EECE7205: Fundamental of Computer Engineering
Homework 4

Ankesh Kumar (002208893)

Problem 1

TODO We analyzed the Selection, Bubble, Insertion, Merge, Heap, and Quick sort algorithms.

Need write a C++ program with the following requirements:

Analyze the algorithms by sorting, in ascending order, arrays of 2000 integers. You must implement these algorithms without using any C++ related libraries except the <random> library to generate random numbers

For part 1, all the algorithm was collected from the slides presented in class.
For Insertion Sort and Merge Sort, It was implemented as part of previous assignments HW1 and HW2 respectively, and others such as Selection Sort, Bubble Sort, Heap Sort and Quick Sort algorithm was developed based on class lecture slides.
Equation

```
void SelectionSort(int* A, int size);
void BubbleSort(int* A, int size);
void InsertionSort(int* A, int size);
void mergeSort(int* A, int left, int right);
void HeapSort(int* A, int size);
void QuickSort(int* A, int p, int r);
```

TODO Implement other checks to check efficiency of the algorithms.

Implemented 3 different array (Worst, Best and Random array)

In this, as asked, the Best (BST) was iterated from 1 to 2000 and was multipled by 10 to get as asked sort array.
For Average (AVG) case, as mentioned, used rand() function to generate the array.
For Worst (WST) case, iterated from 2000 to 1 to generate a complete unsorted array.

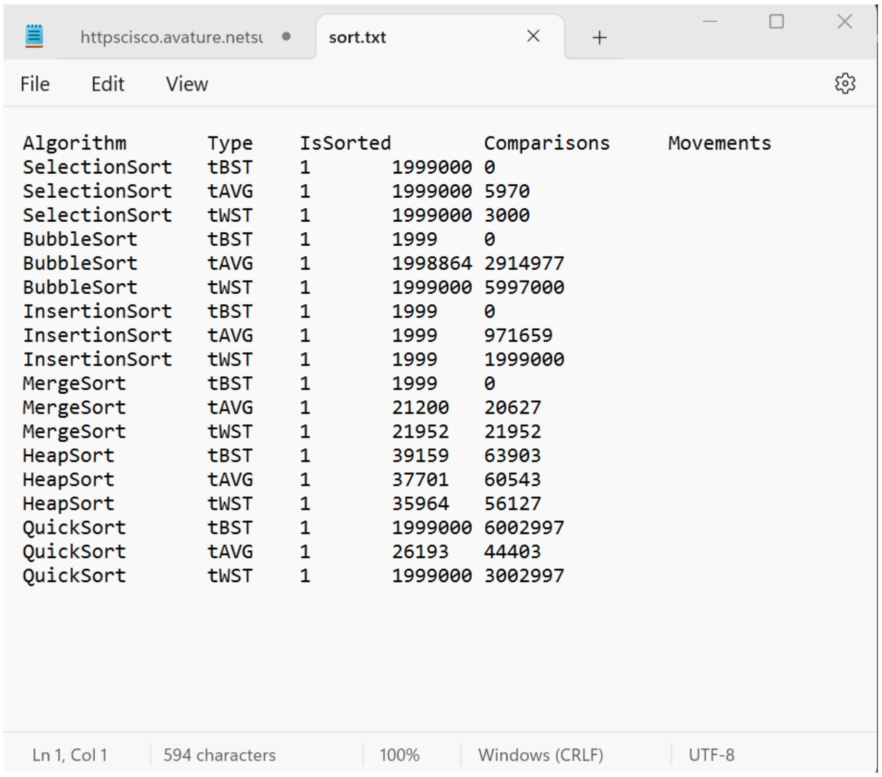
Tested all algorithms with the array created

Defined two global variable : moves and comps as per the question, and how to output into the output file - sort.txt

Created a function for verifying that the array produced is sorted.

Equation `bool checkSortedArray(const int arr[], int n);`

It generates a check isSorted as 1 as array is sorted and 0 if array is not sorted according to the asked in question.



```
Algorithm       Type    IsSorted        Comparisons     Movements
SelectionSort   tBST    1       1999000 0
SelectionSort   tAVG    1       1999000 5970
SelectionSort   tWST    1       1999000 3000
BubbleSort      tBST    1       1999    0
BubbleSort      tAVG    1       1998864 2914977
BubbleSort      tWST    1       1999000 5997000
InsertionSort   tBST    1       1999    0
InsertionSort   tAVG    1       1999    971659
InsertionSort   tWST    1       1999    1999000
MergeSort       tBST    1       1999    0
MergeSort       tAVG    1       21200   20627
MergeSort       tWST    1       21952   21952
HeapSort        tBST    1       39159   63903
HeapSort        tAVG    1       37701   60543
HeapSort        tWST    1       35964   56127
QuickSort       tBST    1       1999000 6002997
QuickSort       tAVG    1       26193   44403
QuickSort       tWST    1       1999000 3002997
```

Figure 1: Sort text data generated from program

Implemetation of the graphs and table generated

Table generated by pivoting after exporting into the excel :

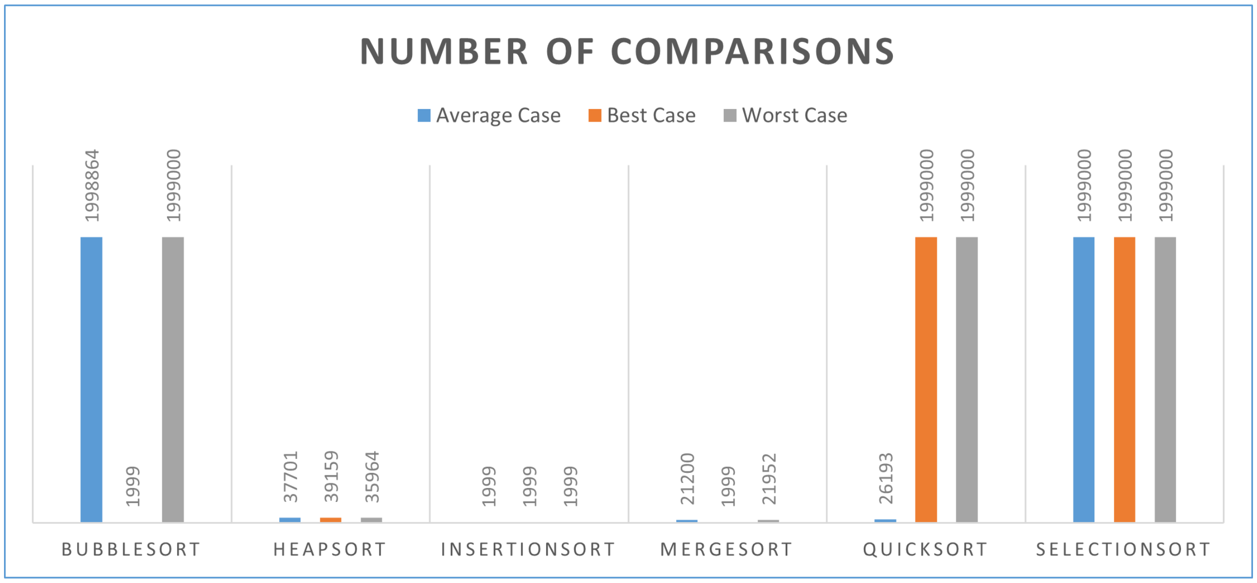
	Average Case	**Best Case**	**Worst Case**
BubbleSort	1998864	1999	1999000
HeapSort	37701	39159	35964
InsertionSort	1999	1999	1999
MergeSort	21200	1999	21952
QuickSort	26193	1999000	1999000
SelectionSort	1999000	1999000	1999000

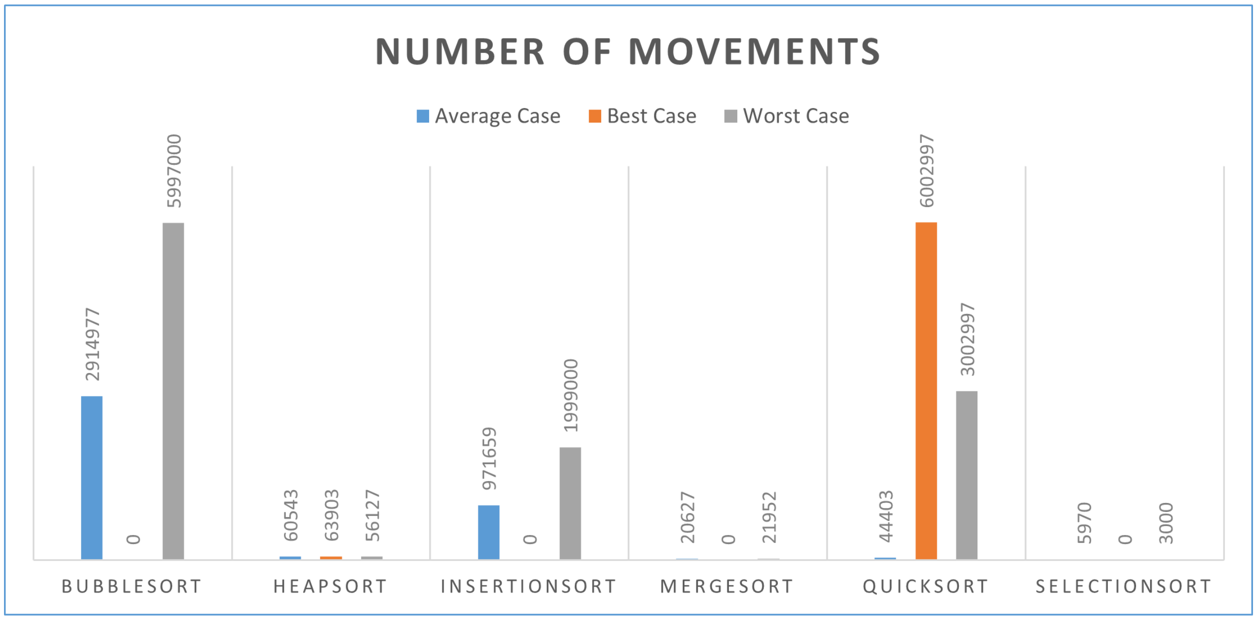
Table 1: Algorithm Comparison Table

	Average Case	**Best Case**	**Worst Case**
BubbleSort	2914977	0	5997000
HeapSort	60543	63903	56127
InsertionSort	971659	0	1999000
MergeSort	20627	0	21952
QuickSort	44403	6002997	3002997
SelectionSort	5970	0	3000

Table 2: Algorithm Movement Table

Graphs from the data to show trends:





Ques 1. *Why do the Selection, Bubble, and Insertion sort algorithms result in zero moves when sorting the BST array?*

Summarizing on the all algorithms used, each has its own growth rate. For BST array, the Selection, Bubble, and Insertion sort algorithms
Selection Sort:
- Finds the minimum element within the unsorted partition.
- Swaps the minimum element with the leftmost unsorted one.
- Repeats the process, moving the unsorted partition to the right.

Bubble Sort: As the algorithm works in the mentioned way
- Compares adjacent elements.
- Swaps if the elements are in the incorrect order.
- Repeats the comparison and swap process until the entire list is sorted.

Insertion Sort involves gradually building the sorted array by comparing new elements with already sorted ones and inserting them in the correct positions.

When given a sorted array, the elements are already positioned accurately, eliminating the need for any movements. In the text file, the "Movements" column for the BST case indicates zero movements, which aligns with the behavior of these algorithms when provided with already sorted lists. Specifically, zero movements mean that the algorithms didn't need to rearrange any elements to sort the already sorted array.

Ques 2. *Why do the Bubble and Insertion sort algorithms result in 1999 comparisons when sorting the BST array?*

As Bubble Sort, in BEST case where array is already sorted, it will perform one complete check without any swap to maintain the already sorted situation. For an array of 2000, it will check for 1999 times on first iteration and hence we get 1999 as output.
In case of the Insertion Sort, to determine the proper place for the current element, the algorithm compares each element with the items in the sorted subarray on its left. Since the array is already sorted, it just has to make one comparison for each additional element after the first to confirm that it is greater than the previous element. So, for size n array, it will compare for n-1 times and hence we get 1999 as final value.

Ques 3. *Why does the Selection sort algorithm result in 3000 moves when sorting the WST array?*

The Selection sort works in this process -
It looks for the minimum value in the unsorted array
Then it will swap with the first element of the unsorted array.
So, talking about how we have created the array: It starts from [2000, 1999, 1997 .... 3,2,1]. And as each swap function will count 3 moves as total operation, when in the first iteration, 1001 is compared with 1000, we swap the value and swap happens. Consecutively, it will keep swapping the outer loops and only values of 1 (Considering i where it lies from 1 to 1000) need to be swapped with its reverse (i.e. 2000 - i) and it is 1000 times comparison and then swapping. And hence we get the sorted array after 3000 moves (i.e. 1000 times comparison) as the last move would be checking the 1 and 2000 and them swapping the values.

Ques 4. *Why does the Merge sort algorithm result in the same number of moves to sort the three arrays?*

`Note` `In the developed algorithm for Merge Sort, we check if the loop is already sorted.`
I applied check that if the loop Merge() function checks that the loop is already sorted, we would not running the whole merge algorithm and for Best(BST) case it, returns 0 movement.
Explaining the behaviour, Merge Sort algorithm is "Divide and conquer" where we divide the array and then sort it.
It recursively divides the array into subarrays till size 1 is reached and then compare and sorts it if required.
So, for same size of array, it will approximately do same number of calculations.
But in my algorithm, as if the array is sorted, it doesn't compare and hence we increased the **Comparison** value but not **moves** value, it comes out to be zero. But for WST and AVG cases, roughly the movements remains the same.

Ques 5. *How are the heap sort moves affected when sorting the BST array compared to sorting the WST array? Explain the reason.*

From the data generated, we can see
Heap Sort on BST array: Comparisons: 39159 Movements: 63903
Heap Sort on WST array: Comparisons: 35964 Movements: 56127

This behavior could be based on how the array is defined.
As BST array has largest element (20000) at end (i.e. index 2000) and the WST array has largest element (2000) as index 1.
Hence, during the heapify and max-heapify process, there are more number of moves required in BST array compared to the WST array.
Heap Sort has time complexity of *O(nlogn)* for all cases but the number of moves was based on the arrangement of the array and as we need to create a binary tree out of the count is not much intutive to show a trend of specific behaviour.

Ques 6. *Based on the two excel graphs, how does each algorithm perform under different scenarios (best, average, and worst)?*

I first would like to make a note of all the time complexity data I collected related to the 6 different sorting algorithms which was asked in this assignment.

Algorithm	**Best Case Time Complexity**	**Worst Case Time Complexity**
Bubble Sort	$O(n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$
Insertion Sort	$O(n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$

Table 3: Best and worst case time complexities of sorting algorithms.

Taking on the behaviour shown by two different graphs demonstrated -

Bubble sort shows a dependable pattern in many scenarios; in the best case, it shows no movements, indicating an optimized situation in which the list has already been sorted. On the other hand, in the worst and average circumstances, it experiences a significant rise in both movements and comparisons, indicating its inefficiency with unsorted data.

Heap sort handles unsorted data better than Bubble sort, but it still performs inefficiently in complex scenarios. In the best case, it requires comparatively few moves, while in the average and worst circumstances, it rises up.

Insertion sort exhibits remarkable performance in the best case similar to Bubble sort where there are zero movements; however, in the average and worst instances, it also experiences a large increase in complexity.

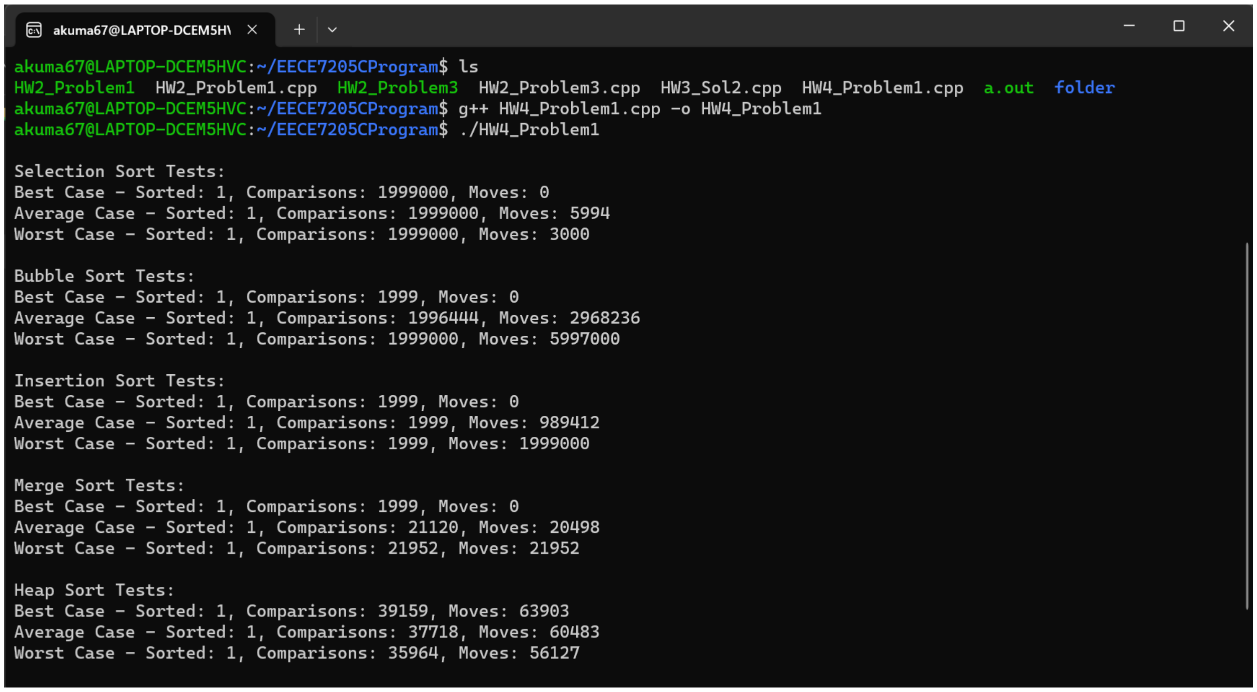
Merge sort performs well in an array of settings, as evidenced by its exceptional consistency, which includes zero changes in the best case and a slight rise in the average and worst instances.

With zero movements *(Merge algorithm was modified to not move if already sorted)* in the best case and a their is good performance/ increase in the average and worst cases, **Merge sort** demonstrates consistency across each type of array AVG, BST, WST.

In the best case, **Quick sort** exhibits a mixed performance with a large number of movements, which is unusual and suggests a specific implementation or scenario; however, it scales up less dramatically, suggesting better average performance even with the higher initial cost.

Selection sort exhibits minimal movement in the best case, but similarly to **Bubble sort**, it experiences significant increases in the average and worst cases, suggesting poor performance when dealing with unsorted data.

Execution Screenshot



```
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$ ls
HW2_Problem1  HW2_Problem1.cpp  HW2_Problem3  HW2_Problem3.cpp  HW3_Sol2.cpp  HW4_Problem1.cpp  a.out  folder
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$ g++ HW4_Problem1.cpp -o HW4_Problem1
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$ ./HW4_Problem1

Selection Sort Tests:
Best Case - Sorted: 1, Comparisons: 1999000, Moves: 0
Average Case - Sorted: 1, Comparisons: 1999000, Moves: 5994
Worst Case - Sorted: 1, Comparisons: 1999000, Moves: 3000

Bubble Sort Tests:
Best Case - Sorted: 1, Comparisons: 1999, Moves: 0
Average Case - Sorted: 1, Comparisons: 1996444, Moves: 2968236
Worst Case - Sorted: 1, Comparisons: 1999000, Moves: 5997000

Insertion Sort Tests:
Best Case - Sorted: 1, Comparisons: 1999, Moves: 0
Average Case - Sorted: 1, Comparisons: 1999, Moves: 989412
Worst Case - Sorted: 1, Comparisons: 1999, Moves: 1999000

Merge Sort Tests:
Best Case - Sorted: 1, Comparisons: 1999, Moves: 0
Average Case - Sorted: 1, Comparisons: 21120, Moves: 20498
Worst Case - Sorted: 1, Comparisons: 21952, Moves: 21952

Heap Sort Tests:
Best Case - Sorted: 1, Comparisons: 39159, Moves: 63903
Average Case - Sorted: 1, Comparisons: 37718, Moves: 60483
Worst Case - Sorted: 1, Comparisons: 35964, Moves: 56127
```