**Algorithms and Data Structures**

**Project I:**

**Comparison-based Sorting Algorithms**

**PROJECT MEMBERS**

Varshitha Yataluru

Neela Ayshwaria Alagappan

**INTRODUCTION:**

In this project, some comparison-based algorithms like Insertion Sort, Merge Sort, Heap Sort, In-place Quick Sort and Modified Quick sort are implemented and executed for

various input datasets and the time taken for the executions are recorded and plotted against

the dataset size. In addition to that, 2 special cases of the input array being sorted and reverse

sorted are implemented and performances are recorded.

**IMPLEMENTATION AND OBSERVATIONS:**

**INSERTION SORT:**

Insertion sort is a simple and efficient sorting algorithm efficient only for very small

datasets and mostly sorted lists. The algorithm works by inserting each element into its correct

position in the sorted list by comparing the element with every element in the sorted list. The

operation is repeated until the input array is sorted. The runtime for the implementation is O(n^2)

The insertion sort algorithm is inefficient for larger input datasets when compared to

other algorithms. However, insertion sort is efficient for smaller input datasets.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:

[Array prior to the insertion of x](https://en.wikipedia.org/wiki/File:Insertionsort-before.png)

becomes

[Array after the insertion of x](https://en.wikipedia.org/wiki/File:Insertionsort-after.png)

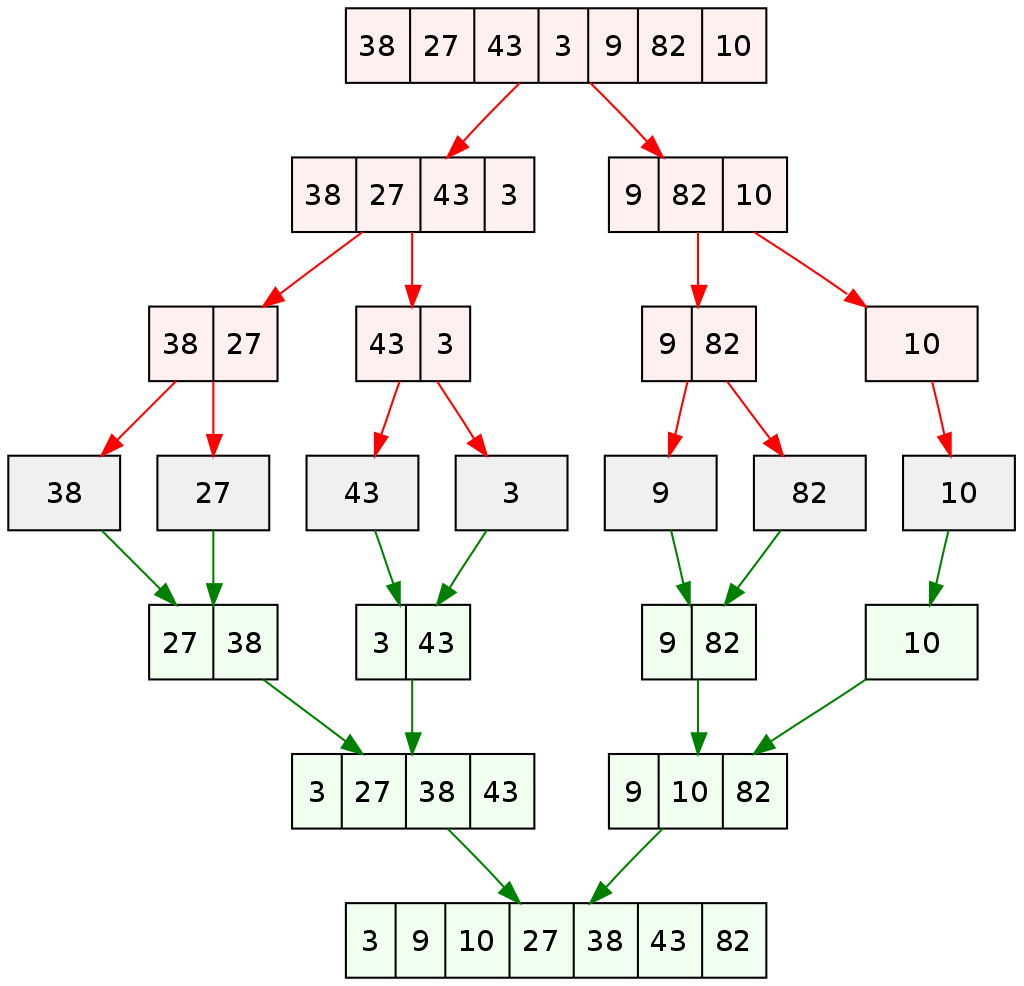
with each element greater than *x* copied to the right as it is compared against *x*.

Time Complexity :

**MERGE SORT:**

Merge sort uses the divide and conquer paradigm. Conceptually, a merge sort works as Divide the unsorted list into *n* sublists, each containing one element (a list of one element is considered sorted).Repeatedly [merge](https://en.wikipedia.org/wiki/Merge_algorithm) sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list. Merge sort is an efficient algorithm compared to the insertion sort. The runtime for the implementation is O(nlogn). However, the implementation takes extra memory and hence when large data sets are given as input it takes a large memory to execute.

An example of merge sort. First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.



**HEAP SORT:**

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a Heap data structure from the given array and then utilizing the Heap to sort the array.

Heap sort algorithm is divided into two basic parts:

1. Creating a Heap of the unsorted list/array.
2. Then a sorted array is created by repeatedly removing the largest/smallest element from the heap and inserting it into the array. The heap is reconstructed after each removal.
3. Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.
4. In the below algorithm, initially heapsort() function is called, which calls heapify() to build the heap.

The sorting method involves below points:

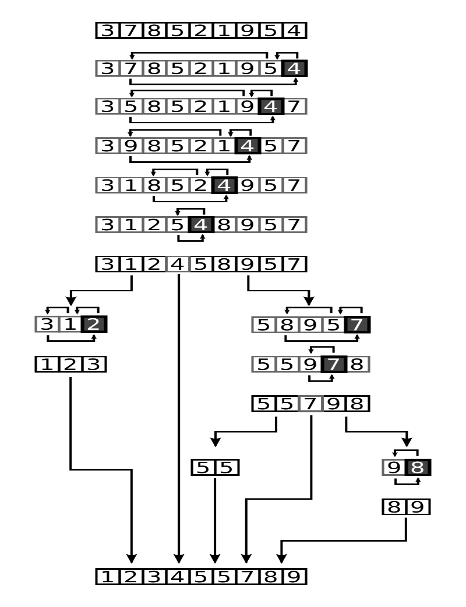
Build a min heap from the input data.

At this point, the smallest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.  
Repeat above steps while size of heap is greater than 1.

**QUICK SORT:**

Quick sort (sometimes called **partition-exchange sort**) similar to merge sort uses a divide and conquer method to sort the input array elements. But unlike merge sort it doesn’t need an extra memory space for sorting.

Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps include Pick an element, called a *pivot*, from the array. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation. [Recursively](https://en.wikipedia.org/wiki/Recursion_(computer_science)) apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.



The base case of the recursion is arrays of size zero or one, which are in order by definition so they never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

**QUICK SORT - MEDIAN OF 3:**

Quicksort with median-of-three partitioning functions nearly the same as normal quicksort with the only difference being how the pivot item is selected. In normal quicksort the first element is automatically the pivot item. This causes normal quicksort to function very inefficiently when presented with an already sorted list. The division will always end up producing one sub-array with no elements and one with all the elements (minus of course the pivot item). In quicksort with median-of-three partitioning the pivot item is selected as the median between the first element, the last element, and the middle element (elements at positions first, middle and last are in ascending order). In the cases of already sorted lists this should take the middle element as the pivot thereby reducing the inefficiency found in normal quicksort.

**COMPARISON OF SORTING ALGORITHMS:**

By observing the table of results, we have plotted a graph for each algorithm to analyze the

practical performance.

**Analysis :**

From the graph, it’s evident that insertion sort is suitable only for small

datasets. Merge sort, quicksort and modified quicksort takes less time than the insertion sort.

However, the modified quicksort takes more execution time when a very large input dataset is

given, then the merge sort and quicksort. The merge sort and quicksort outperform all the other

sorting algorithms and takes less execution time, proving to be efficient.

**COMPLEXITY ANALYSIS :**

## Insertion sort :

**O(n^2)** – From the figure we can observe that insertion sort performs less efficient than rest of the sorting algorithms.

However, insertion sort has got best case when the input is already sorted or almost sorted.

**Merge sort :**

**O(n log(n)) –** From the figure we can observe that merge sort outperforms all the other sorting algorithm in all case.

**Heap sort :**

**O(n log(n))**-- From the figure we can observe that heap sort works efficiently in all the special cases.

**In-place quick sort :**

**O(n^2) –** From the figure we can observe that in-place quick sort performs efficiently for smaller input ranges.

**Median of three quick sort :**

**O(n^2) –** Fromthe figure we can observe that median of three quick sort performs efficiently for smaller input ranges.

**DATA STRUCTURES :**

Array is used for implementing the algorithms.

**PROGRAM CODE :**

**import** random  
**import** matplotlib **as** plt  
**import** time  
**from** sys **import** setrecursionlimit  
  
setrecursionlimit(300000)  
arr = []  
mergearr = []  
case1In = []  
case1Me = []  
case2Me = []  
case2In = []  
case1H = []  
case2H = []  
case1Q=[]  
case2Q=[]  
case1Mq=[]  
case2Mq=[]  
print(**'Enter the input range'**)  
y = int(input())  
**for** x **in** range(y):  
 arr.append(random.randint(1, 50000))  
insertarr = arr[:]  
mergearr = arr[:]  
heapsort = arr[:]  
quicksort = arr[:]  
median = arr[:]  
print(**'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*INSERTION SORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'**)  
  
  
**def** insertion(arr):  
 **for** i **in** range(1, len(arr)):  
 key = arr[i]  
 j = i - 1  
 **while** j >= 0 **and** key < arr[j]:  
 arr[j + 1] = arr[j]  
 j -= 1  
 arr[j + 1] = key  
  
  
start = time.time()  
print(**'Array before sorting'**)  
**for** i **in** range(len(arr)):  
 print(**"%d"** % arr[i], **' '**, end=**''**)  
print(insertion(arr))  
print(**'Array after Insertion sorting'**)  
**for** i **in** range(len(arr)):  
 print(**"%d"** % arr[i], **' '**, end=**''**)  
print(**"\n"**)  
end = time.time()  
elapse = end - start  
case1In = arr  
start1 = time.time()  
print(**'Case 1(Insertion):Running sorting algorithm on the sorted array'**)  
**for** i **in** range(len(case1In)):  
 print(**"%d"** % case1In[i], **' '**, end=**''**)  
print(insertion(case1In))  
print(**"\n"**)  
print(**'Result of sorting on Sorted array'**)  
**for** i **in** range(len(case1In)):  
 print(**"%d"** % case1In[i], **' '**, end=**''**)  
end1 = time.time()  
elapse1 = end1 - start1  
case2In = arr  
**def** Reverse(case2In):  
 case2In.reverse()  
 **return** case2In  
  
  
print(**"\n"**)  
print(**'Case 2(Insertion): Running sorting algorithm for reversed input array'**)  
print(Reverse(case2In))  
start3 = time.time()  
print(insertion(case2In))  
**for** i **in** range(len(case2In)):  
 case2In[i]  
 print(**"%d"** % case2In[i], **' '**, end=**''**)  
end3 = time.time()  
elapse3 = end3 - start3  
print(**"\n"**)  
print(**"Time taken for insertion sorting on an array"**, elapse \* 100)  
print(**"Time taken for insertion sorting on an sorted array"**, elapse1 \* 100)  
print(**"Time taken for insertion sorting on an reversely sorted array"**, elapse3 \* 100)  
print(  
 **"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MERGE SORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
startMe = time.time()  
  
  
**def** mergeSort(mergearr):  
 **if** len(mergearr) > 1:  
 mid = len(mergearr) // 2  
 L = mergearr[:mid]  
 R = mergearr[mid:]  
 mergeSort(L)  
 mergeSort(R)  
 i = j = k = 0  
 **while** i < len(L) **and** j < len(R):  
 **if** L[i] < R[j]:  
 mergearr[k] = L[i]  
 i += 1  
 **else**:  
 mergearr[k] = R[j]  
 j += 1  
 k += 1  
  
 **while** i < len(L):  
 mergearr[k] = L[i]  
 i += 1  
 k += 1  
  
 **while** j < len(R):  
 mergearr[k] = R[j]  
 j += 1  
 k += 1  
  
  
**def** printList(mergearr):  
 **for** i **in** range(len(mergearr)):  
 print(**"%d"** % mergearr[i], **' '**, end=**''**)  
 print()  
  
  
printList(mergearr)  
mergeSort(mergearr)  
print(**"\n"**)  
print(**"after sorting"**)  
printList(mergearr)  
endMe = time.time()  
elapseMe = endMe - startMe  
startMe1 = time.time()  
case1Mer = mergearr  
print(**"\n"**)  
print(**'Case 1(Merge) :Running sorting algorithm on the below sorted array'**)  
**for** i **in** range(len(case1Mer)):  
 print(**"%d"** % case1Mer[i], **' '**, end=**''**)  
print(mergeSort(case1Mer))  
print(**"\n"**)  
print(**'Result of sorting on Sorted array'**)  
**for** i **in** range(len(case1Mer)):  
 print(**"%d"** % case1Mer[i], **' '**, end=**''**)  
endMe1 = time.time()  
elapseMe1 = endMe1 - startMe1  
case2Mer = mergearr  
startMe3 = time.time()  
  
  
**def** Reverse(case2Mer):  
 case2Mer.reverse()  
 **return** case2Mer  
  
  
print(**"\n"**)  
print(**'Case 2(Merge): Running sorting algorithm for reversed input array'**)  
print(Reverse(case2Mer))  
print(mergeSort(case2Mer))  
**for** i **in** range(len(case2Mer)):  
 case2Mer[i]  
 print(**"%d"** % case2Mer[i], **' '**, end=**''**)  
print(**"\n"**)  
endMe3 = time.time()  
elapseMe3 = endMe3 - startMe3  
print(**"Time taken for Merge sorting on an array"**, elapseMe \* 100)  
print(**"Time taken for Merge sorting on an sorted array"**, elapseMe1 \* 100)  
print(**"Time taken for Merge sorting on an reversely sorted array"**, elapseMe3 \* 100)  
  
print(**'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*HEAP SORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'**)  
  
**def** heapify(heapsort, n, i):  
 smallest = i  
 l = 2 \* i  
 r = 2 \* i + 1  
  
  
 **if** l < n **and** heapsort[i] < heapsort[l]:  
 smallest = l  
  
  
 **if** r < n **and** heapsort[smallest] < heapsort[r]:  
 smallest = r  
  
 **if** smallest != i:  
 heapsort[i],heapsort[smallest] = heapsort[smallest],heapsort[i]  
  
  
 heapify(heapsort, n, smallest)  
  
  
**def** heapSort(heapsort):  
 n = len(heapsort)  
  
  
 **for** i **in** range(n, -1, -1):  
 heapify(heapsort, n, i)  
  
  
 **for** i **in** range(n-1, 0, -1):  
 heapsort[i], heapsort[0] = heapsort[0], heapsort[i]  
 heapify(heapsort, i, 0)  
  
n = len(heapsort)  
print(**"\n"**)  
print(**'Array before sorting'**)  
**for** i **in** range(n):  
 print(**"%d"** %heapsort[i], **' '**, end=**''**)  
startH=time.time()  
print(**"\n"**)  
print(**"Array after sorting"**)  
heapSort(heapsort)  
**for** i **in** range(n):  
 print(**"%d"** %heapsort[i], **' '**, end=**''**)  
endH=time.time()  
elapseH=endH-startH  
startH1=time.time()  
case1H=heapsort  
print(**"\n"**)  
print(**'Case 1(Heap) :Running sorting algorithm for sorted input array'**)  
**for** i **in** range(len(case1H)):  
 print (**"%d"** %case1H[i],**' '**,end=**""**)  
print(**'Result of sortin on Sorted array'**)  
heapSort(case1H)  
**for** i **in** range(n):  
 print(**"%d"** %heapsort[i], **' '**, end=**''**)  
endH1=time.time()  
elapseH1=endH1-startH1  
start2H=time.time()  
case2H=heapsort  
**def** Reverse(case2H):  
 case2H.reverse()  
 **return** case2H  
print(**"\n"**)  
print(**'Case 2(Heap): Running sorting alogrithm for reversed input array'**)  
print(Reverse(case2H))  
heapSort(case2H)  
**for** i **in** range(n):  
 print(**"%d"** %case2H[i], **' '**, end=**''**)  
end2H=time.time()  
elapse2H=end2H-start2H  
print(**"\n"**)  
print(**'Time elapse for heap sort'**)  
print(**"Time taken for Heap sorting on an array"**, elapseH \* 100)  
print(**"Time taken for Heap sorting on an sorted array"**, elapseH1 \* 100)  
print(**"Time taken for Heap sorting on an reversely sorted array"**, elapse2H \* 100)  
  
print(**'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*IN PLACE QUICKSORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'**)  
  
  
**def** partition(quicksort, low, high):  
 i = (low - 1)  
 pivot = quicksort[high]  
 **for** j **in** range(low, high):  
 **if** quicksort[j] <= pivot:  
 i = i + 1  
 quicksort[i], quicksort[j] = quicksort[j], quicksort[i]  
 quicksort[i + 1], quicksort[high] = quicksort[high], quicksort[i + 1]  
 **return** (i + 1)  
  
  
**def** quickSort(quicksort, low, high):  
 **if** low < high:  
 pi = partition(quicksort, low, high)  
 quickSort(quicksort, low, pi - 1)  
 quickSort(quicksort, pi + 1, high)  
  
  
startQ = time.time()  
print(**"array before sorting"**)  
**for** i **in** range(n):  
 print(**"%d"** % quicksort[i], **' '**, end=**''**)  
n = len(quicksort)  
quickSort(quicksort, 0, n - 1)  
print(**"\n"**)  
print(**"Sorted array is:"**)  
**for** i **in** range(n):  
 print(**"%d"** % quicksort[i], **' '**, end=**''**),  
endQ = time.time()  
elapseQ = endQ - startQ  
start1Q = time.time()  
case1Q = quicksort  
print(**"\n"**)  
print(**'Case 1(In-place Quick): Running sorting algorithm for sorted input array'**)  
**for** i **in** range(len(case1Q)):  
 print(**"%d"** % case1Q[i], **' '**, end=**''**)  
print(**'Result of sorting on Sorted array'**)  
quickSort(case1Q, 0, n - 1)  
**for** i **in** range(n):  
 print(**"%d"** % case1Q[i], **' '**, end=**''**),  
end1Q = time.time()  
elapse1Q = end1Q - start1Q  
start2Q = time.time()  
case2Q = quicksort  
  
  
**def** Reverse(case2Q):  
 case2Q.reverse()  
 **return** case2Q  
  
  
print(**"\n"**)  
print(**'Case 2(In-place Quick): Running sorting algorithm for reverse input array'**)  
print(Reverse(case2Q))  
quickSort(case2Q, 0, n - 1)  
**for** i **in** range(n):  
 print(**"%d"** % case2Q[i], **' '**, end=**''**),  
end2Q = time.time()  
elapse2Q = end2Q - start2Q  
print(**"\n"**)  
print(**"Time taken for Quick sorting on an array"**, elapseQ \* 100)  
print(**"Time taken for Quick sorting on an sorted array"**, elapse1Q \* 100)  
print(**"Time taken for Quick sorting on an reversely sorted array"**, elapse2Q \* 100)  
  
print(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MEDIAN OF THREE QUICK SORT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
  
  
**def** quickSortM(L, ascending=**False**):  
 quicksorthelp(L, 0, len(L), ascending)  
  
  
**def** insertionM(L, low, high):  
 **for** i **in** range(low, high):  
 key = L[i]  
 j = i - 1  
 **while** j >= 0 **and** key < L[j]:  
 L[j + 1] = L[j]  
 j -= 1  
 L[j + 1] = key  
  
  
**def** quicksorthelp(L, low, high, ascending=**False**):  
 result = 0  
 **if** (low + 10 <= high):  
 insertionM(L, low, high)  
 **else**:  
 pivot\_location, result = Partition(L, low, high, ascending)  
 result += quicksorthelp(L, low, pivot\_location, ascending)  
 result += quicksorthelp(L, pivot\_location + 1, high, ascending)  
 **return** result  
  
  
**def** median\_of\_three(L, low, high):  
 mid = (low + high - 1) // 2  
 a = L[low]  
 b = L[mid]  
 c = L[high - 1]  
 **if** a <= b <= c:  
 **return** b, mid  
 **if** c <= b <= a:  
 **return** b, mid  
 **if** a <= c <= b:  
 **return** c, high - 1  
 **if** b <= c <= a:  
 **return** c, high - 1  
 **return** a, low  
  
  
**def** Partition(L, low, high, ascending=**False**):  
 result = 0  
 pivot, pidx = median\_of\_three(L, low, high)  
 L[low], L[pidx] = L[pidx], L[low]  
 i = low + 1  
 **for** j **in** range(low + 1, high, 1):  
 result += 1  
 **if** (ascending **and** L[j] < pivot) **or** (**not** ascending **and** L[j] > pivot):  
 L[i], L[j] = L[j], L[i]  
 i += 1  
 L[low], L[i - 1] = L[i - 1], L[low]  
 **return** i - 1, result  
  
  
startMq = time.time()  
**for** i **in** range(n):  
 print(**"%d"** % median[i], **' '**, end=**''**)  
quickSortM(median, **True**)  
print(**"\n"**)  
print(**'Array after sorting:'**)  
**for** i **in** range(n):  
 print(**"%d"** % median[i], **' '**, end=**''**)  
endMq = time.time()  
elapseMq = endMq - startMq  
start1Mq = time.time()  
case1Mq = median  
print(**"\n"**)  
print(**'Case 1(Modified Quick) :Running sorting algorithm for sorted input array'**)  
**for** i **in** range(len(case1Mq)):  
 print(**"%d"** % case1Mq[i], **' '**, end=**''**)  
print(**"\n"**)  
print(**'Result of sorting on Sorted array'**)  
quickSortM(case1Mq, **True**)  
**for** i **in** range(n):  
 print(**"%d"** % case1Mq[i], **' '**, end=**''**),  
end1Mq = time.time()  
elapse1Mq = end1Mq - start1Mq  
startMq2 = time.time()  
case2Mq = median  
  
  
**def** Reverse(case2Mq):  
 case2Mq.reverse()  
 **return** case2Mq  
  
  
print(**"\n"**)  
print(**'Case 2(Modified Quick): Running sorting algorithm for reverse input array'**)  
print(Reverse(case2Mq))  
quickSortM(case2Mq, **True**)  
**for** i **in** range(n):  
 print(**"%d"** % case2Mq[i], **' '**, end=**''**),  
end2Mq = time.time()  
elapse2Mq = end2Mq - startMq2  
print(**"\n"**)  
print(**"Time taken for Modified Quick sorting on an array"**, elapseMq \* 100)  
print(**"Time taken for Modified Quick sorting on an sorted array"**, elapse1Mq \* 100)  
print(**"Time taken for Modified Quick sorting on an reversely sorted array"**, elapse2Mq \* 100)  
print(**"\n"**)  
  
print(**'Time complexity'**)  
print(**"Insertion"**)  
print(elapse \* 100)  
print(elapse1 \* 100)  
print(elapse3 \* 100)  
  
print(**'\n'**)  
print(**"Merge"**)  
print(elapseMe \* 100)  
print(elapseMe1 \* 100)  
print(elapseMe3 \* 100)  
  
print(**'\n'**)  
print(**"Heap"**)  
print(elapseH \* 100)  
print(elapseH1 \* 100)  
print(elapse2H \* 100)  
  
print(**'\n'**)  
print(**"In-place Quick"**)  
print(elapseQ \* 100)  
print(elapse1Q \* 100)  
print(elapse2Q \* 100)  
  
print(**'\n'**)  
print(**"Modified Quick"**)  
print(elapseMq \* 100)  
print(elapse1Mq \* 100)  
print(elapse2Mq \* 100)

**RESULTS:**

**Input size :1000**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Random Array** | **Sorted Array** | **Reverse Sorted Array** |
| **Insertion** | 22.57354259 | 0.508833 | 50.05856 |
| **Merge** | 1.30868 | 5.842996 | 1.403451 |
| **Heap** | 2.040052 | 6.985664 | 1.821089 |
| **In-place Quick** | 1.311779 | 57.4203 | 12.11705 |
| **Modified Quick** | 20.99733 | 0.408554 | |  | | --- | | 45.69271 | |

**Input Size: 2000**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Random Array** | **Sorted Array** | **Reverse Sorted Array** |
| **Insertion** | 99.08394814 | 2.935982 | 314.6614 |
| **Merge** | 3.368306 | 10.99942 | 9.99825 |
| **Heap** | 8.681893 | 9.255362 | 9.407878 |
| **In-place Quick** | 9.28545 | 225.1514 | 64.88199 |
| **Modified Quick** | 128.291 | 5.377674 | 203.782 |

**Input Size: 5000**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Random Array** | **Sorted Array** | **Reverse Sorted Array** |
| **Insertion** | 506.7486525 | 7.732177 | 1011.418 |
| **Merge** | 12.66778 | 17.80524 | 11.92641 |
| **Heap** | 26.18985 | 40.91618 | 50.00417 |
| **In-place Quick** | 49.96128 | 1183.283 | 366.867 |
| **Modified Quick** | 523.2923 | 7.934737 | 1090.635 |

**Input Size: 10000**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Random Array** | **Sorted Array** | **Reverse Sorted Array** |
| **Insertion** | 2338.158 | 69.68572 | 4830.774 |
| **Merge** | 64.30566 | 159.9885 | 175.0047 |
| **Heap** | 94.93434 | 175.0812 | 169.985 |
| **In-place Quick** | 164.9359 | 5576.446 | 3021.323 |
| **Modified Quick** | 2347.558 | 59.97226 | 4955.001 |

**RANDOM ARRAY:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| InputSize | Insertion | Merge | Heapsort | In-place | Median of 3 |
| 1000 | 6.383061 | 1.995134 | 1.396251 | 2.094364 | 8.278227 |
| 2000 | 23.33696 | 3.49381 | 2.792501 | 3.291225 | 23.93358 |
| 5000 | 139.5298 | 9.17511 | 7.184744 | 7.779193 | 134.0447 |
| 10000 | 2338.158 | 64.30566 | 94.93434 | 164.9359 | 2347.558 |
| 20000 | 10216.33 | 308.7902 | 190.1837 | 335.7426 | 10479.29 |

**SORTED DATA:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input Size | Insertion | Merge | Heapsort | In-place | Median of 3 |
| 1000 | 1.299644 | 1.79491 | 2.094412 | 14.66074 | 1.396084 |
| 2000 | 2.59347 | 3.390789 | 4.090238 | 53.95579 | 2.595496 |
| 5000 | 6.382966 | 8.574033 | 10.86712 | 324.3358 | 6.382895 |
| 10000 | 69.68572 | 159.9885 | 175.0812 | 5576.446 | 59.97226 |
| 20000 | 290.7438 | 340.2832 | 360.2955 | 26976.68 | 298.4941 |

**REVERSE SORTED DATA**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| InputSize | Insertion | Merge | Heapsort | In-place | Median of 3 |
| 1000 | 10.37211 | 1.193523 | 1.795244 | 10.07652 | 10.56807 |
| 2000 | 43.08805 | 2.094531 | 2.691698 | 34.00836 | 42.38725 |
| 5000 | 259.9083 | 5.983973 | 7.280517 | 195.4739 | 257.5113 |
| 10000 | 4830.774 | 175.0047 | 169.985 | 3021.323 | 4955.001 |
| 20000 | 21280.76 | 349.6951 | 359.9837 | 12895.18 | 21300.62 |