Part 1

Algorithm and Data Structure

Aneesh Garg - agarg6@uncc.edu - 800815619
3/11/2014

1.0 How to run program

Prerequisite: Python 3.3 and matplotlib package

- Go to part one folder and run open python command prompt from there.
- Run Implementation.py file using following command
  `python Implementation.py MaxSize Repetition`
  Here MaxSize can be any integer such that all input sizes are less than MaxSize (default 1,00,000)
  Repetition is number of times each input size be tested and average time taken (default 10)

2.0 Input Sizes

I have tested all algorithms on following input sizes. Each algorithm has been run 10 times and average time for those 10 runs is taken.

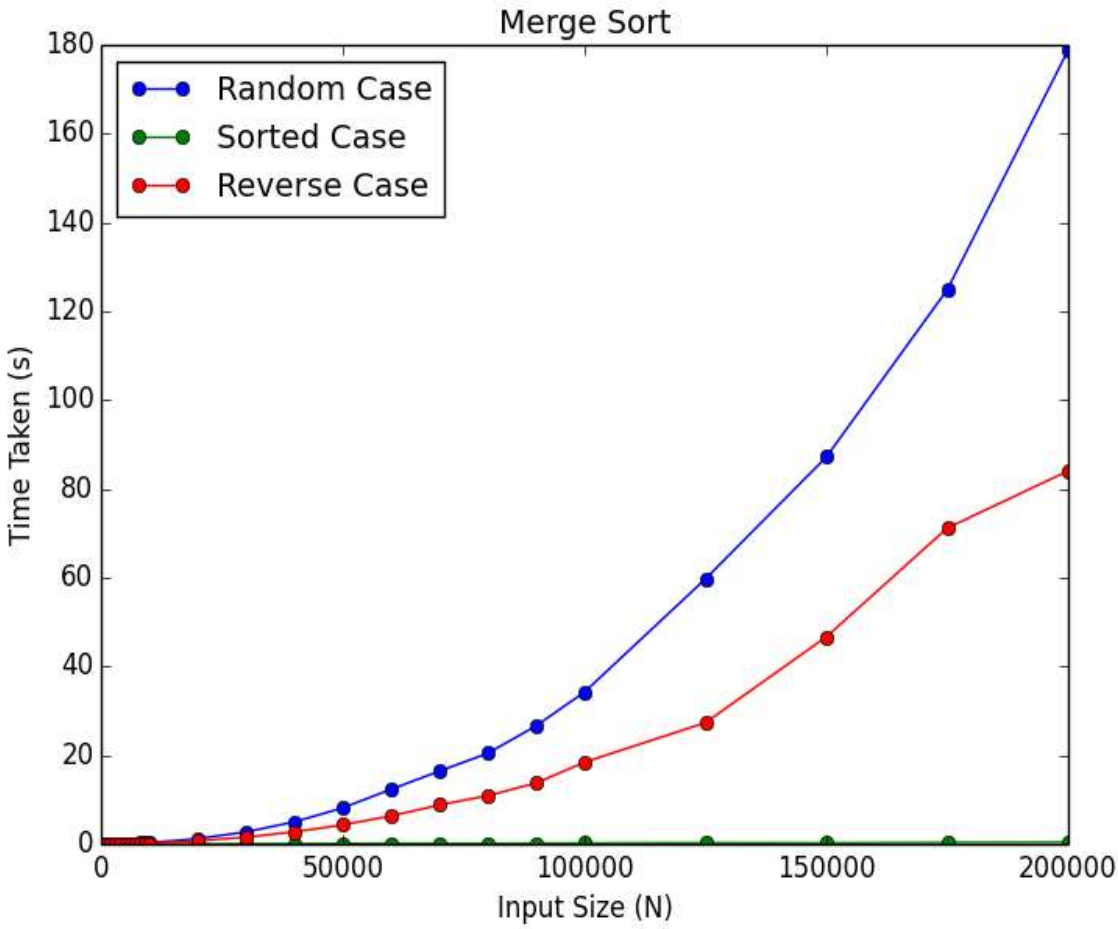
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000, 125000, 150000, 175000, 200000]

3.0 Merge Sort

3.1 Algorithm

```
11
12      def mergeSort(self, data):
13          if len(data) <= 1:
14              return data
15          else:
16              center = int(len(data)/2)
17              left, right = data[:center], data[center:]
18              left = self.mergeSort(left)
19              right = self.mergeSort(right)
20              if left[-1] <= right[0]:
21                  return left + right
22              else:
23                  return self.merge(left, right)
24
25      def merge(self, left, right):
26          result = []
27          while len(left) > 0 and len(right) > 0:
28              if left[0] <= right[0]:
29                  result.append(left[0])
30                  left = left[1:]
31              else:
32                  result.append(right[0])
33                  right = right[1:]
34          if len(left) > 0:
35              result += left
36          if len(right) > 0:
37              result += right
38          return result
```

3.2 Result



In Sorted Case Scenario run time is very less because of my improvement in the algorithm. At line number 20 I am checking if last element of left is less than first element of right. Since left , and right are both sorted arrays, therefore if the condition is true then we can directly append both hence reducing the total number of comparisons while merging.

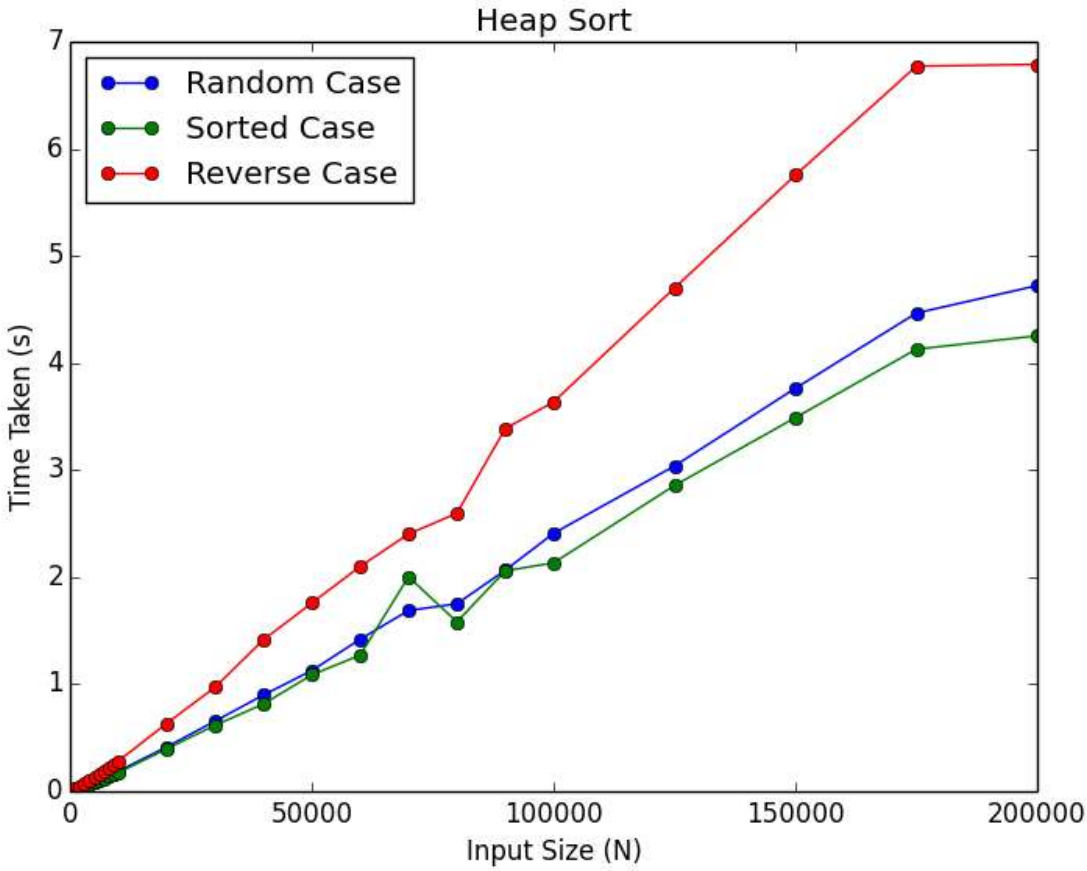
4.0 Heap Sort

4.1 Algorithm

```
    def createHeap(self, data):
        print("Creating heap of size "+ str(len(data)))
        for key in data:
            self.insertItem(key)
        print("Created heap of size "+ str(len(data)))

    def isEmpty(self):
        return self.size == 0

    def min(self):
        return self.heap[1]

    def insertItem(self, key):
        self.last += 1
        self.heap.append(key)
        self.upheap()
        self.size += 1

    def removeMin(self):
        if self.size == 0:
            return None
        else:
            minimum = self.min()
            self.heap[1] = self.heap[self.last]
            self.last -= 1
            self.size -= 1
            self.downheap()
            return minimum

    def downheap(self):
        """
        This method will restore heap order after and element is
inserted
        """
        currentIndex = 1
        while True:
            childIndex = currentIndex * 2
```

```
            if childIndex > self.size:
                break
            if (childIndex + 1) < self.size:
                #This means there are two children. Take smaller
or the left if they are eqaul
                if self.heap[childIndex] > self.heap[childIndex
+ 1]:
                    childIndex += 1
            if self.heap[currentIndex] <= self.heap[childIndex]:
                break
            #swap the two element if parent is greater than
child
            self.heap[currentIndex], self.heap[childIndex] =
self.heap[childIndex], self.heap[currentIndex]
            currentIndex = childIndex

    def upheap(self):
        """
        This method will restore heap order after and element is
removed
        """
        index = self.last
        while index > 1:
            parent = math.floor(index/2)
            #break if parent is
            if self.heap[parent] <= self.heap[index]:
                break
            self.heap[parent], self.heap[index] =
self.heap[index], self.heap[parent]
            index = parent
```

4.2 Result

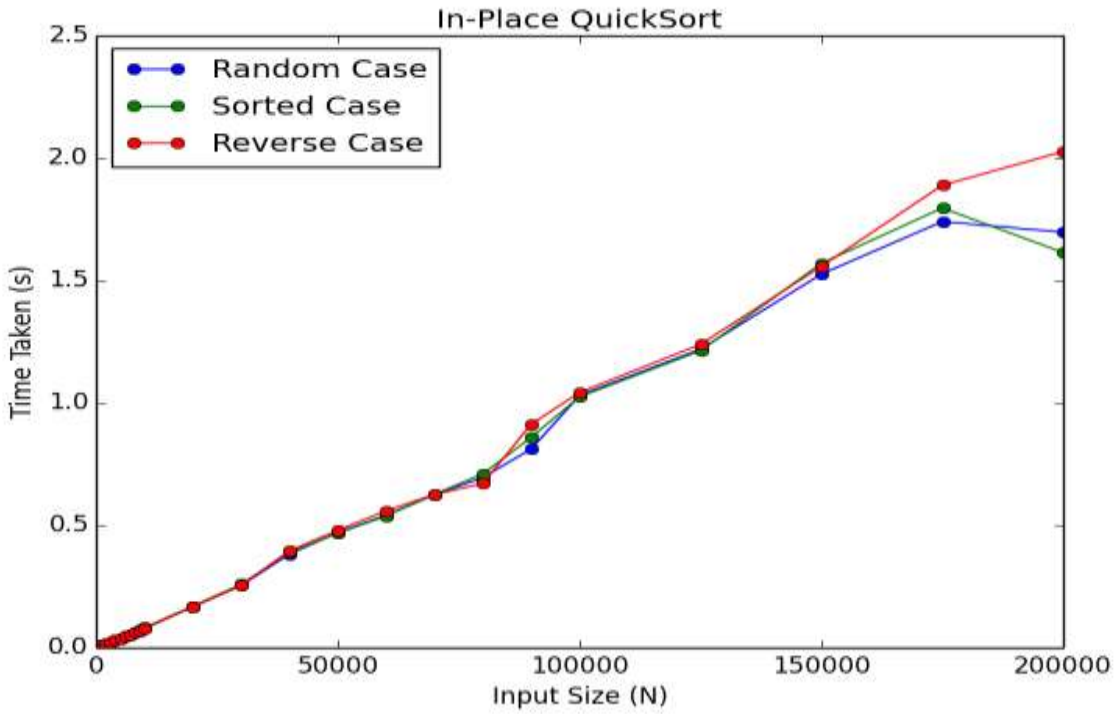

Heap Sort
Random Case
Sorted Case
Reverse Case
Time Taken (s)
Input Size (N)
0
1
2
3
4
5
6
7
0
50000
100000
150000
200000

5.0 In-Place QuickSort

5.1 Algorithm

```
19
20      def inPlaceQuickSort(self, data, left, right):
21          size = right - left + 1
22          #print("InPlace Quick sort on input size = " + str(size))
23          if left >= right:
24              return data
25          if not self.modifiedSort or (size > 10 and self.modifiedSort):
26              pivotIndex, data = self.getPivotRank(data, left, right)
27              #print(str(data) + " pivotIndex=" + str(pivotIndex))
28              newPivotIndex, data = self.inPlacePartition(data, left, right, pivotIndex)
29              #print(str(data) + " newPivotIndex=" + str(newPivotIndex))
30              data = self.inPlaceQuickSort(data, left, newPivotIndex-1)
31              data = self.inPlaceQuickSort(data, newPivotIndex + 1, right)
32          else:
33              data = self.insertionSort(data, left, right)
34          return data
35
```

5.2 Result



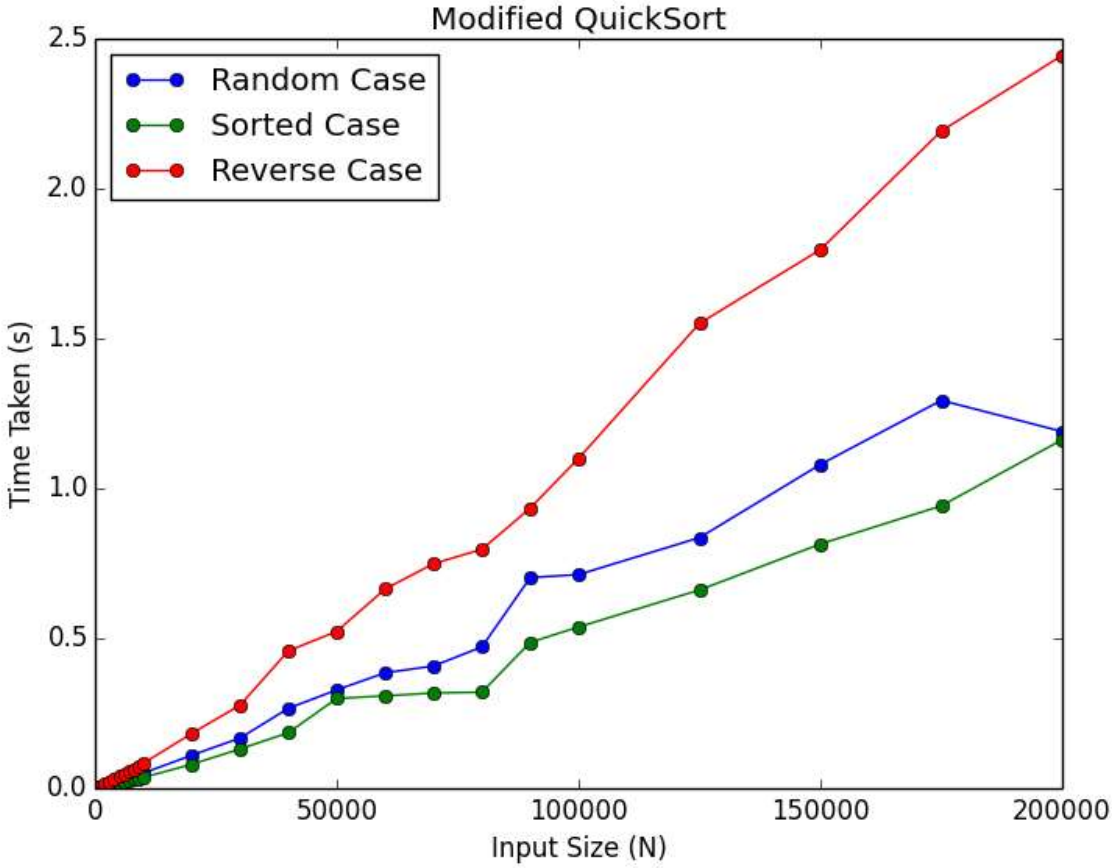
6.0 Modified Quick Sort

6.1 Algorithm

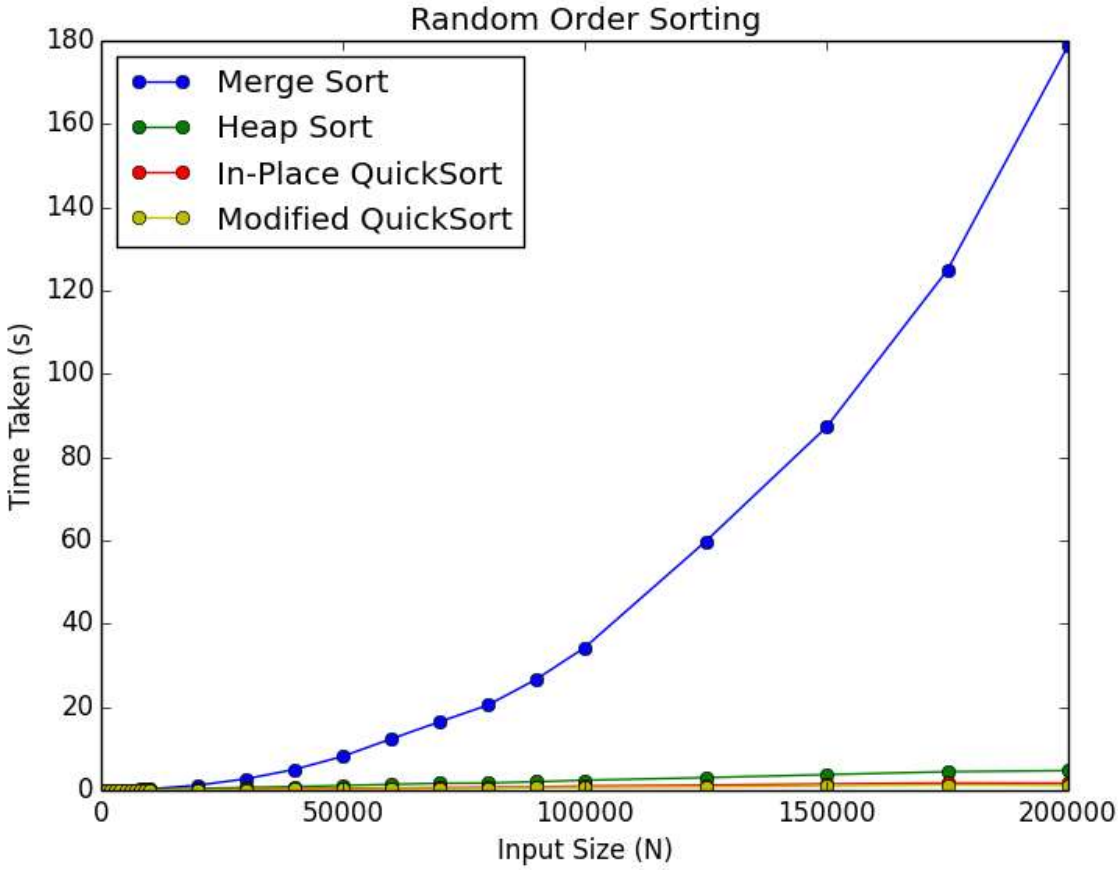
```
19
20      def inPlaceQuickSort(self, data, left, right):
21          size = right - left + 1
22          #print("InPlace Quick sort on input size = " + str(size))
23          if left >= right:
24              return data
25          if not self.modifiedSort or (size > 10 and self.modifiedSort):
26              pivotIndex, data = self.getPivotRank(data, left, right)
27              #print(str(data) + " pivotIndex=" + str(pivotIndex))
28              newPivotIndex, data = self.inPlacePartition(data, left, right, pivotIndex)
29              #print(str(data) + " newPivotIndex=" + str(newPivotIndex))
30              data = self.inPlaceQuickSort(data, left, newPivotIndex-1)
31              data = self.inPlaceQuickSort(data, newPivotIndex + 1, right)
32          else:
33              data = self.insertionSort(data, left, right)
34          return data
35
```

```
36      def insertionSort(self, data, left, right):
37          size = right - left + 1
38          #print("Insertion sort on input size = " + str(size))
39          for j in range(left, right + 1):
40              #print(str(j)+"  "+str(data[j]))
41              key = data[j]
42              i = j - 1
43              while i >= 0 and data[i] > key:
44                  data[i+1] = data[i]
45                  i -= 1
46                  data[i+1] = key
47          return data
48
49      def inPlacePartition(self, data, left, right, pivotIndex):
50          pivotValue = data[pivotIndex]
51          data[pivotIndex], data[right] = data[right], data[pivotIndex] #Move pivot to the end
52          storeIndex = left
53          for i in range(left, right):
54              if data[i] <= pivotValue:
55                  data[i], data[storeIndex] = data[storeIndex], data[i]
56                  storeIndex += 1
57          data[storeIndex], data[right] = data[right], data[storeIndex]
58          return storeIndex, data
59
60      def getPivotRank(self, data, minimum, maximum):
61          rank = random.randint(minimum, maximum)
62          if self.pivotMode == FIRST_PIVOT:
63              rank = minimum
64          elif self.pivotMode == LAST_PIVOT:
65              rank = maximum
66          elif self.pivotMode == MEDIAN_PIVOT:
67              center = math.floor((minimum + maximum)/2)
68              if data[center] < data[minimum]:
69                  data[center], data[minimum] = data[minimum], data[center]
70              if data[maximum] < data[minimum]:
71                  data[maximum], data[minimum] = data[minimum], data[maximum]
72              if data[maximum] < data[center]:
73                  data[maximum], data[center] = data[center], data[maximum]
74              rank = center
75          return rank, data
76
```

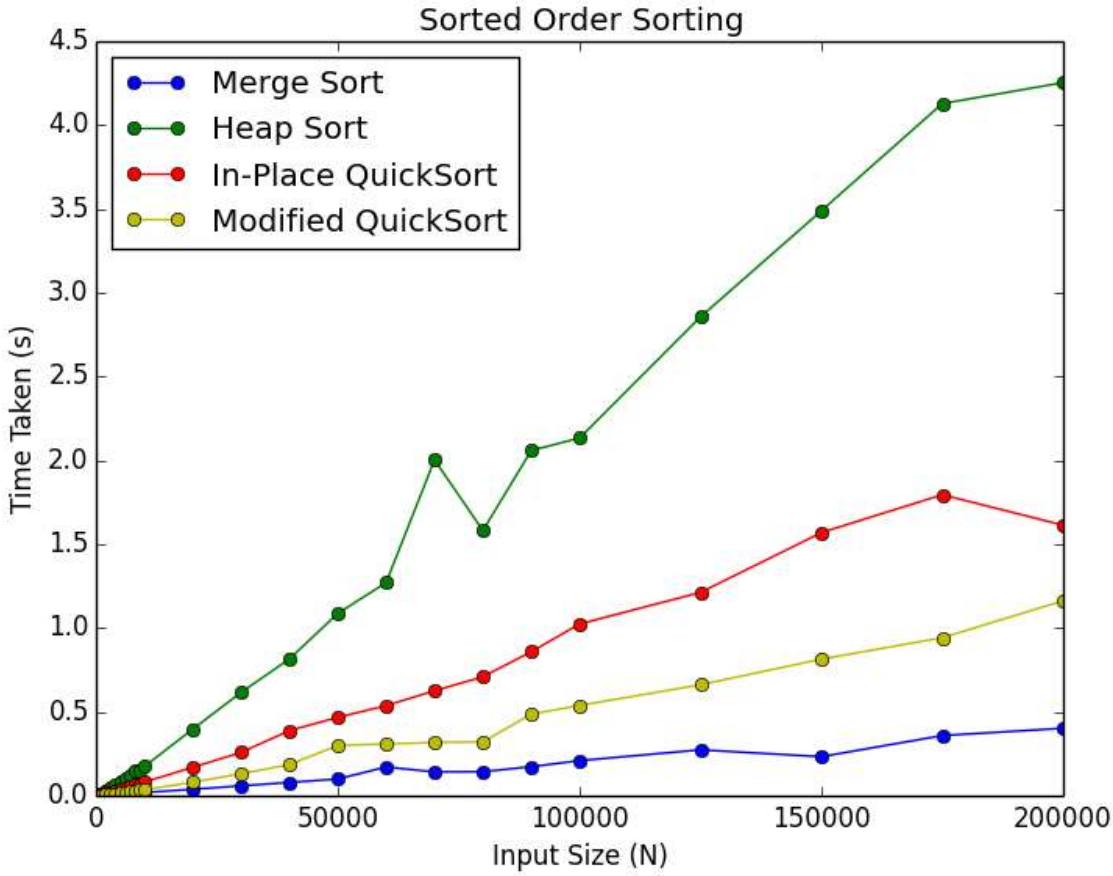
6.2 Result


Modified QuickSort
Random Case
Sorted Case
Reverse Case
Time Taken (s)
Input Size (N)
0.0
0.5
1.0
1.5
2.0
2.5
0
50000
100000
150000
200000

7.0 Random Order Comparison



8.0 Sorted Order Comparison



9.0 Reverse Sorted Order Comparison