and worst case complexity are of Ο(n2), where n is the number of inputs given.

The median quicksort is much better algorithm compared to in-place quicksort as it takes very less time to sort the input because the pivot is always the last element in the input for in-place quicksort.

**Results :**

We have generated random numbers using random python module and saved this sequence to a file named “random.txt” . This file consists of 50000 random numbers which will be used as the input to run the algorithms.

**Note : All the algorithms are tested under following hardware specifications :**

**Ram : 2 GB**

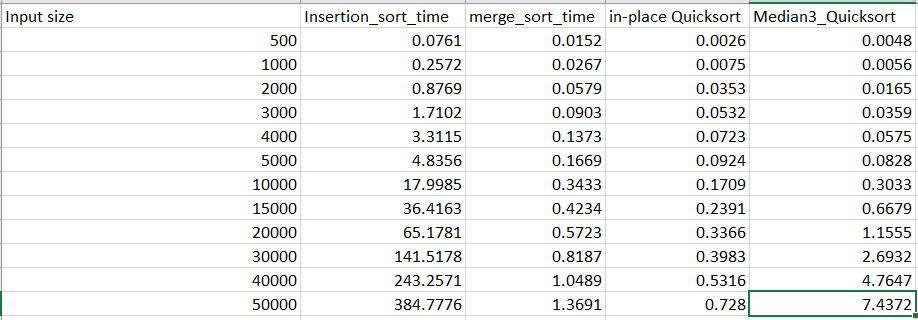
**Operating systems : Ubuntu 16.4**

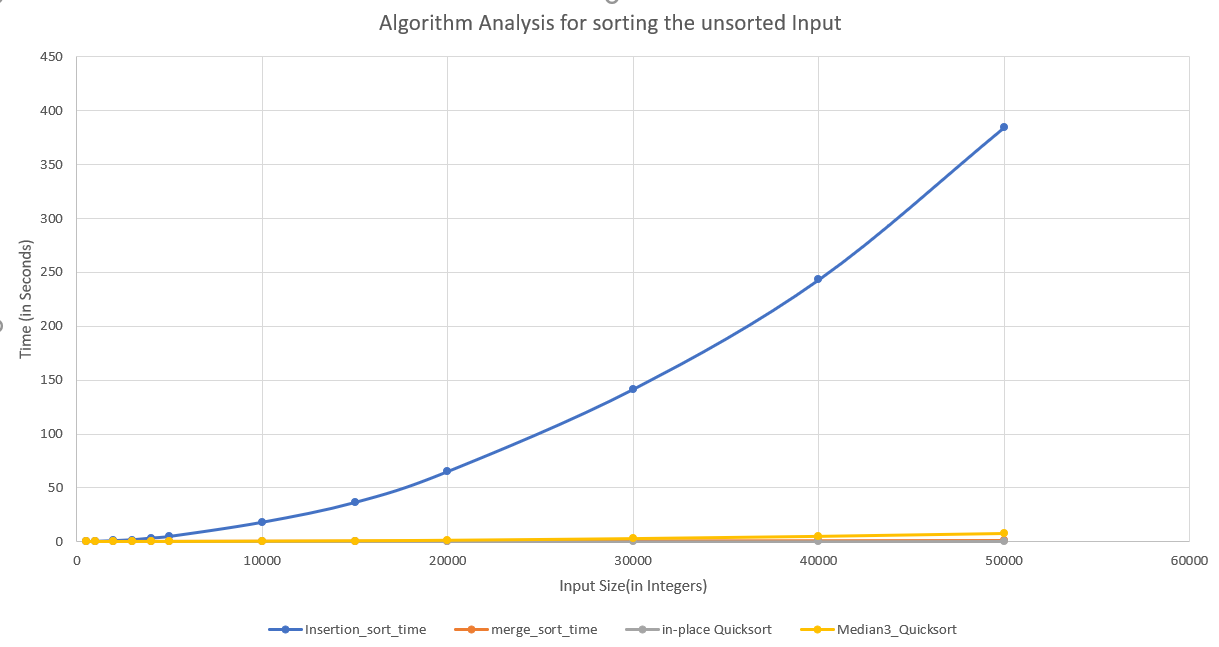
**Python IDE : python 2.7**

**# CPU : 1 cycle**

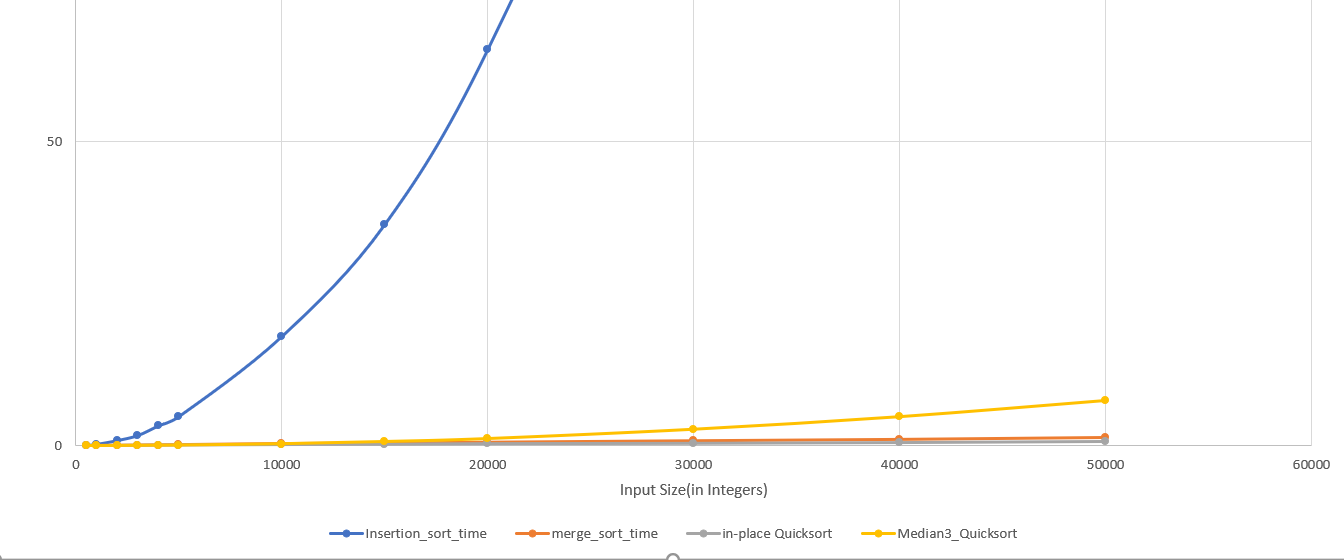
**Microprocessor : Intel core i7 8th generation**

Sorting the unsorted input



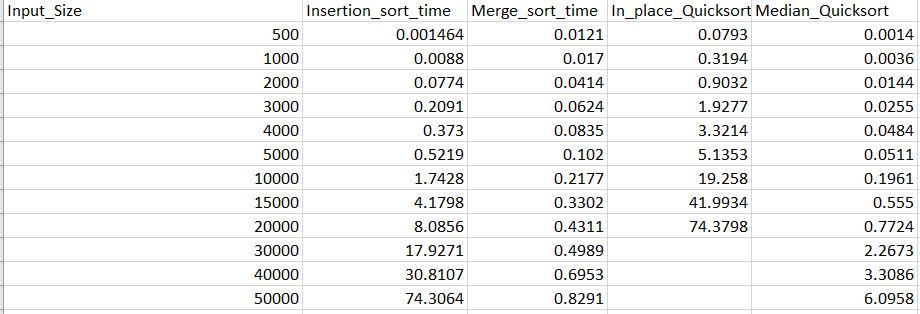


Enlarged View of the above Graph.



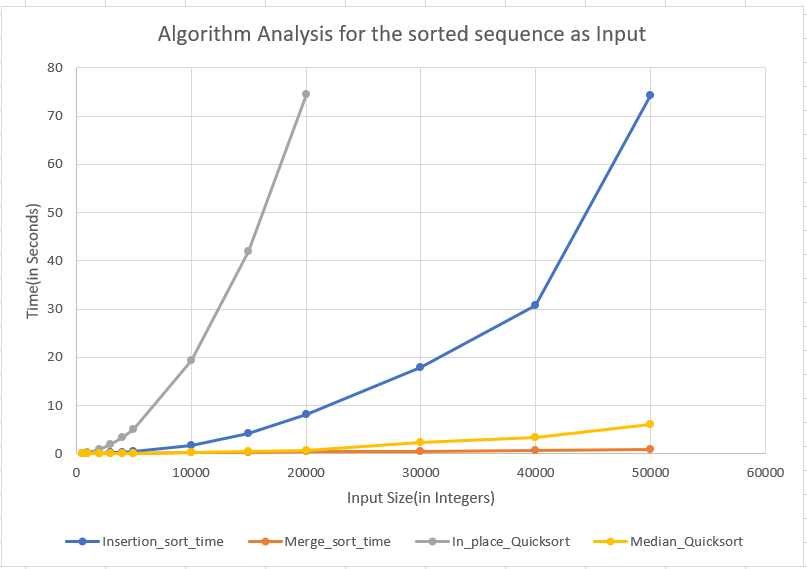
Special cases

2a : Sorted sequence as input

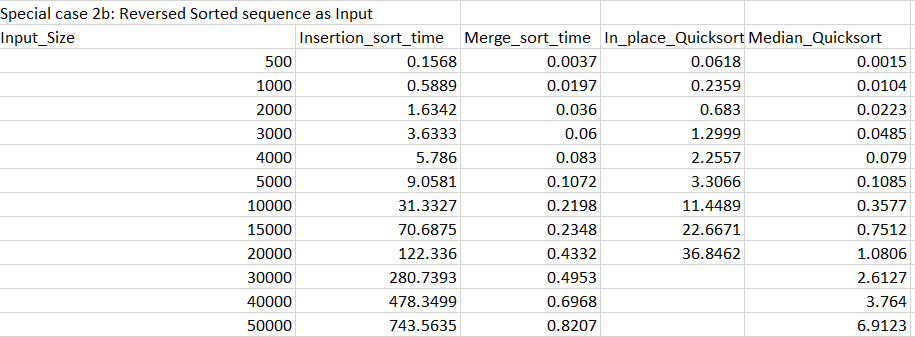
****

As seen in the above table, the in\_place\_quicksort algorithm stops at 20000 input size due to segmentation fault error which is due to memory size overflow. We tried to increase the heap size by setting the Xss256m which didn’t work fine.

Graph for the above table.

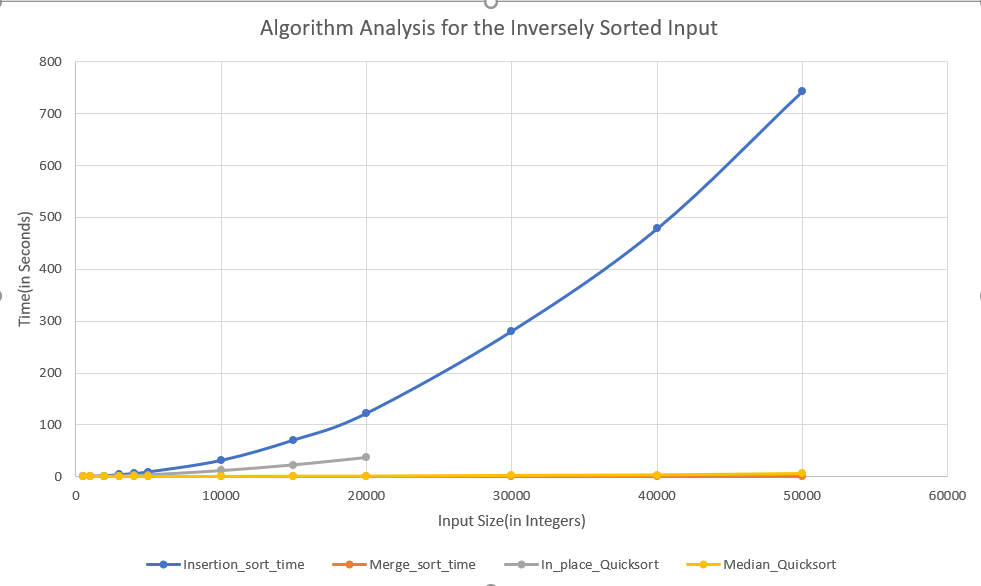
****

2b : Sorted sequence in reverse order as input



As seen in the above table, the in\_place\_quicksort algorithm stops at 20000 input size due to segmentation fault error which is due to memory size overflow. We tried to increase the heap size by setting the Xss256m which didn’t work fine.

Graph for the above table.



**Code of Insertion Sort**

def insertion\_sort(lst):

for i in range(1,len(lst)):

for j in range(i-1,-1,-1):

if lst[j] > lst[j+1]:

lst[j],lst[j+1] = lst[j+1],lst[j]

else:

Break

**Code of Merge Sort**

def mergesort(inp):

if len(inp)>1:

mid = len(inp)//2

i = 0

lefthalf = []

while(i != mid):

lefthalf.append(inp[i])

i += 1

j = mid

righthalf = []

n = len(inp)

while(j != n):

righthalf.append(inp[j])

j += 1

mergesort(lefthalf)

mergesort(righthalf)

''' Merging Code'''

i = j = k = 0

while i < len(lefthalf) and j < len(righthalf):

if lefthalf[i] < righthalf[j]:

inp[k]=lefthalf[i]

i=i+1

else:

inp[k]=righthalf[j]

j=j+1

k=k+1

while i < len(lefthalf):

inp[k]=lefthalf[i]

i=i+1

k=k+1

while j < len(righthalf):

inp[k]=righthalf[j]

j=j+1

k=k+1

**Code of In-place Quicksort**

def partition(lst,first,last):

pivot = lst[last]

i = first - 1

for k in range(first,last):

if lst[k] <= pivot:

i += 1

lst[i],lst[k] = lst[k],lst[i]

lst[i+1],lst[last] = lst[last],lst[i+1]

return i+1

def quicksort(unsorted\_list,first,last):

if last > first:

pivot = partition(unsorted\_list,first,last)

quicksort(unsorted\_list,first,pivot-1)

quicksort(unsorted\_list,pivot+1,last)

**Code of Median Quicksort**

def insertion\_sort(a,low,high):

for ele in range(low+1,high+1):

key = a[ele]

j = ele-1

while(j>=0 and key < a[j]):

a[j+1] = a[j]

j -= 1

a[j+1] = key

def decide\_algo(a,low,high):

if(low + 10 <= high):

if low<high:

piv = partition(a,low,high)

decide\_algo(a,low,piv-1)

decide\_algo(a,piv+1,high)

else:

insertion\_sort(a,low,high)

def quicksort(a,low,high):

decide\_algo(a,low,high)

def median(a, low, high, median):

if a[low] < a[high]:

return high if a[high] < a[median] else median

else:

return low if a[low] < a[median] else median

def partition(a,low,high):

pindex = median(a, low, high, (low + high) // 2)

a[low], a[pindex] = a[pindex], a[low]

pivot = a[low]

left = low

right = high

flag = False

while not flag:

while left <= right and a[left] <= pivot:

left = left + 1

while a[right] >= pivot and right >= left:

right = right -1

if right < left:

flag = True

else:

a[left],a[right] =a[right],a[left]

temp = pivot

a[a.index(pivot)] = a[right]

a[right] = temp

return right

HOW TO RUN THE PROGRAM

$python file\_name.py

ENTER THE INPUT SIZE

$500

DISPLAYS **3** OUTPUT TIMES

OUTPUT LINE1 🡪 Time taken to sort the unsorted input sequence.

OUTPUT LINE2 🡪 Time taken to sort the sorted input sequence.

OUTPUT LINE3 🡪 Time taken to sort the reversed sorted input sequence.

