Insertion Sort, Merge sort and Hybrid Merge-insertion sort.

Report written by:
Pratik Antoni Patekar (1001937948)

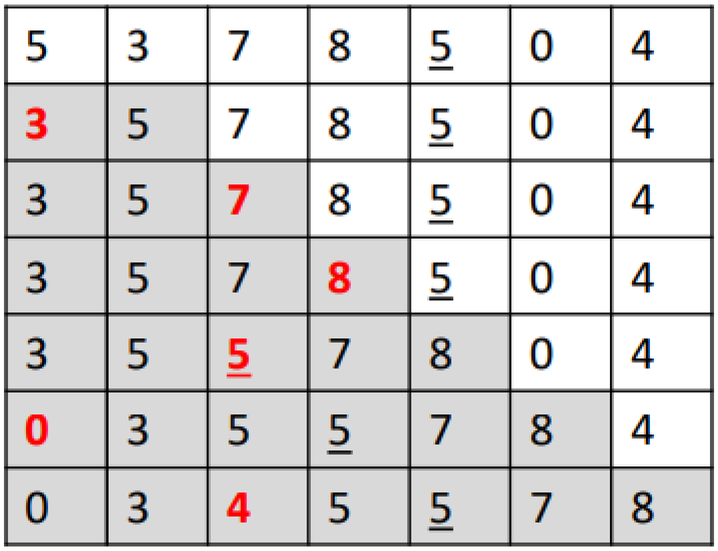
Table of contents

1.0. Introduction:

In this report we will be discussing the two popular sorting algorithms namely insertion sort and merge sort. And after that we will discuss their time complexity analysis.
Later we discuss which one of the two algorithms is better and how their advantages can be combined into one. This algorithm that combines the advantages of insertion and merge sort is called the ‘hybrid merge-insertion sort algorithm’.
We then discuss the running time analysis of hybrid merge-insertion sort and show the code implementation and output results (time required comparisons of all three algorithms).

2.0. Sorting algorithms:

2.1. Insertion sort:
In insertion sort, the input array is processed/ sorted from left to right. At every step it sorts one item more compared to the previous step.



5	3	7	8	5	0	4
3	5	7	8	5	0	4
3	5	7	8	5	0	4
3	5	7	8	5	0	4
3	5	5	7	8	0	4
0	3	5	5	7	8	4
0	3	4	5	5	7	8

We start with the second element in the input array and check whether it is smaller or greater than the element in the left. If the second element (in this case = 3) is smaller than the first element, then we send it to the left (i.e. swap the elements). Now these two elements on the left are sorted as part of the input array.

In the next iteration, we check where the third element in the input array can be placed in this sorted left part of the array (gray shaded portion). This is repeated in every iteration. In each iteration we check and insert the new element from the right unsorted part of the input array to the left sorted part of the input array. By the time we reach the last element, the whole input array is sorted.

2.1.1. Pseudo code for insertion sort:
Assumption: A is the list of numbers to be sorted and n is the size of the array.

```
Insertion sort(A, n):
    for i = 1 to n
        key = A[i]
        k = i - 1
        while k >= 0 and A[k] > key:
            A[k+1] = A[k]
            k = k - 1
        A[k+1] = key
```

2.1.2. Running time analysis of insertion sort:

There are different methods and algorithms to find the time complexity of an algorithm, but we will be using **proof by loop invariant** method here. The proof by loop invariant includes 3 steps:

1. Initialization
2. Maintenance
3. Termination

We first need to find the loop invariant here. If we see the image in the insertion sort algorithm, we can see that the part of the array to the right of the key is always sorted. This is nothing but the loop invariant here.
While calculating the time complexity, we generally calculate the time complexity for the worst scenario. Here the worst-case scenario is when the list of numbers is in the reverse order. Now assuming that the list of numbers is in reverse order we can say that,
(Initialization)
In first iteration, there is only one data move required - 1
In second iteration, there are 2 data moves required - 2
(Maintenance)
This is goes on till the last element
(Termination)
In nth iteration there are n data moves required - n
Thus in total, the number of data moves required are - (1 + 2 + 3 + ….. + n) = $\frac{n(n+1)}{2}$

Therefore the time complexity $T(n) = \frac{n(n+1)}{2} = O(n^2)$ ….(using Asymptotic analysis notation discussed at the end).

Insertion sort uses in-place memory i.e., it does not require extra memory therefore the space complexity is $O(n)$.

Note that here runtime depends on input order and input size.

2.1.3. Best case, worst case and average case analysis:
As mentioned above the runtime depends on the input order. Depending on this there are three possible scenarios and thus 3 different time complexity analysis possible. They are best case, worst case and average case.

1. **Best case:** If the items in the array are already sorted then the while loop in the code will never be executed and thus the number of iterations will be N. Thus resulting in the best case TC to be $\Theta(n)$.
2. **Worst case:** If the items in the array are exactly in reverse order then the while loop will be executed every time to its maximum (N). Thus, resulting in the worst-case TC to be $\Theta(n^2)$.
3. **Average case:** It is the average of best and worst cases or in other words, it is the scenario when half of the elements are sorted, and another half aren't. Thus, the TC for average case is $(\Theta(n) + \Theta(n^2))/2 = \Theta((n + n^2)/2) = \Theta(n^2)$.

2.2. Merge sort:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list

Approach:

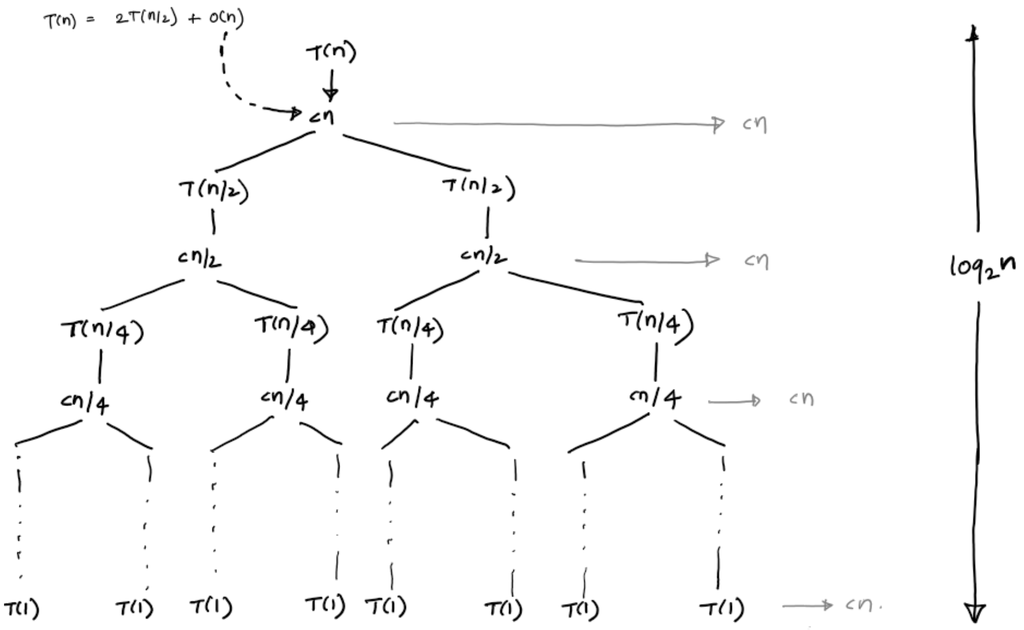
1. If n = 1 then done. ($O(1)$)
2. If n > 1 then recursively sort i.e., divide the unsorted list into 2 sub lists each time and recursively sort each sub list till the base case is reached. ($2T(n/2)$)
3. Once each sublist is sorted then merge the two sub lists such that the resulting merged list is sorted. ($O(n)$)

Running time $T(n) = 2T(n/2) + O(n)$
This recursive relation can be solved using recursion tree method (shown after Pseudocode)

2.2.1. Pseudocode for merge sort:
Assumption: A is the list of numbers to be sorted and n is the size of the array where p is start index and r is end index.

```
Merge_sort(A, p, r)
    if p < r
        q = ⌊(p + r)/2⌋
        Merge-sort (A, p, q)
        Merge-sort (A, q+1, r)
        Merge (A, p, q, r)
Merge (A, p, q, r)
    $n_1 = q - p + 1$
    $n_2 = r - q$
    Let $L[1, ..., n_1 + 1]$ and $R[1, ..., n_2 + 1]$
    for $i = 1$ to $n_1$
        $L[i] = A[p + i - 1]$
    for $j = 1$ to $n_2$
        $R[j] = A[q + j]$
    $L[n_1 + 1] = \infty$
    $R[n_2 + 1] = \infty$
    $i = 1$
    $j = 1$
    for $k = p$ to $r$
        if $L[i] \leq R[j]$
            $A[k] = L[i]$
            $i = i + 1$
        else $A[k] = R[j]$
            $j = j + 1$
```

2.2.2. Recursion tree for $T(n) = 2T(n/2) + O(n)$:



Now at each level we can see that the sum is cn and the height of the recursion tree is $log_2 n$.

Therefore we can say that the sum of complexity from each level is,

$$T(n) = 2T(n/2) + O(n) = \sum_{i=0}^{log_2 n} cn = cnlog_2 n = O(n.\, log_2 n)$$

$\therefore T(n) = O(n.\, log_2 n)$

2.3. Which one is better? Insertion sort or Merge sort?

As mentioned earlier, the basic version of the merge sort typically divides the array into subproblems till the base case i.e., till the problem size reduces to 1 and then merges them together in such a manner that the merged array is sorted and thus the name ‘merge sort’. Whereas the insertion sort iterates through every item in the array from the left size (or right size) and finds the position where the element needs to be inserted and thus the name ‘insertion sort’.

The time complexity of merge sort is $O(nlog(n))$ and that of insertion sort is $O(n^2)$. That said, we can say that clearly insertion sort is very bad and should not be used. Is that true?

The Big-Oh notation shows how time required by an algorithm grows with increasing size of the input and not how fast the algorithm is. The Big-Oh notation ignores the values in the running time expression with lower powers. These lower power expressions have a significant value when the value of n is small.

For insertion sort these lower power expressions are very small for small values of n whereas for the merge sort they are very large. Thus, the insertion sort works best or requires a small amount of time to sort arrays with few elements as compared to merge sort.

In all, the above discussion can be summarized as follows,

Size of n	Insertion or Merge sort is better?
$n \leq 32$	Insertion sort is better or requires less amount of time
$n > 32$	Merge sort is better or requires less amount of time

This gives rise to usage of hybrid sorting algorithms which would work better for smaller and bigger input sizes.

2.4. Merge-insertion hybrid sorting algorithm:

As the name suggests, this algorithm is a hybrid sorting algorithm that uses merge and insertion sort. The pseudocode for the algorithm can be given as follows:

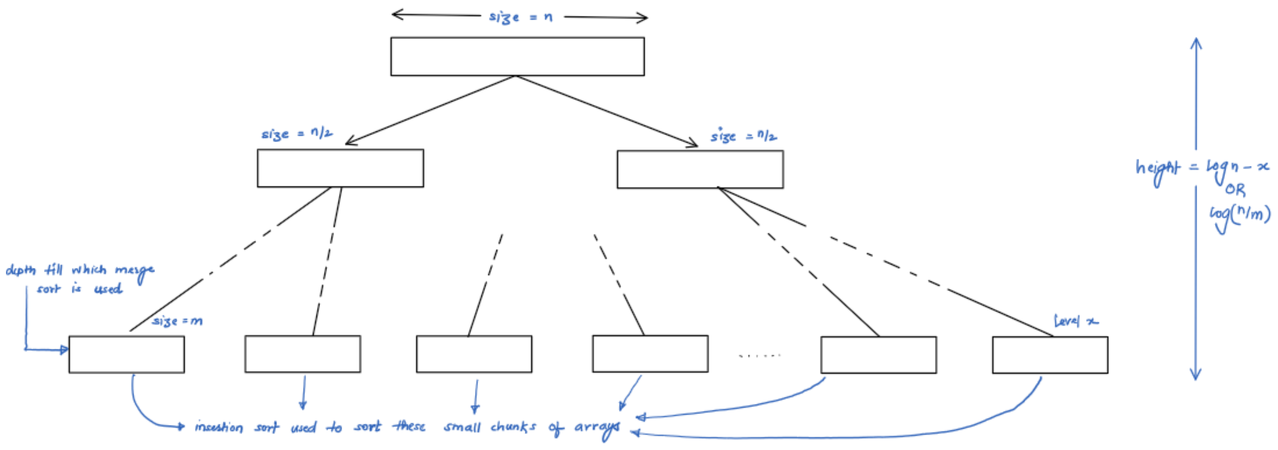
2.4.1. Pseudocode for merge-insertion hybrid sort:

```
Merge_insertion_sort(A,p,r):
    if p < r:
        if (r - p + 1) ≤ 32:
            Insertion_sort(A, r - p + 1)
        else:
            Merge_sort(A, p, r)
```

2.4.2. Running time analysis:

We know that the merge sort splits the given input into two parts at every level till the split/ array size reduces to 1. But here in merge-insertion sort we stop this tree of splits to stop as soon as the height reaches x or as the array size reaches m. After that we use insertion sort on array size of m.

Below is the small illustration of how the merge-sort tree expands till the level x and then insertion sort is used.



size = n
size = n/2
size = n/2
height = log n − x
OR
log(n/m)
depth till which merge sort is used
size = m
......
level x
insertion sort used to sort these small chunks of arrays

(References: Gate overflow https://gateoverflow.in/99455/sorting-algorithm.)

2.4.3. Running time of merge sort function:

The running time of merge sort depends on the height of the tree above and size of the array i.e., n. Thus, the running time of merge sort can be given as,

$$\begin{aligned}T_{merge}(n) &= n * (height\ of\ tree)\\ &= n * (log(n) - x)\\ &= n * (log(n) - log(m))\\ \therefore T_{merge}(n) &= O(n * log(n/m))\end{aligned}$$

2.4.4. Running time of insertion sort function:

The insertion sort will be applied to the arrays of size m and there are a total of n/m number of such arrays. We know that the running time of insertion sort for array size of n is $O(n^2)$ i.e. $Insertion - sort(n) = O(n^2)$ in the worst case and $Insertion - sort(n) = O(n)$ in the best case.

Thus, the running time of insertion sort function can be given as,

$$\begin{aligned}T_{insert}(n) &= (n/m) * (Insertion - sort(m))\\ &= O(n/m * m^2)\end{aligned}$$

$\therefore T_{insert}(n) \quad = O(nm)$

2.4.5. Total running time of merge-insertion sort algorithm:

Thus, the total running time of merge-insertion sort can be given as,

$$\begin{aligned} T_{merge-ins}(n) &= T_{merge}(n) + T_{insert}(n) \\ &= O(n * log(n/m)) + (n/m) * (Insertion - sort(m)) \end{aligned}$$

$\therefore T_{merge-ins}(n) \quad = O(n * log(n/m) + (n/m) * (Insertion - sort(m)))$

Or

$\therefore T_{merge-ins}(n) \quad = O(n * log(n/m) + nm)$

2.4.6. Best case, worst case and average case analysis:

Best case scenario:

The best-case scenario is when all the n/m small arrays are sorted in ascending order i.e. the $Insertion - sort(m) = O(m)$. Thus, the best-case running time can be given as,

$$T_{merge-ins}(n) = O(n * log(n) + n)$$

Worst case scenario:

The worst-case scenario is when all the n/m small arrays are sorted in descending order i.e. the $Insertion - sort(m) = O(m^2)$. Thus, the worst-case running time can be given as,

$$T_{merge-ins}(n) = O(n * log(n) + nm)$$

Average case scenario:

The average case scenario is when half of the n/m small arrays are sorted in ascending and the other half in descending order. Thus, the average case running time can be given as,

$$T_{merge-ins}(n) = O(n * log(n) + \frac{nm+n}{2}) = O(n * log(n) + nm)$$

Therefore,

$$T_{merge-ins}(n) = O(n * log(n) + nm)$$

Important note:

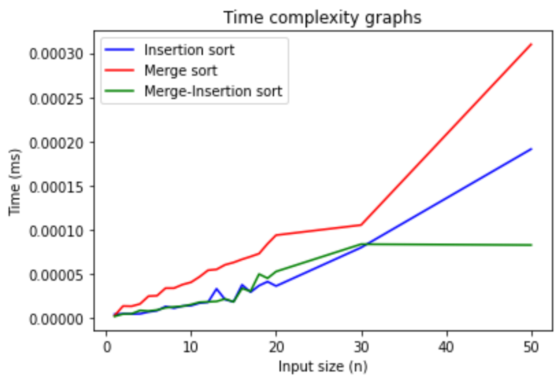
In all the above time complexity equations, we have kept the lower order term as well because for small values of n i.e., $n \leq 32$ the time complexity depends on these lower order terms as well.

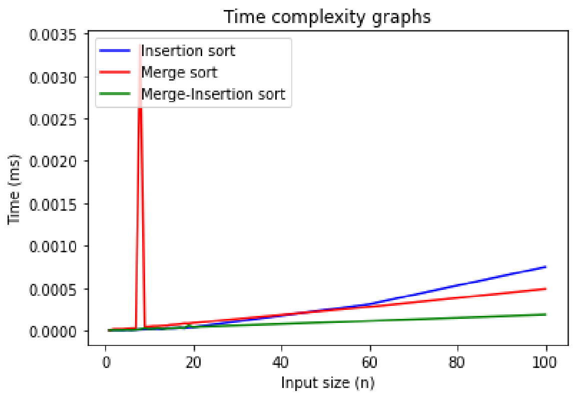
3.0. Code:

```
import random, math, time, matplotlib.pyplot as plt

# Timer objectto record time
class time_recorder:
  # initialize timer objects
  def __init__(self):
    self.time_arr = {}     # dictionary to store object
    self.start = 0         # start time recorder variable
    self.end = 0           # end time recorder variable

  # Start timer function
  def start_timer(self):
    self.start = time.time()

  # Function to stop timer and store the recorded time in dictionary
object
  def end_timer(self, iter_size):
    self.end = time.time()
    self.time_arr[iter_size] = self.end - self.start

# Insertion sort function
def insertion_sort(A, n):
  for i in range(1, n):
    key = A[i]
    k = i - 1
    while k >= 0 and A[k] > key:
      A[k+1] = A[k]
      k -= 1
    A[k+1] = key

# Merge sort function to recursively call merge sort and merge function
for
# sorting the given array A
def merge_sort(A, p, r):
  if p < r:
    q = math.floor((p+r)/2)
    merge_sort(A, p, q)
    merge_sort(A, q+1, r)
    merge(A, p, q, r)

# Merge function to merge first and second half sorted arrays of given
array A
def merge(A, p, q, r):
  L = A[p: q+1]
  R = A[q+1: r+1]
  L.append(math.inf)
  R.append(math.inf)
  i = 0; j = 0
  for k in range(p, r+1):
    if L[i] <= R[j]:
      A[k] = L[i]
      i += 1
    else:
      A[k] = R[j]
      j += 1

# Merge sort function to recursively call merge sort and merge function
for
# sorting the given array A using insertion sort
def merge_ins_sort(A, p, r):
  if p < r:
    if (r-p+1) <= 32:
      insertion_sort(A, r-p+1)
    else:
      q = math.floor((p+r)/2)
      merge_ins_sort(A, p, q)
      merge_ins_sort(A, q+1, r)
      merge_ins(A, p, q, r)


# Merge function to merge first and second half sorted arrays of given
array A
def merge_ins(A, p, q, r):
  L = A[p: q+1]
  R = A[q+1: r+1]
  L.append(math.inf)
  R.append(math.inf)
  i = 0; j = 0
  for k in range(p, r+1):
    if L[i] <= R[j]:
      A[k] = L[i]
      i += 1
    else:
      A[k] = R[j]
      j += 1


def main1():
  timerA = time_recorder()
  timerB = time_recorder()
  timerC = time_recorder()

  for j in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 60, 100]:
    arr = []
    for i in range(j):
      arr.append(random.randint(0,20))
    print(f"\nArray size = {j}")
    if j <= 1000:
      print("Random generated array =")
      print(arr)

    arr1 = arr.copy()
    arr2 = arr.copy()
    arr3 = arr.copy()

    if j < 20000:
      print("Using insertion sort:")

      timerA.start_timer()
      insertion_sort(arr1, len(arr1))
      timerA.end_timer(j)
      if j <= 1000:
        print("Sorted array:", arr1)

    print("Using merge sort:")

    timerB.start_timer()
    merge_sort(arr2, 0, len(arr2)-1)
    timerB.end_timer(j)
    if j <= 1000:
      print("Sorted array:", arr2)

    print("Using merge - insertion sort:")
    timerC.start_timer()
    merge_ins_sort(arr3, 0, len(arr3)-1)
    timerC.end_timer(j)
    if j <= 1000:
      print("Sorted array:", arr3)

  print(timerA.time_arr.keys())
  print(timerA.time_arr.values())
  plt.plot(list(timerA.time_arr.keys()), list(timerA.time_arr.values()),
"-b", label = "Insertion sort")

  print(timerB.time_arr.keys())
  print(timerB.time_arr.values())
  plt.plot(list(timerB.time_arr.keys()), list(timerB.time_arr.values()),
"-r", label = "Merge sort")

  print(timerC.time_arr.keys())
  print(timerC.time_arr.values())
  plt.plot(list(timerC.time_arr.keys()), list(timerC.time_arr.values()),
"-g", label = "Merge-Insertion sort")
  plt.legend(loc="upper left")
  plt.title("Time complexity graphs")
  plt.xlabel("Input size (n)")
  plt.ylabel("Time (ms)")

  plt.show()

def main2():
  timerA = time_recorder()
  timerB = time_recorder()
  timerC = time_recorder()

  for j in [1, 100, 1000, 10000, 100000]:
    arr = []
    for i in range(j):
      arr.append(random.randint(0,20))
    print(f"\nArray size = {j}")
    if j <= 1000:
      print("Random generated array =")
      print(arr)

    arr1 = arr.copy()
    arr2 = arr.copy()
    arr3 = arr.copy()

    if j < 20000:
      print("Using insertion sort:")

      timerA.start_timer()
      insertion_sort(arr1, len(arr1))
      timerA.end_timer(j)
      if j <= 1000:
        print("Sorted array:", arr1)

    print("Using merge sort:")

    timerB.start_timer()
    merge_sort(arr2, 0, len(arr2)-1)
    timerB.end_timer(j)
    if j <= 1000:
      print("Sorted array:", arr2)

    print("Using merge - insertion sort:")
    timerC.start_timer()
    merge_ins_sort(arr3, 0, len(arr3)-1)
    timerC.end_timer(j)
    if j <= 1000:
      print("Sorted array:", arr3)

  print(timerA.time_arr.keys())
  print(timerA.time_arr.values())
  plt.plot(list(timerA.time_arr.keys()), list(timerA.time_arr.values()),
"-b", label = "Insertion sort")
```

```
  print(timerB.time_arr.keys())
  print(timerB.time_arr.values())
  plt.plot(list(timerB.time_arr.keys()), list(timerB.time_arr.values()),
"-r", label = "Merge sort")

  print(timerC.time_arr.keys())
  print(timerC.time_arr.values())
  plt.plot(list(timerC.time_arr.keys()), list(timerC.time_arr.values()),
"-g", label = "Merge-Insertion sort")
  plt.legend(loc="upper left")
  plt.title("Time complexity graphs")
  plt.xlabel("Input size (n)")
  plt.ylabel("Time (ms)")

  plt.show()

main1()
main2()
```

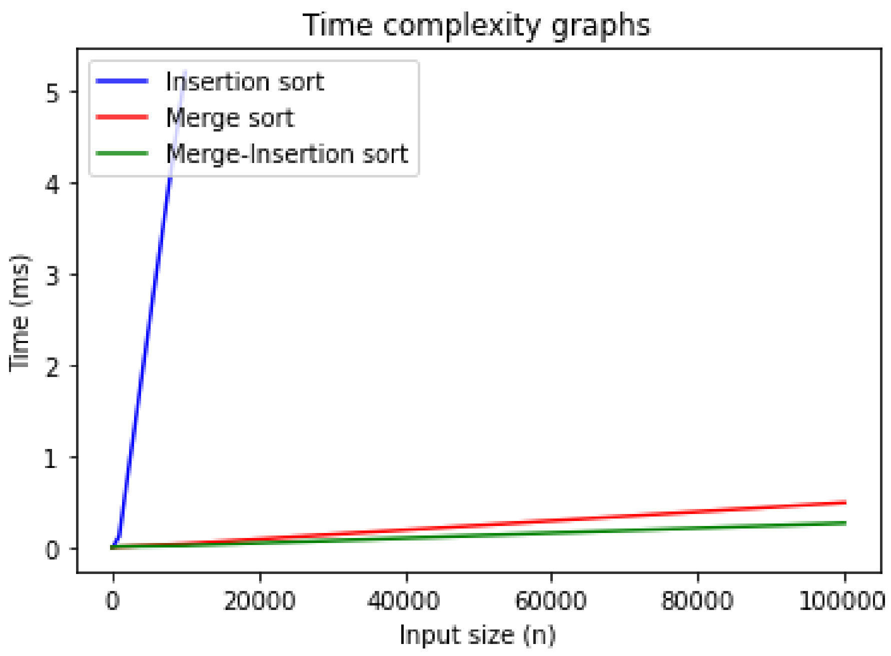
4.0. Output time analysis graph:





Plots for smaller values of n

Above are the two plots for two different series of random inputs for different sizes of input array. In these graph plots, we can see that the time required by merge sort algorithm peaks or is greater for smaller values of n whereas that required by the insertion sort is comparatively low. The time required by the merge-insertion sort is almost equal to that required by insertion sort for lower values of n.



Plot for larger values of n

The above plot is the plot of time required by all 3 algorithms for larger values of n. We can see that the time required by the insertion sort is the maximum whereas that required by the merge sort is extremely less. The merge-insertion sort requires time less than the merge sort as well.

5.0. Conclusion:

Thus, from the above output graphs we can say that the merge-insertion sort works well for both smaller size of input arrays as well as larger sizes unlike merge sort which requires more time for sorting small arrays and insertion sort which requires more time for sorting large arrays.

6.0. Reference:

- Introduction to Algorithms Third edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/insertion-sort/
- https://www.geeksforgeeks.org/sorting-by-combining-insertion-sort-and-merge-sort-algorithms/
- Lecture slides of Algorithms and Data structures by Prof. Stephen Alexandra, The University of Texas at Arlington.