1. Implement and compare the following sorting algorithm:

- Merge sort
- Heap sort
- Quick sort (Regular quick sort* and quick sort using 3 medians)
- Insertion sort
- Selection sort
- Bubble sort

Merge Sort

Merge sorting technique is a divide and conquer approach which follows a top-down sorting method that involves dividing the given array using partition technique into individual bits then sorting and merging them simultaneously to form a sorted list of elements .

Merge Sort is a stable algorithm because it does not change the order of the elements even after the sorting is finished even though merge sort uses partition technique it compares adjacent elements to sort the array which does not change the order of the duplicate elements in the array.

Implementation of Merge sort

Merge sort implementation use arrays as data structures. The input to the arrays is given by a random number generator function-**Random()** with values ranging from 0-1000. The length of the array is decided interactively by the user.

The implementation of merge sort is done by two functions

- **void merge_sort(int array[], int first, int last)** : This function takes three parameters the input array, starting index and last index of the input array. The functionality involves computing the middle element and recursively dividing the entire array into individual bits using recursion and a further call to the function merge.

- **void merge(int array[], int first, int middle, int last)**: This function takes four inputs partitioned array, the first index of the array, the last index of the array and the middle element computed by the function merge_sort. Merge_sort function compares the individual bits and adds them to the final merged sorted array one by one depending on the value of the elements from first-subarray or the second sub-array.

Time complexity :

Best case: O(n logn)

Average case: O(nlogn)

Worst case: O(nlogn)

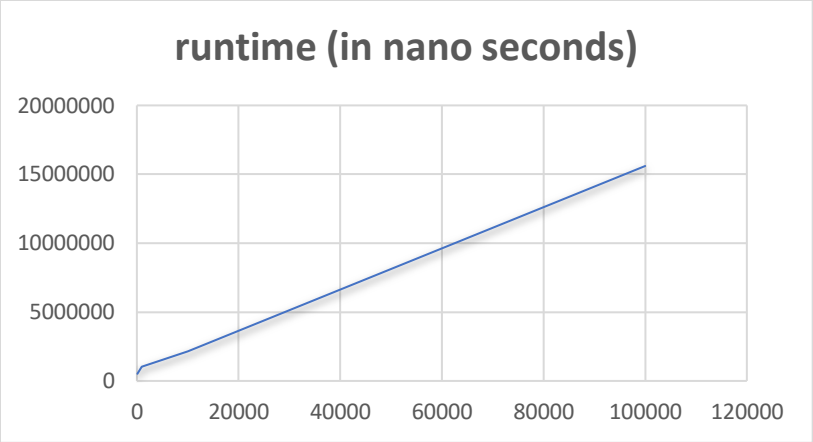
Space Complexity: O(n)

where n represents the length of the input array.

The elapsed time of the entire merge sorting is calculated by using the function .nanoTime() at the start of the process and at the end of the process calculating the difference gives you the total time taken for sorting the input array.

The below chart and table represent the Input sizes vs Run time for the merge sorting implemented in the Project. The inputs being randomized by random number generator

Input sizes	runtime (in nano seconds)
100	518600
1000	1023100
10000	2140000
100000	15603000



Optimization : Optimization to the above implemented sorting involves parallel recursion and parallel merging .An intuitive approach which involves the parallelization of those recursive calls which gives and improvement of just BigOmega(log n) which is not that comprehensive but using **Parallel multiway merge sort** which involves the sorting to be done by k-way merge method in which k sorted sequences are merged together with multiple parallel processes. The running time is then given by O(p log(n/p) log(n)) where p represents the number of parts the data is divided by using partition.

Comparison with other sorting algorithms

Merge sorting technique is more efficient when we use linked lists as data structures for sorting as implementing sorting using linked lists gets more complex when implemented using heap or quick sort. Merge sort uses auxiliary O(1) space in case of linked lists. Merge sort is a stable algorithm which is more efficient at handling slow-access sequential media.

Heap sort

Heap sort is a comparison based sorting technique which first builds a max-heap binary tree and sorts them by replacing each element with the root and removing them from the tree. It is more likely viewed as an improvement to the selection sort.

Heap sort is an in-place ,unstable algorithm as it does not compare adjacent elements rather it replaces the maximum element with the root and removes them by placing it in the sorted region.

Implementation of Heap sort

Heap sort implementation use arrays as data structures. The input to the arrays is given by a random number generator function-**Random()** with values ranging from 0-1000. The length of the array is decided interactively by the user.

Implementation of Heap sort:

The implementation of heap sort involves three functions:

- **void heap_sort(int []array):** This function takes the input array as input and calls the build_heap() function which builds the max heap , then swaps and heapifies the structure based on the max-heap properties.

- **void build_heap(int []arrray):** This functions builds the max-heap after the call from the heap_sort().

- **void heapify(int array[],int j , int m):** This function takes three inputs the input array and two other indexes to heapify between those indexes. This functions determines the left and right childs of the index j with values obtained it checks whether all the properties of the max heap are satisfied and places them accordingly in the array

Time complexity :

Best case: O(n logn)

Average case: O(nlogn)

Worst case: O(nlogn)

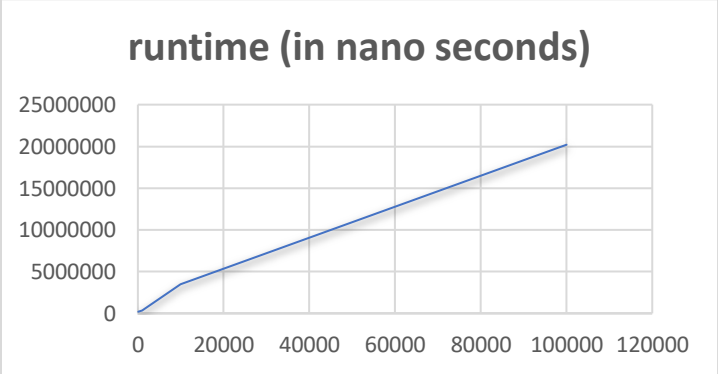
Space Complexity: O(1)

where n represents the length of the input array.

The elapsed time of the entire heap sorting is calculated by using the function .nanoTime() at the start of the process and at the end of the process calculating the difference gives you the total time taken for sorting the input array.

The below chart and table represent the Input sizes vs Run time for the Heap sorting implemented in the Project. The inputs being randomized by random number generator

Input Size	runtime (in nano seconds)
100	201600
1000	328900
10000	3499400
100000	20199500



Optimization: The existing heap sorting technique swaps only the largest/smallest element in the array at a time but by using **Two-swap method** which is the optimized version of the heap sort involving two-swaps for the construction and the sorting of the heap at a time which reduces the time complexity by 30-50%.

Comparison with other sorting algorithms

Heap sort is compared to quick sort most of the times but with the worst-case run time complexity of quick sort is O(n^2) which is not efficient in case of larger datasets. Heap sort has several drawbacks when compared to merge sort it being not a stable algorithm and merge sort can be parallelized easily and also easy to implement on linked lists.

Quick Sort-Regular method

Quick sort is follows divide and conquer approach to sort the arrays by picking an element as the pivot and sorting the elements around that pivot element by placing all the lower elements to the left of the pivot and all the higher value elements to the right of the pivot.

Quick sort is not a stable algorithm as it doesn't involve swapping or changing the position of adjacent elements which leads to its un-stability. The algorithm can be made stable by swapping it with the indexes and using it as comparison parameter.

Implementation of Quick sort-Regular method

Quick sort implementation use arrays as data structures. The input to the arrays is given by a random number generator function-**Random()** with values ranging from 0-1000. The length of the array is decided interactively by the user.

The implementation of quick sort is done by functions

- **void recursive_Quicksort(int array[], int first, int last)** : This recursive function takes three parameters as input the first one being the input array, the starting index and the ending index of the input array. With the partition index returned by the partition_quicksort function it recursively checks each and every index of the input array by dividing the array into two parts (i. e,) before the pivot element and after the pivot element.

- **int partition_quicksort(int array[], int first, int last)**: This function takes its call from the recursive_Quicksort function with three parameters the first one being the input array, the starting index and the ending index of the input array. The pivot element is chosen as the first element of the input array, then every value is compared with the pivot element and swapped and the partitioned index is returned to the left of which the sorted array is present.

Time complexity :

Best case: O(n log n)

Average case: O(n log n)

Worst case: O(n^2) when the largest/smallest element is picked as the pivot.

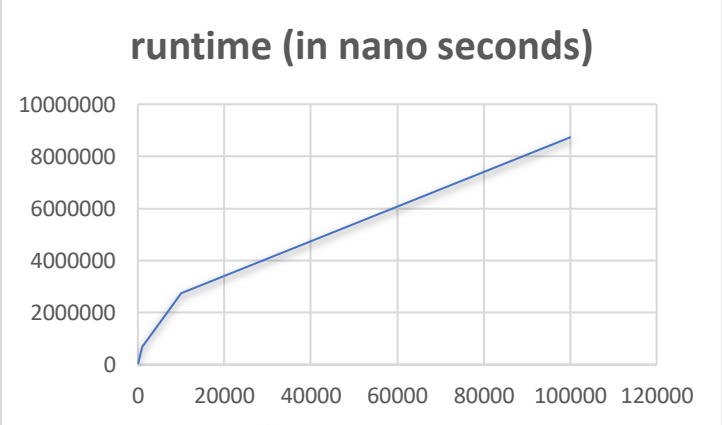
Space Complexity: O(log n)

where n represents the length of the input array.

The elapsed time of the entire quick sorting is calculated by using the function .nanoTime() at the start of the process and at the end of the process calculating the difference gives you the total time taken for sorting the input array.

The below chart and table represent the Input sizes vs Run time for the quick sorting implemented in the Project.

Input size	runtime (in nano seconds)
100	67700
1000	692200
10000	2743400
100000	8731000



Optimization: Optimization for the quick sorting technique is done by using **Tail call method** by making using of at-most O(Log n) space to be used by not including the already sorted subarray and to iterate sort through only the larger side.

Quick Sort-3 median approach

Implementation of 3 median method

Quick sort using 3 median approach uses the following methods:

- **void quick_sort3(int array[], int first, int last):** This recursive function takes three parameters as input the first one being the input array, the starting index and the ending index of the input array. It first checks if the length of the array is greater or lesser than 3 then implements accordingly. If the size is less than three the function sort_quick() function is called as it does not require any partition but if the size is greater than three the respective three_median(), partition_quicksort3() functions are called and then recursively sorted using the same function.

- **void three_median(int array[], int first, int last):** This function takes three parameters as input the first one being the input array, the starting index and the ending index of the input array. This function returns the median pivot element based on the 3-median approach which checks the values of the first, middle and last elements of the array and swapping them based on their values. This process is repeated for each recursive call to the quick_sort3() method.

- **void partition_quicksort3(int array[], int first, int last ,int pivot):** This function takes four parameters as input the first one being the input array, the starting index ,the ending index of the input array and the pivot element. The functionality is similar to that of the partition() function in regular quick sort and returns the partitioned index.

- **void sort_quick(int array[],int first, int last):** This function takes three parameters as input the first one being the input array, the starting index and the ending index of the input array. The functionality is that if the size is less than three of the input array it swaps the positions of the elements by direct comparison without any calls to other functions.

Time complexity :

Best case: O(n log n)

Average case: O(n log n)

Worst case: O(n^2)

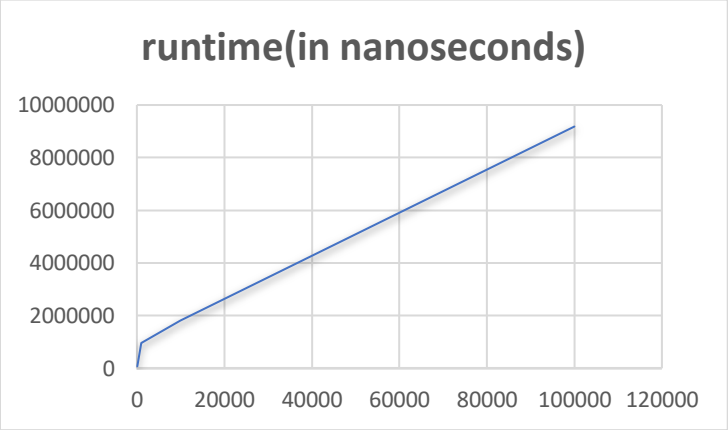
Space Complexity: O(log n)

where n represents the length of the input array.

The elapsed time of the entire quick sorting is calculated by using the function .nanoTime() at the start of the process and at the end of the process calculating the difference gives you the total time taken for sorting the input array.

The below chart and table represent the Input sizes vs Run time for the quick sorting 3-median approach implemented in the Project.

Input size	runtime(in nanoseconds)
100	73000
1000	964300
10000	1803600
100000	9182600



Comparison with other sorting algorithms

The best competitor of quick sort is heap sort but the average time complexity of heap sort is lesser when compared to the in-place quick sort. When compared to the merge sort the quick sort's worst case time complexity of O(n^2) makes it a less efficient sorting algorithm but the disadvantage of merge sort is that it requires O(n) auxiliary space which is higher than the in-place quick sort algorithm with O(log n).

Insertion Sort

Insertion sort is a simple sorting technique with the simplest implementation and not much efficient when compared to other sorting algorithms but is much efficient to smaller data sets.

Insertion sort is an in-place , stable and adaptive sorting algorithm.

Implementation of Insertion sort

Insertion sort implementation use arrays as data structures. The input to the arrays is given by a random number generator function-**Random()** with values ranging from 0-1000. The length of the array is decided interactively by the user.

The implementation of Insertion sort is done by a single function:

- **void insertion_sort(int array[]):** This function takes the input array as the parameter and iteratively compares the elements finds the correct position for the element to be placed and leaves the sorted region behind .

Time complexity :

Best case: O(n)

Average case: O(n^2)

Worst case: O(n^2)

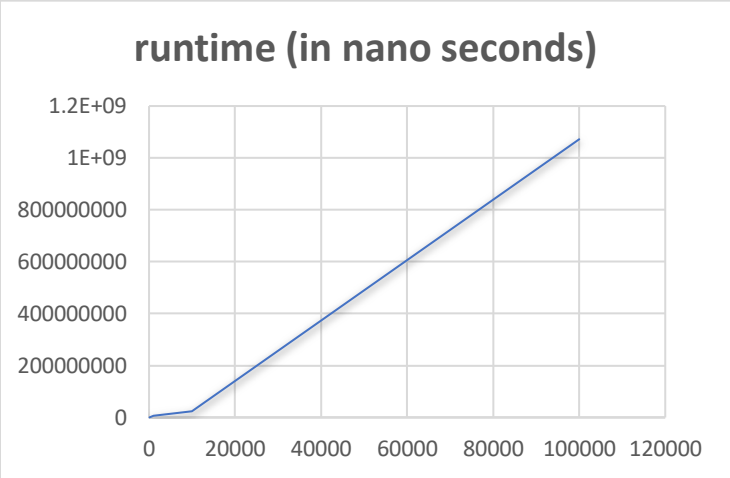
Space Complexity: O(1)

where n represents the length of the input array.

The elapsed time of the entire insertion sorting is calculated by using the function .nanoTime() at the start of the process and at the end of the process calculating the difference gives you the total time taken for sorting the input array.

The below chart and table represent the Input sizes vs Run time for the insertion sorting approach implemented in the Project.

Input size	runtime (in nano seconds)
100	102500
1000	5349300
10000	24822100
100000	1.072E+09



Optimization: Optimization to the insertion sort can be done by Binary insertion sort which implements the binary search technique to find the next element in each iteration though it improves the searching from O(n) to O(log n) but still the sorting entirely takes O(n^2). Further if we use skip list method which is implemented using linked lists reduces the run time for insertion sort to O(log n)

Comparing with other sorting algorithms: When compared to other advanced sorting algorithms like merge, heap and quick sorts insertion sort is less efficient but when compared to bubble and selection sort the best case complexity is O(n) which is faster than the other two sorting techniques. It is advantageous because of its adaptiveness and the in-place property which uses O(1) auxiliary storage for only swapping.

Selection sort

Selection sort is a simple sorting algorithm whose performance is inefficient than the similar insertion sort. It consists of two sorted and unsorted sub-lists after each iteration the length of the sorted sub-list increases with initially the unsorted sub-list being the entire input array.

Selection sort is an in-place, unstable sorting algorithm as it does not preserve the relative order of the duplicate elements in the array.

Implementation of Selection sort

Selection sort implementation use arrays as data structures. The input to the arrays is given by a random number generator function-**Random()** with values ranging from 0-1000. The length of the array is decided interactively by the user.

Implementation involves the call to the function

- **void selection_sort(int []array):** This function implements the selection sort by dividing the array into two parts. It iteratively searches the smallest element in the unsorted sub-list and places it in the sorted sub-list, incrementing the sorted sub-list length for each iteration and replacing the new minimum to the most recent sorted element.

Time complexity :

Best case: O(n^2)

Average case: O(n^2)

Worst case: O(n^2)

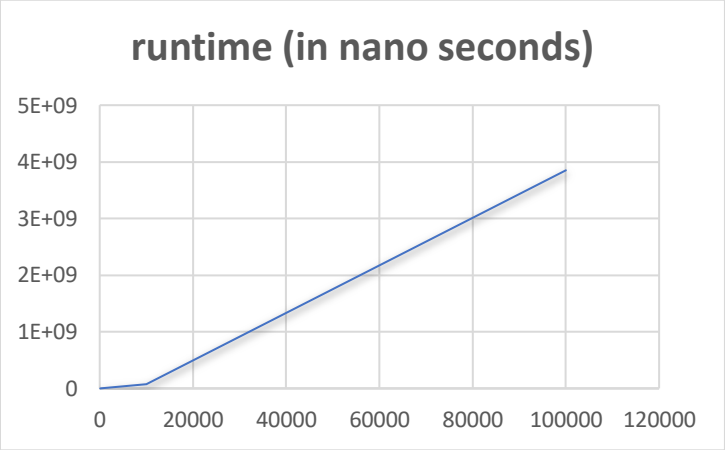
Space Complexity: O(1)

where n represents the length of the input array.

The elapsed time of the entire Selection sorting is calculated by using the function .nanoTime() at the start of the process and at the end of the process calculating the difference gives you the total time taken for sorting the input array.

The below chart and table represent the Input sizes vs Run time for the selection sorting approach implemented in the Project.

Input size	runtime (in nano seconds)
100	208600
1000	5899700
10000	73268700
100000	3.85E+09



Optimization: Optimization to the selection sort is the bi-directional selection sort or the **Double selection sort** in each pass it finds the maximum and the minimum element which compares with corresponding maximum and the minimum element this method reduces the runtime by almost 25%.This is also called the **cocktail sort.**

Comparing with other sorting techniques

Implementation of selection sorting using linked is easier when compared to other sorting methods. Among the quadratic sorting algorithms it stays above bubble sort but is less efficient when compared to insertion sort . The number of comparisons stays identical for selection sort but varies with the order of the array for insertion sort so it is less efficient than insertion sort.

Bubble Sort

Bubble sorting technique is a simple sorting method which compares the adjacent elements and swaps them if they are not in order. The above step is repeated until the entire array is sorted. Similar to insertion and selection sorts this sorting technique is inefficient for large data sets.

Bubble sort is a stable ,in-place algorithm as it compares only the adjacent elements for sorting the array.

Implementation of Bubble sort

Bubble sort implementation use arrays as data structures. The input to the arrays is given by a random number generator function-**Random()** with values ranging from 0-1000. The length of the array is decided interactively by the user.

- **void bubble_sort(int []array):** This method sorts the array by comparing the adjacent elements and swaps them if there in the wrong order starting from the beginning of the list.

Time complexity :

Best case: O(n^2)

Average case: O(n^2)

Worst case: O(n^2)

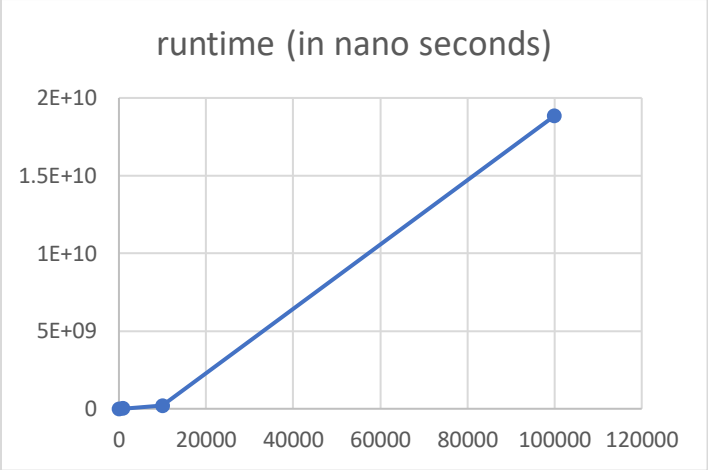
Space Complexity: O(1)

where n represents the length of the input array.

The elapsed time of the entire Bubble sorting is calculated by using the function .nanoTime() at the start of the process and at the end of the process calculating the difference gives you the total time taken for sorting the input array.

The below chart and table represent the Input sizes vs Run time for the bubble sorting approach implemented in the Project.

Input size	runtime (in nano seconds)
100	422600
1000	8087300
10000	190898200
100000	1.885E+10



Optimization: Optimization to the bubble sort is just ignoring the comparison after the last iteration as it need not to be checked again because the only elements will already be sorted so there is no need of that comparison which results in the worst case improvement by 50%.

Comparing with other sorting algorithms

It has the best case complexity of O(n) in case of already sorted array. Using the optimized approach of bubble sort the already sorted array input can be identified in the first pass itself. Bubble sort is the simplest sorting technique that can be implemented. It can be considered as one of the least efficient sorting algorithms keeping in mind the number of comparisons and the swaps required for sorting.

Methods for swapping and printing the array

- **void swap(int array[], int i, int j):** This function creates a temporary variable and swaps the positions of the elements whenever called by any function.

- **void print_array(int array[]):** This function is used to print the array it consists a for loop to print the entire array. All the sorted and unsorted arrays are printed using this function.

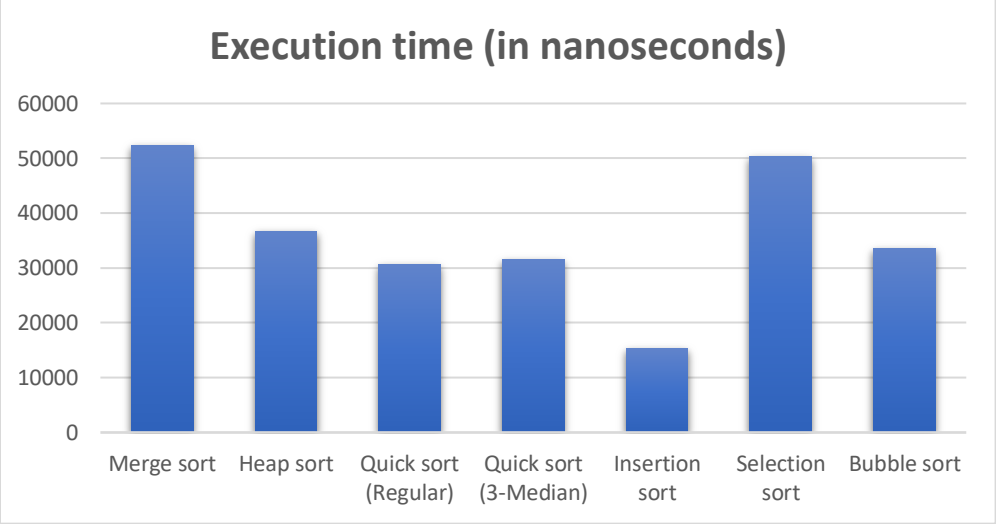
Comparing all sorting algorithms

Sorting techniques	Best case	Average case	Worst case	Space Complexity	Stable
Merge sort	O(nLogn)	O(nLogn)	O(nLogn)	O(n)	Yes
Heap sort	O(nLogn)	O(nLogn)	O(nLogn)	O(1)	No
Quick sort	O(nLogn)	O(nLogn)	O(n^2)	O(Logn)	No
Insertion sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Selection sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No
Bubble sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes

Comparing all the sorting algorithms with same input

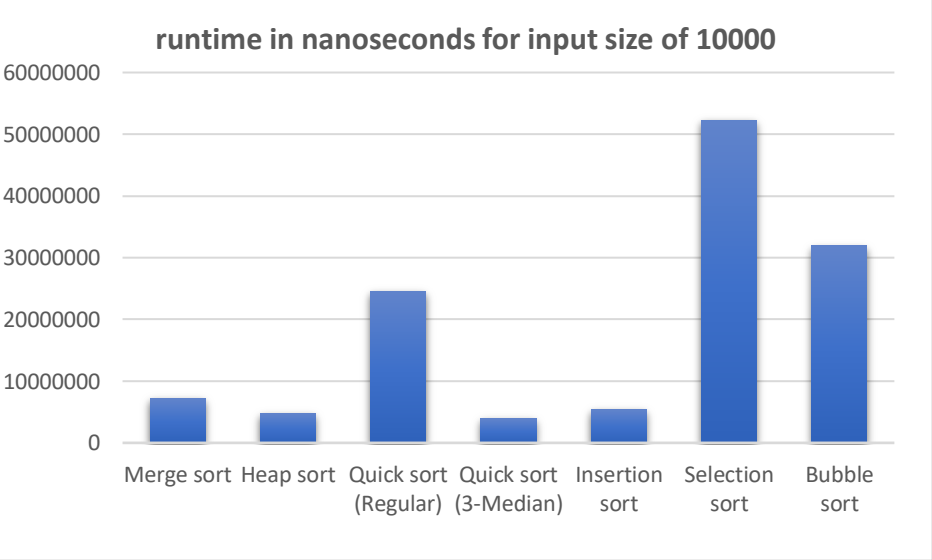
For this analysis, the input array for each algorithm will be the same and it is included in the read me file. The below graph and table show the execution time taken by each sorting array for the same input array of size 35 included in the INPUT file.

Sorting Technique	Execution time (in nanoseconds)
Merge sort	52400
Heap sort	36700
Quick sort (Regular)	30700
Quick sort (3-Median)	31500
Insertion sort	15400
Selection sort	50400
Bubble sort	33600



For this analysis, the input array for each algorithm will be the same and it is included in the read me file. The below graph and table show the execution time taken by each sorting array for the same input array of size 10000

Sorting Technique	Execution time (in nanoseconds)
Merge sort	7221300
Heap sort	4803800
Quick sort (Regular)	24519300
Quick sort (3-Median)	3968000
Insertion sort	540800
Selection sort	52260400
Bubble sort	31902600



Contributions

Chandra Shekhar Kasturi UTA ID: 1001825454.

Our team consists of two members. My significant contribution is being able to analyze and draw effective comparisons between the sorting techniques. We have divided the entire project into two significant halves which includes coding the algorithms and preparing the project report in each half. My contributions include the coding of merge sort, quick sort-regular method , heap sort and the major part of the project report which includes determining the efficiency of our algorithms, drawing comparisons between them and detailed specification of each algorithm implemented by both of us .My analysis includes line, bar graphs of input sizes vs execution time and a table differentiating each sorting algorithms. The detailed description includes the functions implemented in each sorting algorithm and data structures involved.

YESHWANTH TEJA DITY : 1001823070

Our team consists of two members. We divided project into two halves. My contribution is to analyze, code and compare the sorting techniques like quick sort 3-median, bubble sort, insertion sort, selection sort and preparing my half project report. Description includes the functions, methods included in sorting techniques and data structure involved.I have also given the entire description of the project In the readme file which helps to understand the functions and methods involved in the project and steps to execute the project. Execution time and analysis of graphs of each of sorting techniques of my part is described.