CHARLOTTE

Comparison based Sorting Algorithms

PROJECT REPORT

Devashri Gadgil - 801243925

Table of contents

Problem Statement

For any given input sequence of size n, implement comparison-based sorting algorithms that will sort elements of an input sequence in ascending order. Analyze their performance against various input sizes. Document their performance for 3 cases :

1. The input sequence of size n that contains randomly generated integers
2. The input sequence of size n that is already sorted
3. The input sequence of size n that is reversely sorted

There are a few prerequisites followed to enhance the accuracy of the results:

1. All sorts are given exact same input array of size n
2. A sort is performed thrice and time taken for execution is noted for each run
3. An average time taken by each sort is calculated based on data gathered in the previous step.
4. A graph is plotted that indicates input size n on the x-axis and the average time taken(in seconds) on the y-axis.

Sorting mechanisms that are considered for this exercise are listed below.

1. Insertion sort
2. Merge sort
3. Vector-based heap sort
4. In place quick sort
5. Modified quick sort

Detailed analysis for each of these algorithms will be presented in the next sub-sections. I have chosen Python as a programming language to implement mentioned sorting techniques.

Insertion sort

Concept:

Insertion sort is one of the most simple sorting algorithms invented so far. Imagine a bunch of cards placed on a table facing downwards. You pick up the first card from that bunch and place it on your left hand. Since it was the first card you ever picked up, it is assumed that the current position of that card is correct. For each cad that you select later, we compare it with all previous cards. If any of the

previous cards have a value greater than the currently selected card we simply move the previous card to the right by one position. This way we make room for the currently selected cards. After placing the currently selected card in the left hand, we can conclude that all cards in the left hand are sorted in ascending order. At the same time, the remaining cards on the table are still unsorted. The process is continued until there is no card left on the table. The same concept is applied for developing the insertion sort technique.

Data structures used:

1. An input sequence is stored in a python list. A python list is analogous to an array structure that is used in almost all high-level programming languages.
2. Sorting is performed on the same python list. This mechanism is thus called the in-place sorting technique. The sorted python list is returned as output.

Time complexity:

Execution time taken by insertion sort is directly proportional to the input size n.

1. Best case:
   When the input sequence is already sorted, time is spent during comparison. No element is moved to the right. Using Big-Oh notation, the time complexity for insertion sort, in this case, can be expressed as **O(n)**.
2. Worst case:
   When the input array is reversely sorted, for every element comparison the program has to shift all previous elements to right by one place. Using Big-Oh notation, the time complexity for insertion sort, in this case, can be expressed as $\mathbf{O(n^2)}$.

Results:

The algorithm is tested with total ten input sizes: 1000, 2000, 3000, 5000, 10000, 20000, 30000, 40000, 50000, 60000. The results are summarized in the below table. The execution time is in seconds.

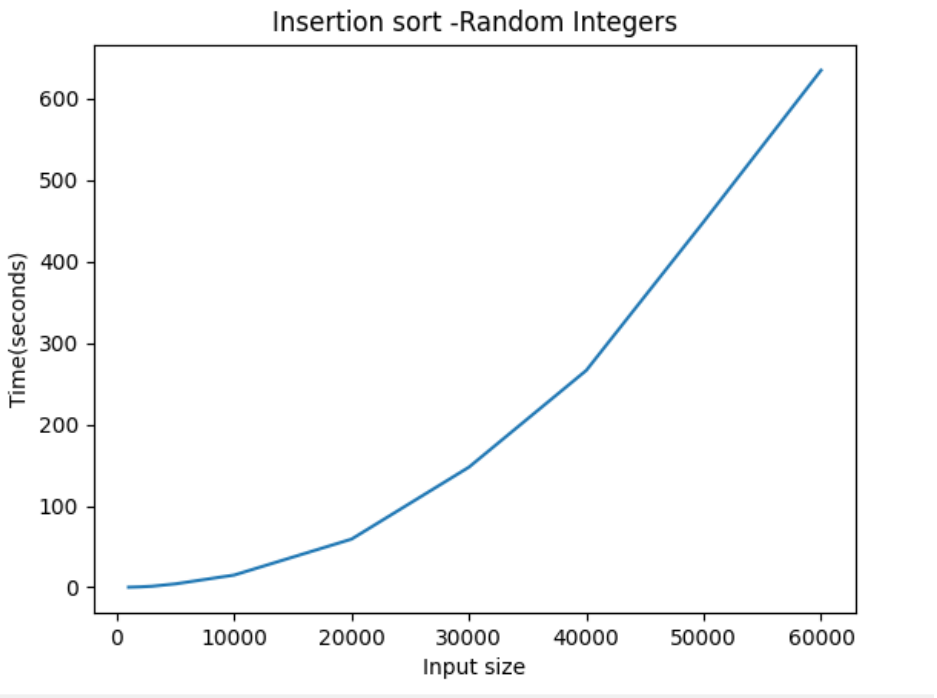
Input Type	1000	2000	3000	5000	10000
Random integers	0.16	0.60	1.36	4.27	15.07

Sorted sequence	0.00	0.00	0.00	0.01	0.01
Reverse sorted sequence	0.27	1.12	2.75	7.14	35.51

Input Type	20000	30000	40000	50000	60000
Random integers	59.18	147.45	266.17	448.47	634.38
Sorted sequence	0.02	0.02	0.03	0.05	0.05
Reverse sorted sequence	124.11	303.14	615.46	951.24	1441.78

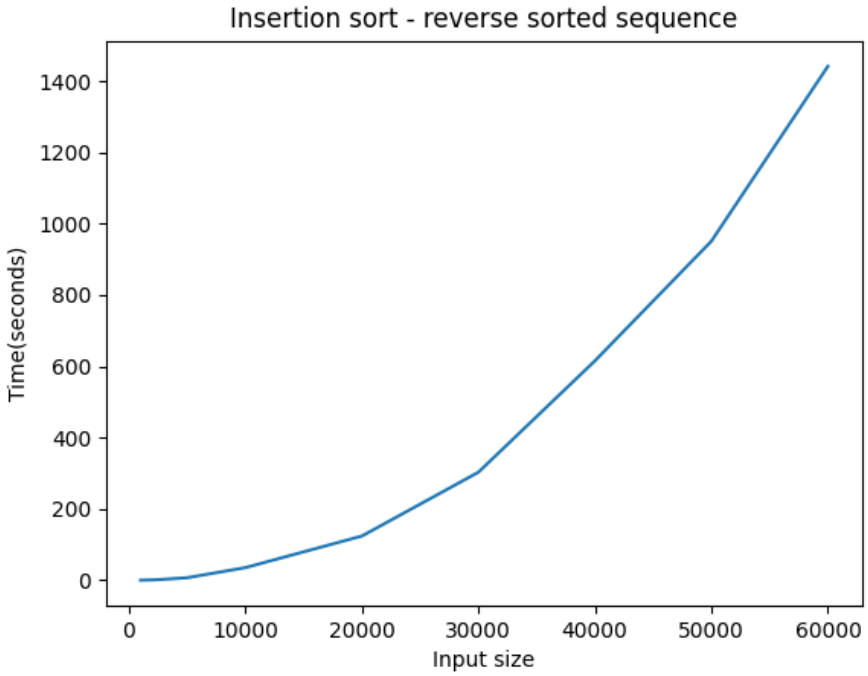
Graphs:

1. Insertion Sort: Random Integers



2. Insertion Sort: Sorted sequence

3. Insertion sort-Reverse sorted sequence



Code:

```
def sort(self):
        # print("Input array : " + str(self.inputArray))
        for i in range(1,len(self.inputArray)):
            key = self.inputArray[i]
            j = i -1
            while(j >= 0 and self.inputArray[j] > key):
                self.inputArray[j+1] = self.inputArray[j]
                j -= 1
            self.inputArray[j+1] = key
        return self.inputArray
```

Merge sort

Concept:

Merge sort applies the divide and conquer principle for obtaining the results. The strategy is to divide the main problem into smaller subproblems. Each subproblem is then solved individually. The solutions for all subproblems are combined together to solve the main problem. For sorting an input sequence of size n, the first step is to partition input into subsequences by calculating the middle index. Each partition is divided further until there remains only one element per partition. Since there is only one element per partition, it is already sorted. Now we call merge function that merges all sub partitions to create sorted array from input sequence. For merging two subsequences, we compare first element from both subsequences. The smaller element out of compared elements is appended in the sorted python list. The pointer on the subsequence to which the smaller element belonged is incremented. This process is repeated until all elements from both subsequences are compared and appended to sorted sequence.

Data structure used:

1. A python list is used for providing input sequence of size n
2. A similar python list is used to store sorted sequence.This is the reason why it is not an in-place sorting technique
3. Sorted python list is however treated as queue. Elements are inserted at rear end. This is implemented using append method.

Time complexity:

1. Best case :
   At every steps elements are divided into halves which takes log n time. The amount of work done at depth i is O(n). Thus total time taken for executing merge sort can be expressed as **O(n log n)**.
2. Worst case:
   The worst-case occurs when partitions ready for merge have alternate elements. In this case, each element has to be compared once. However time complexity still remains to be **O(n log n)** for input size n.

Results:

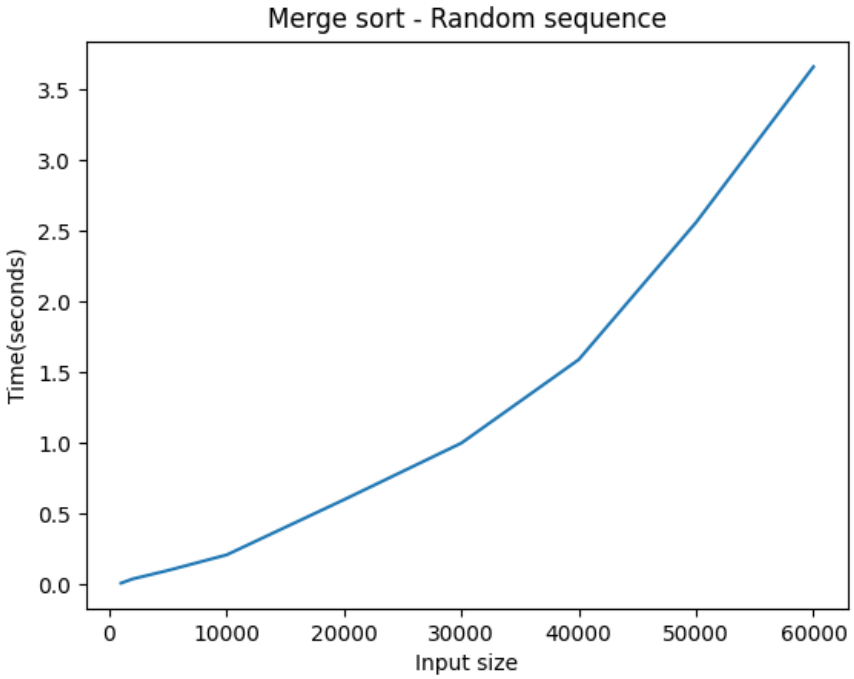
The algorithm is tested with total ten input sizes: 1000, 2000, 3000, 5000, 10000, 20000, 30000, 40000, 50000, 60000. The results are summarized in the below table.

Input Type	1000	2000	3000	5000	10000
Random integers	0.01	0.04	0.06	0.10	0.21
Sorted sequence	0.01	0.03	0.06	0.14	0.19
Reverse sorted sequence	0.01	0.03	0.05	0.08	0.27

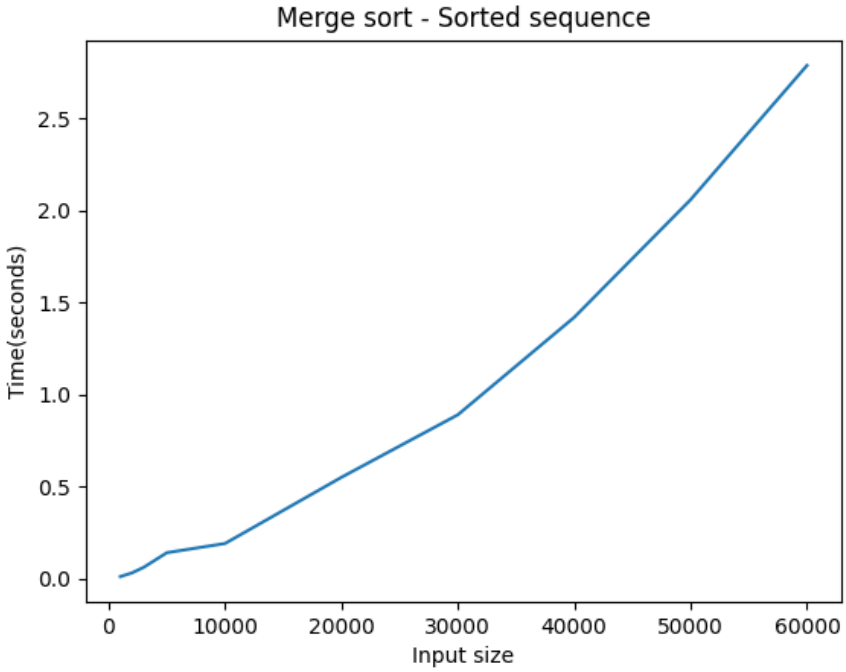
Input Type	20000	30000	40000	50000	60000
Random integers	0.60	1.00	1.59	2.56	3.66
Sorted sequence	0.55	0.89	1.42	2.06	2.79
Reverse sorted sequence	0.60	0.91	1.61	2.49	3.15

Graphs:

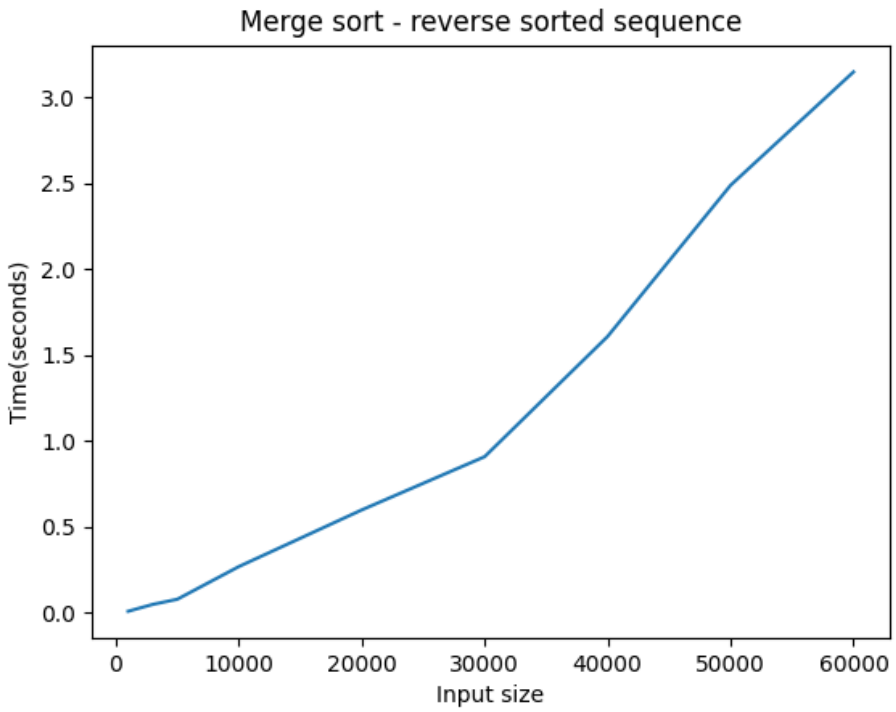
1. Merge sort - Random Integers



2. Merge sort - sorted sequence



3. Merge sort - reverse sorted sequence



Code:

```
def mergeSort(self):
    self.sortedQueue = self.partitionSequence(self.inputArray)

def partitionSequence(self,arrayForPartition):
    print("Array for partition", arrayForPartition)
    if len(arrayForPartition) > 1:
        half = len(arrayForPartition) // 2
        left = arrayForPartition[:half]
        right = arrayForPartition[half:]
        left = self.partitionSequence(left)
        right = self.partitionSequence(right)
        arrayForPartition = self.mergeSequence(left,right)
    return arrayForPartition


def mergeSequence(self,queueOne,queueTwo):
    print("queue 1" + str(queueOne))
```

```
    print("queue 2 " + str(queueTwo))
    sortedQueue = []
    while len(queueOne) > 0 and len(queueTwo) > 0:
        if queueOne[0] < queueTwo[0]:
            sortedQueue.append(queueOne.pop(0))
        else:
            sortedQueue.append(queueTwo.pop(0))
    while len(queueOne) > 0:
        sortedQueue.append(queueOne.pop(0))
    while len(queueTwo) > 0:
        sortedQueue.append(queueTwo.pop(0))
    print(sortedQueue)
    return sortedQueue
```

Heap Sort

Concept:

Heap is a binary tree in which the key of a child is always greater than or equal to the key of its parent. This variation is called min-heap. There is another variation of heap which is a max heap in which there is the exact opposite condition. However, in our case, we are going to use a min-heap for sorting an input sequence of size n. There are two methods required for heap sort:

1. Inserting a new element - For a new element to be inserted in an existing heap, we find a vacant spot for the new element. Once the element is inserted, the key of the newly inserted element is compared with its parent and then ancestors to satisfy the heap property. This process is called up-heap.
2. Removing minimum element - The smallest element from the heap is always stored at the root node. Hence, we remove an element from the root node. We then perform a down heap operation to satisfy the heap property. In this way, the second smallest element is brought up to the root.

In our implementation, we are using a vector data structure to implement the heap sort.

Data structure used:

1. This is an in-place sorting technique that uses vectors to store input and ultimately sorted sequence.
2. Vector is implemented using python list. For every node i, the left child is stored at index 2i in the heap while a right child is stored at index 2i+1. This is the reason why the cell at index 0 is kept empty.

Time complexity:

1. Best case :
   Heapsort is faster for larger inputs. The best-case time complexity is **O(n log n)** for the input sequence of size n.
2. Worst case:
   In the worst-case scenario also heap sort offers **O(n log n)** time complexity for input sequence of size n.

Results:

The algorithm is tested with total ten input sizes: 1000, 2000, 3000, 5000, 10000, 20000, 30000, 40000, 50000, 60000. The results are summarized in the below table.

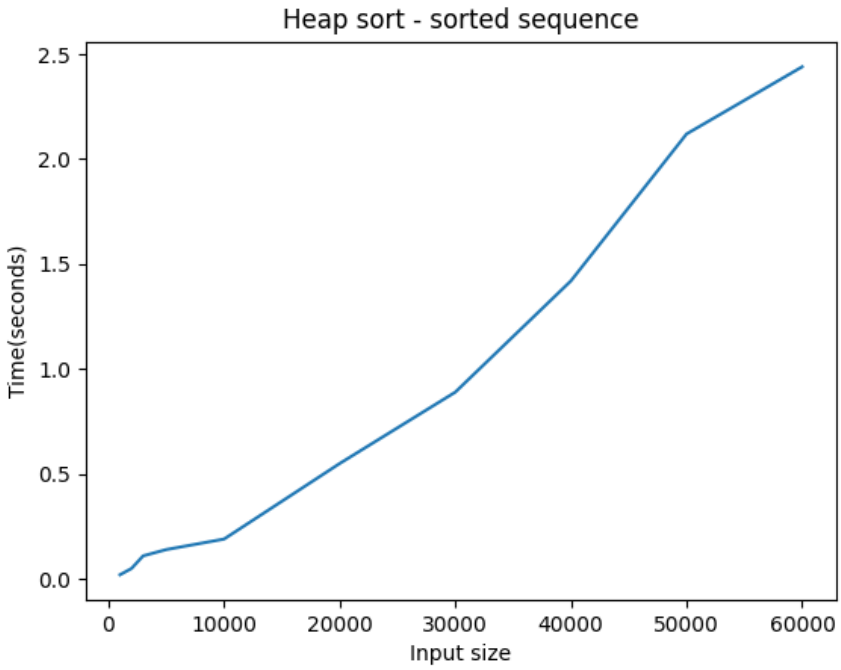
Input Type	1000	2000	3000	5000	10000
Random integers	0.02	0.05	0.11	0.15	0.31
Sorted sequence	0.02	0.05	0.11	0.14	0.19
Reverse sorted sequence	0.03	0.08	0.12	0.20	0.46

Input Type	20000	30000	40000	50000	60000
Random integers	0.70	1.25	1.62	2.35	2.35
Sorted sequence	0.55	0.89	1.42	2.12	2.44
Reverse sorted sequence	1.06	1.47	2.52	2.91	4.03

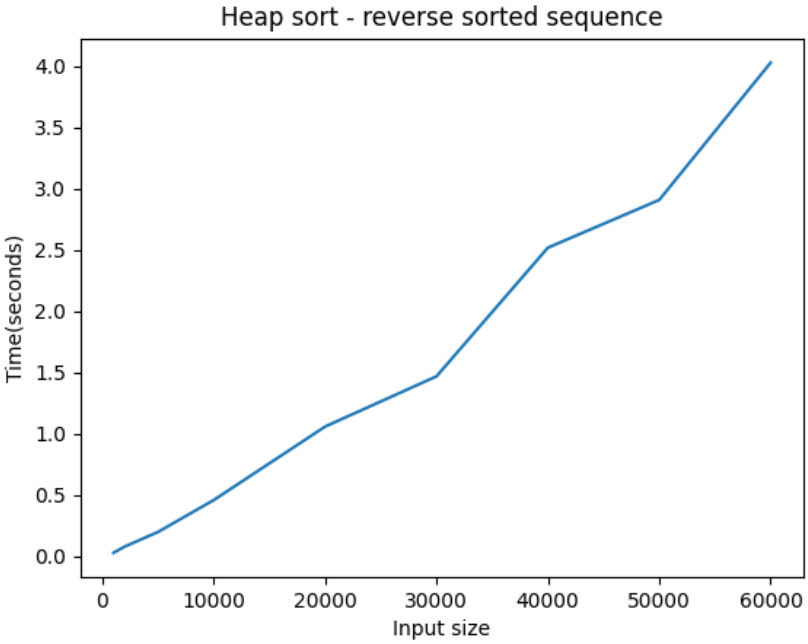
Graphs:

1. Heap sort - random sequence

2. Heap sort - sorted sequence



3. Heap sort - reverse sorted sequence



Code:

```
def heapSort(self):
        for i in range(len(self.inputSequence)):
            self.insertElm(self.inputSequence[i])
        for j in range(len(self.vectorHeap) - 1):
            minElm = self.removeMin()
            self.sortedSequence.append(minElm)
        return self.sortedSequence

    def insertElm(self,item):
        self.insertionIndex = self.insertionIndex + 1
        self.vectorHeap[self.insertionIndex] = item
        # Upheap to satisfy heap property
        counter = self.insertionIndex
        while(counter > 1 and self.vectorHeap[counter//2] > self.vectorHeap[counter]):
            self.vectorHeap[counter//2],self.vectorHeap[counter] =
self.vectorHeap[counter],self.vectorHeap[counter//2]
            counter = counter//2

    def removeMin(self):
```

```
        minElm = self.vectorHeap[1]
        self.vectorHeap[1] = self.vectorHeap[self.insertionIndex]
        self.insertionIndex = self.insertionIndex -1
        counter = 1
        while counter < self.insertionIndex:
            if((2 * counter) + 1) <= self.insertionIndex:
                if(self.vectorHeap[counter] <= self.vectorHeap[2 * counter] and
self.vectorHeap[counter] <= self.vectorHeap[(2 * counter) + 1]):
                    # no need for downheap;
                    return minElm
                else:
                    if self.vectorHeap[2 * counter] < self.vectorHeap[(2 * counter) + 1]:
                        j = 2 * counter
                        self.vectorHeap[j],self.vectorHeap[counter] =
self.vectorHeap[counter],self.vectorHeap[j]
                        counter = j
                    else:
                        j = (2 * counter) + 1
                        self.vectorHeap[j],self.vectorHeap[counter] =
self.vectorHeap[counter],self.vectorHeap[j]
                        counter = j
            else:
                if(2 * counter) <= self.insertionIndex:
                    if self.vectorHeap[counter] > self.vectorHeap[2 * counter]:
                        self.vectorHeap[2 * counter],self.vectorHeap[counter] =
self.vectorHeap[counter],self.vectorHeap[2 * counter]

                return minElm
```

In-place quick sort

Concept:

Quicksort is one of the fastest algorithms and is used in many real-world applications. It adapts to the ‘divide and conquer’ strategy. However, unlike merge sort, quick sort is responsible for sorting the input elements during the partition

stage itself. For comparison, we randomly select an element from the input array. We call this element a pivot. The next step is to remove each element from the input sequence. All elements less than a pivot would be appended to one subsequence while all elements greater than or equal to the pivot go to other subsequence. This partitioning is a recursive process and is carried out for all subsequences generated from the previous step until there remains only one element per subsequence. The subsequences are coupled together to generate the final output.

The advantage of an in-place quicksort is that we do not use any extra memory. All processing is carried out on a single input sequence using i and j pointers.

Data structure used:

A python list contains an input sequence. Sorting is carried out on the same list. The sorted sequence is placed in the same list since it is an in-place sorting technique.

Time Complexity:

1. Best case :
   Quicksort is suitable for sorting large input sequences. The best case is when the pivot is chosen such that the input sequence is halved at each step. The depth of recursion would be log n At each level of recursion, all the partitions that are on that level do work in linear time which is n. Time complexity in the best case is n * log n = **O(n log n)**.
2. Worst case :
   The worst-case occurs when an input sequence is divided into two subsequences. One of which contains zero elements and the other contains the rest of the elements from the input sequence. We don't recur on the zero-length part. We can conclude that the worst-case scenario would happen when the pivot is selected as a unique minimum or maximum. Time complexity, in this case, would be **O(n^2)**.

Results:

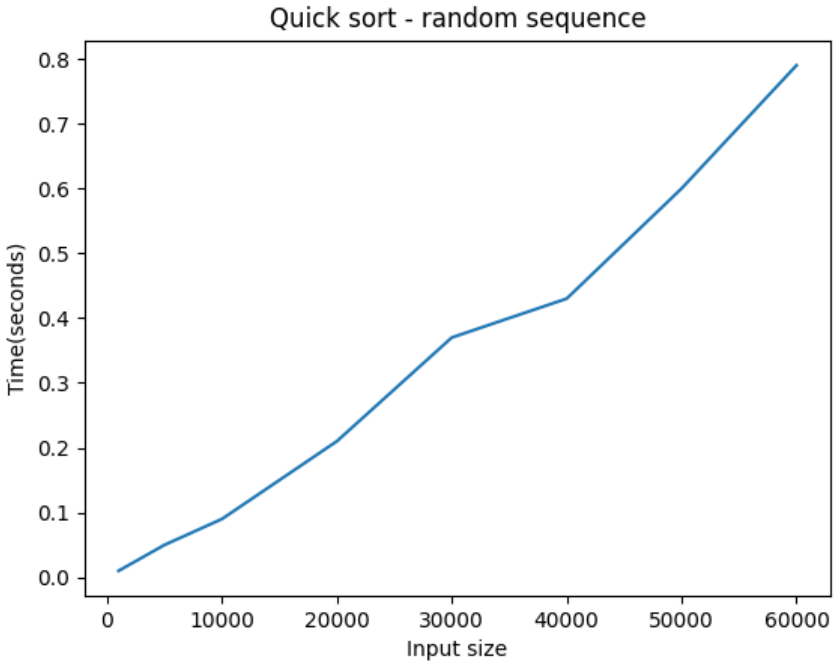
The algorithm is tested with total ten input sizes: 1000, 2000, 3000, 5000, 10000, 20000, 30000, 40000, 50000, 60000. The results are summarized in the below table.

Input Type	1000	2000	3000	5000	10000
Random integers	0.01	0.02	0.03	0.05	0.09
Sorted sequence	0.15	0.66	1.38	3.78	15.18
Reverse sorted sequence	0.16	0.64	1.38	3.63	15.22

Input Type	20000	30000	40000	50000	60000
Random integers	0.21	0.37	0.43	0.60	0.79
Sorted sequence	60.84	Recursion level exceeded maximum depth			
Reverse sorted sequence	61.44				

Graph:

1. In place quick sort - random sequence



Code:

```
def inPlaceQuickSort(self,seq,low,high):
      if low >= high:
         return
      pivot = seq[high]
      i = low
      j = high - 1
      while i <= j:
         while i <= j and seq[i] <= pivot:
            i = i + 1
         while i <= j and seq[j] >= pivot:
            j = j - 1
         if i < j:
            seq[i],seq[j] = seq[j],seq[i]

      seq[high],seq[i] = seq[i],seq[high]

      self.inPlaceQuickSort(seq,low,i-1)
      self.inPlaceQuickSort(seq,i+1,high)

      return seq
```

Modified quick sort

Concept:

Quicksort can be tweaked to improve its performance. Instead of randomly choosing pivot from the input element, we implement a median of three method. The first element, last element, and middle element from an input sequence are given as input to this method.

1. We place the smallest of the three selected elements at index zero in an input sequence

2. We place the largest element of the three selected elements at (length of input sequence - 1) position in the input sequence. It basically occupies the last position in the input sequence.
3. A median of three is placed at the center index of the input sequence.
4. We then select pivot as a median of three selected in the previous step
5. The pivot element is moved to the second last position in the input sequence
6. We can thus guarantee that the last two elements in the input sequence are in sorted order.

The rest of the quicksort functionality remains the same. However, there is a special restriction implemented.

1. If the total number of input elements is less than 10 then the sorting algorithm chosen is insertion sort
2. Else call is given to quicksort function.

Data structure used:

Python list is provided as a data structure for storing input elements which are then sorted.

Time complexity:

1. The best-case occurs when median is always found at middle index i an input sequence thus resulting in a balanced partition at every step. Time complexity in terms of a Big-Oh notation is **O(n log n)**.
2. The worst-case occurs when the pivot is selected as the smallest element from the input sequence and thus resulting in an unbalanced partition at every step. Time complexity in terms of a Big-Oh notation is **$O(n^2)$**.

Results:

The algorithm is tested with total ten input sizes: 1000, 2000, 3000, 5000, 10000, 20000, 30000, 40000, 50000, 60000. The results are summarized in the below table.

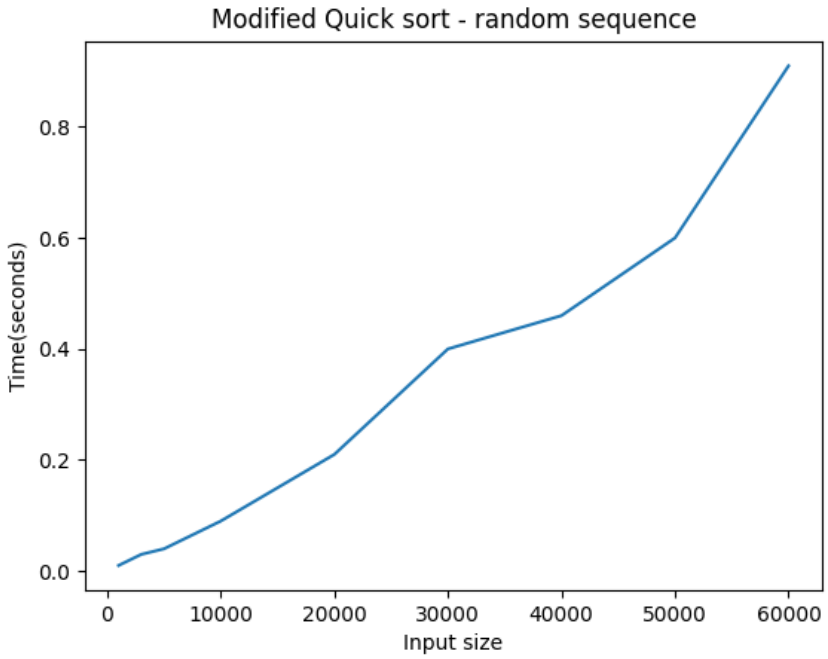
Input Type	1000	2000	3000	5000	10000
Random integers	0.01	0.02	0.03	0.04	0.09

Sorted sequence	0.00	0.01	0.02	0.05	0.05
Reverse sorted sequence	0.01	0.02	0.02	0.04	0.08

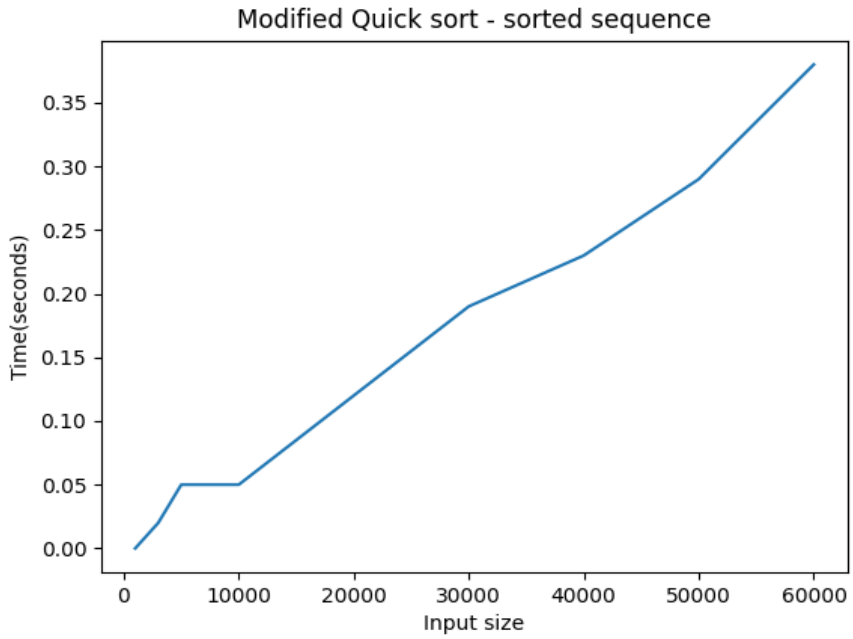
Input Type	20000	30000	40000	50000	60000
Random integers	0.21	0.40	0.46	0.60	0.91
Sorted sequence	0.12	0.19	0.23	0.29	0.38
Reverse sorted sequence	0.17	0.22	0.37	0.45	0.59

Graphs:

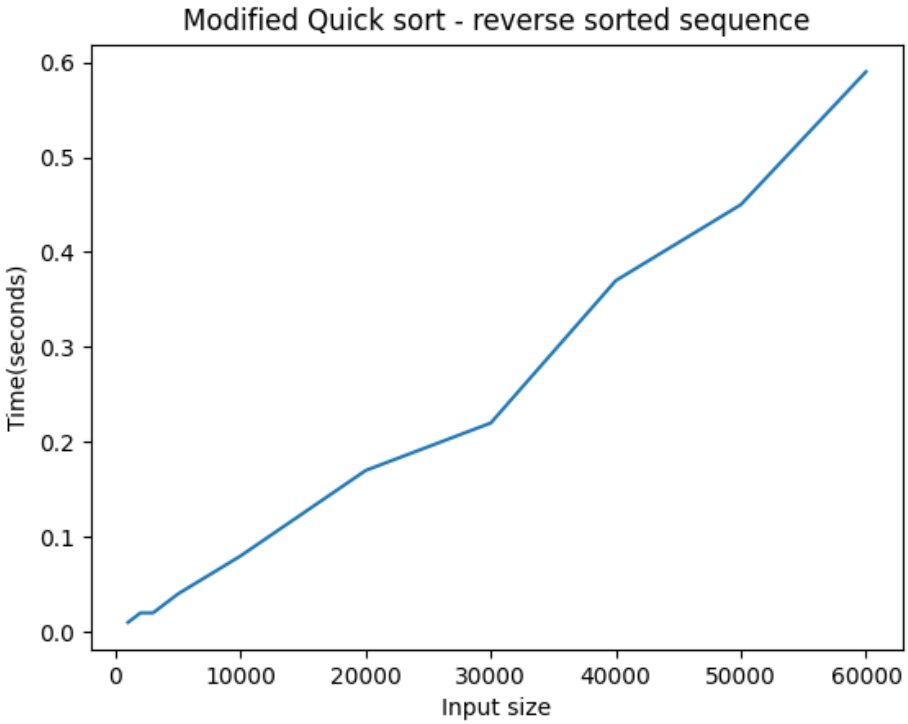
1. Modified quick sort - Random sequence



2. Modified Quick Sort - Sorted sequence



3. Modified quick sort: reverse sorted sequence



Code:

```
def quickSort(self,inputSeq,low,high):
    # decide sorting mechanism
    if low + 10 >= high:
        self.insertionSort(inputSeq,low,high)

    else:
        if low >= high:
            return

        # calculate pivot based on median of three
        middleIndex = (low+high) // 2
        arrayForMedian = [inputSeq[low],inputSeq[middleIndex],inputSeq[high]]

        inputSeq[low] = min(arrayForMedian)
        arrayForMedian.remove(min(arrayForMedian))
        inputSeq[high] = max(arrayForMedian)
        arrayForMedian.remove(max(arrayForMedian))
        inputSeq[middleIndex] = arrayForMedian[0]

        # Pivot selection
        pivot = inputSeq[middleIndex]
        # swap with elm at high -1
        temp = pivot
        inputSeq[middleIndex] = inputSeq[high - 1]
        inputSeq[high - 1] = temp
        # Inplace quick sort
        i = low
        j = high - 2
        while i <= j:
            while i <= j and inputSeq[i] <= pivot:
                i = i + 1
            while i <= j and inputSeq[j] >= pivot:
                j = j - 1
            if i < j:
                inputSeq[i],inputSeq[j] = inputSeq[j],inputSeq[i]
```

```
        inputSeq[high - 1],inputSeq[i] = inputSeq[i],inputSeq[high - 1]

        self.quickSort(inputSeq,low,i-1)
        self.quickSort(inputSeq,i+1,high)

    return inputSeq


def insertionSort(self,inputSeq,low,high):
    for i in range(low+1,high+1):
        key = inputSeq[i]
        j = i -1
        while(j >= low and inputSeq[j] > key):
            inputSeq[j+1] = inputSeq[j]
            j -= 1
        inputSeq[j+1] = key
```

Sample Output

Case one : Input size 2000 , randomly generated input sequence

```
PS F:\MS\sorts> py .\SortMain.py
Enter input size: 2000
Choose Input Type:
1.Random integers
2.Sorted Sequence
3.Reverse Sorted Sequence
1
Inside process
------Run 1-------
Running insertion sort for input size: 2000
Total Time : --- 0.60 seconds ---
Running merge sort for input size: 2000
Total Time : --- 0.04 seconds ---
Running heap sort for input size: 2000
Total Time : --- 0.07 seconds ---
Running in place quick sort for input size: 2000
```

Total Time : --- 0.02 seconds ---
Running modified quick sort for input size: 2000
Total Time : --- 0.02 seconds ---
------Run 2-------
Running insertion sort for input size: 2000
Total Time : --- 0.57 seconds ---
Running merge sort for input size: 2000
Total Time : --- 0.04 seconds ---
Running heap sort for input size: 2000
Total Time : --- 0.06 seconds ---
Running in place quick sort for input size: 2000
Total Time : --- 0.01 seconds ---
Running modified quick sort for input size: 2000
Total Time : --- 0.02 seconds ---
------Run 3-------
Running insertion sort for input size: 2000
Total Time : --- 0.57 seconds ---
Running merge sort for input size: 2000
Total Time : --- 0.03 seconds ---
Running heap sort for input size: 2000
Total Time : --- 0.06 seconds ---
Running in place quick sort for input size: 2000
Total Time : --- 0.02 seconds ---
Running modified quick sort for input size: 2000
Total Time : --- 0.02 seconds ---
Average time taken by insertion sort: 0.58 seconds
Average time taken by merge sort: 0.03 seconds
Average time taken by heap sort: 0.06 seconds
Average time taken by quick sort: 0.02 seconds
Average time taken by modified quick sort: 0.02 seconds

Case two: Input size 2000, sorted input sequence

Enter input size: 2000
Choose Input Type:
1.Random integers
2.Sorted Sequence
3.Reverse Sorted Sequence
2
Inside process
------Run 1-------
Running insertion sort for sorted input size: 2000

Total Time : --- 0.00 seconds ---
Running merge sort for sorted input size: 2000
Total Time : --- 0.04 seconds ---
Running heap sort for sorted input size: 2000
Total Time : --- 0.06 seconds ---
Running in place quick sort for sorted input size: 2000
Total Time : --- 0.67 seconds ---
Running modified quick sort for sorted input size: 2000
Total Time : --- 0.01 seconds ---
------Run 2-------
Running insertion sort for sorted input size: 2000
Total Time : --- 0.00 seconds ---
Running merge sort for sorted input size: 2000
Total Time : --- 0.04 seconds ---
Running heap sort for sorted input size: 2000
Total Time : --- 0.05 seconds ---
Running in place quick sort for sorted input size: 2000
Total Time : --- 0.63 seconds ---
Running modified quick sort for sorted input size: 2000
Total Time : --- 0.01 seconds ---
------Run 3-------
Running insertion sort for sorted input size: 2000
Total Time : --- 0.00 seconds ---
Running merge sort for sorted input size: 2000
Total Time : --- 0.03 seconds ---
Running heap sort for sorted input size: 2000
Total Time : --- 0.05 seconds ---
Running in place quick sort for sorted input size: 2000
Total Time : --- 0.62 seconds ---
Running modified quick sort for sorted input size: 2000
Total Time : --- 0.01 seconds ---
Average time taken by insertion sort: 0.00 seconds
Average time taken by merge sort: 0.04 seconds
Average time taken by heap sort: 0.05 seconds
Average time taken by quick sort: 0.64 seconds
Average time taken by modified quick sort: 0.01 seconds

Case 3: Input size 2000, reverse sorted sequence

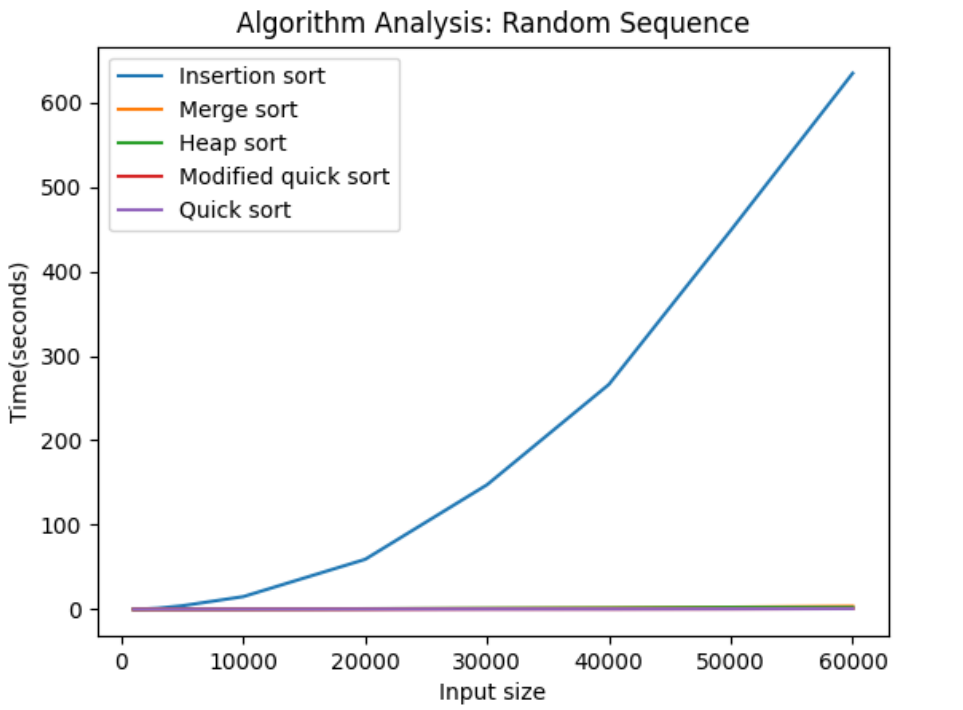
PS F:\MS\sorts> py .\SortMain.py
Enter input size: 2000
Choose Input Type:

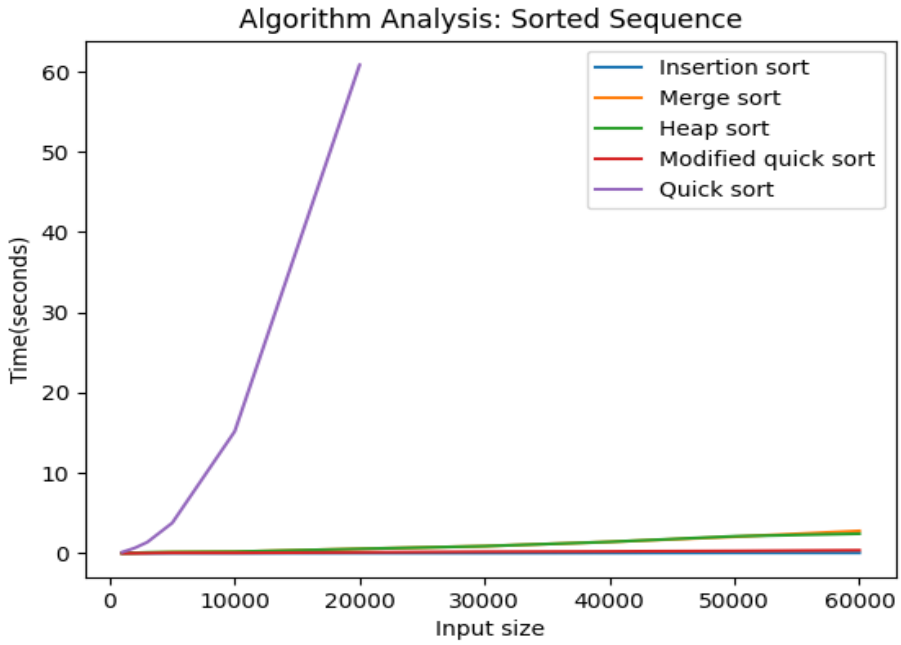
```
1.Random integers
2.Sorted Sequence
3.Reverse Sorted Sequence
3
Inside process
------Run 1-------
Running insertion sort for reverse sorted input size: 2000
Total Time : --- 1.16 seconds ---
Running merge sort for reverse sorted input size: 2000
Total Time : --- 0.03 seconds ---
Running heap sort for reverse sorted input size: 2000
Total Time : --- 0.07 seconds ---
Running in place quick sort for reverse sorted input size: 2000
Total Time : --- 0.64 seconds ---
Running modified quick sort for reverse sorted input size: 2000
Total Time : --- 0.02 seconds ---
------Run 2-------
Running insertion sort for reverse sorted input size: 2000
Total Time : --- 1.13 seconds ---
Running merge sort for reverse sorted input size: 2000
Total Time : --- 0.03 seconds ---
Running heap sort for reverse sorted input size: 2000
Total Time : --- 0.07 seconds ---
Running in place quick sort for reverse sorted input size: 2000
Total Time : --- 0.64 seconds ---
Running modified quick sort for reverse sorted input size: 2000
Total Time : --- 0.02 seconds ---
------Run 3-------
Running insertion sort for reverse sorted input size: 2000
Total Time : --- 1.13 seconds ---
Running merge sort for reverse sorted input size: 2000
Total Time : --- 0.03 seconds ---
Running heap sort for reverse sorted input size: 2000
Total Time : --- 0.08 seconds ---
Running in place quick sort for reverse sorted input size: 2000
Total Time : --- 0.64 seconds ---
Running modified quick sort for reverse sorted input size: 2000
Total Time : --- 0.02 seconds ---
Average time taken by insertion sort: 1.14 seconds
Average time taken by merge sort: 0.03 seconds
Average time taken by heap sort: 0.07 seconds
Average time taken by quick sort: 0.64 seconds
```

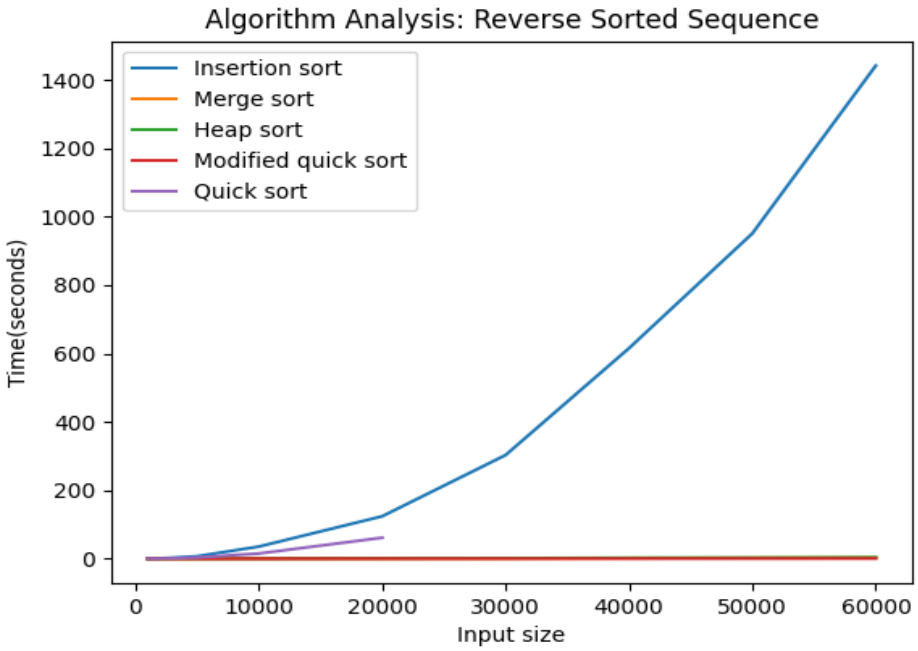
Average time taken by modified quick sort: 0.02 seconds

Performance analysis

1. Insertion sort: works well with smaller input sequences. As the size of input increases, the execution time grows exponentially. In the case of a sorted input sequence, no element is shifted to the right. Time is spent only during comparing elements. Thus insertion sort performs best when the input sequence is already sorted. In the case of the reverse sorted input sequence, for every element insertion, we have to shift all previously inserted elements to right by one position. Insertion sort thus performs terribly in the case of the reversely sorted input sequence.
2. Merge sort: It relies on partitioning elements at each step and then merging them in ascending order. It performs equally well in the case of random integers, sorted sequence, or reverse sorted sequence
3. Heap Sort: Heap sort scales well as input size n grows. It performs significantly better than in place quick sort when the input sequence is reversely sorted. Its performance is actually consistent may it be random integers or sorted sequence or reversely sorted sequence.
4. In place quick sort: Performance of in place quick sort largely depends on pivot selection. In our case, we have chosen an element at the very last index from our input array. In the case of the sorted sequence, the element at the very last index is always going to be the largest element. Thus subpartitions at each stage are always going to be one containing zero elements and the other containing all the elements from the input array. Performance of quick sort degrades when the input sequence is already sorted or reversely sorted. Due to hardware limitation and restrictive recursion stack implementation of python, we observe that the code throws RecursionError: maximum recursion depth exceeded in comparison when input size > 20000
5. Modified quick sort: It has a provision of calling insertion sort when element count is less than 10. Another advantage is we are not randomly selecting the pivot element. The median of three is applied to get a sensible pivot so that the partitions are balanced. It thus performs equally well in the case of sorted elements and reversely sorted elements.


Algorithm Analysis: Random Sequence
Insertion sort
Merge sort
Heap sort
Modified quick sort
Quick sort
Time(seconds)
0
100
200
300
400
500
600
0
10000
20000
30000
40000
50000
60000
Input size


Algorithm Analysis: Sorted Sequence
Insertion sort
Merge sort
Heap sort
Modified quick sort
Quick sort
Time(seconds)
0
10
20
30
40
50
60
0
10000
20000
30000
40000
50000
60000
Input size



Please note that -
For sorted sequence and reverse sorted sequence data is plotted till input size = 20000. As mentioned earlier, for further input sizes, python throws recursion level crossed maximum depth error due to hardware limitation on my machine. This threshold can be different for different hardware.

Inbuilt Python functions used :

1. We have used seed and randInt functions available in the random module of python to generate random integers of input size n
2. To provide sorted input as an array we are using the inbuilt sort function from python which will in turn sort the input sequence.

How to run the code:

1. Unzip the source code folder
2. Refer to the readme file and follow the instructions given carefully