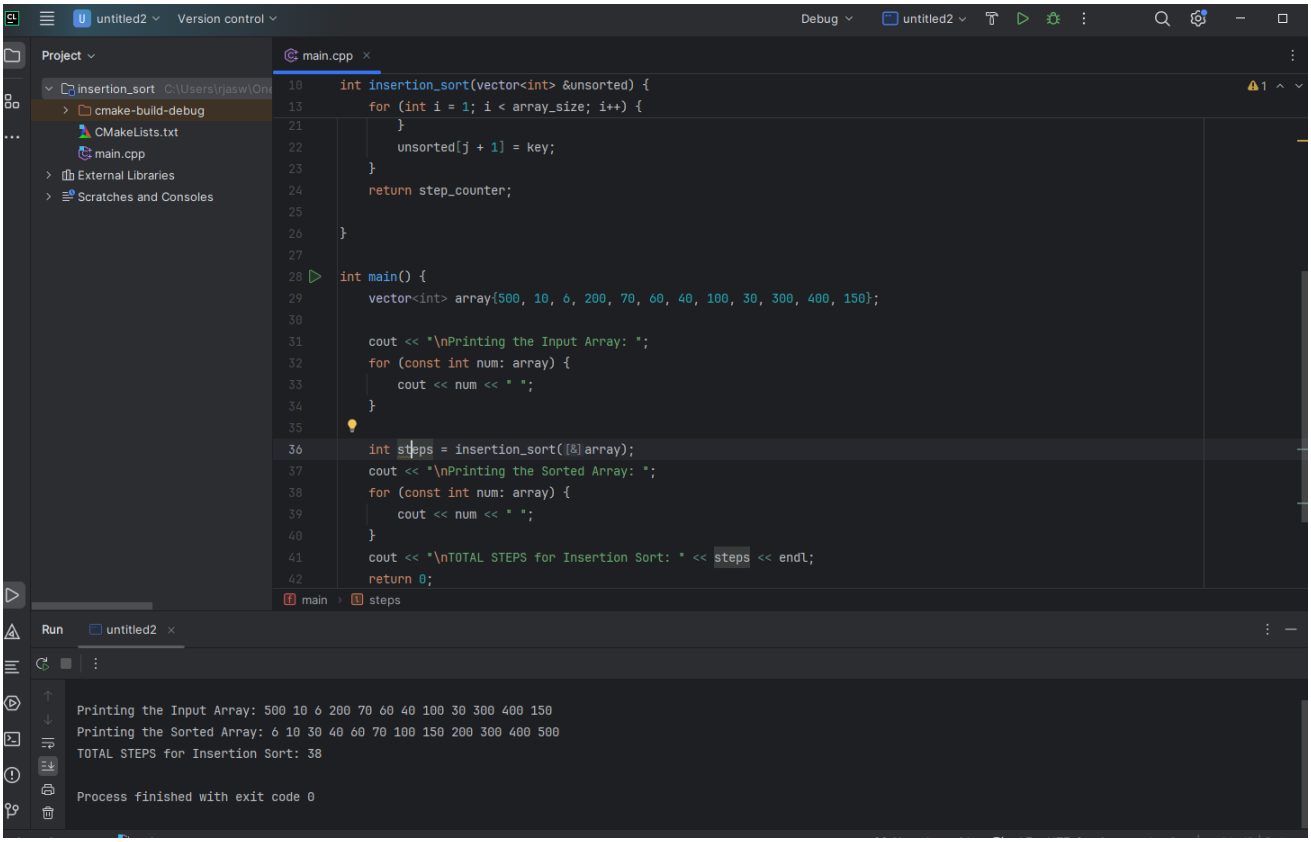
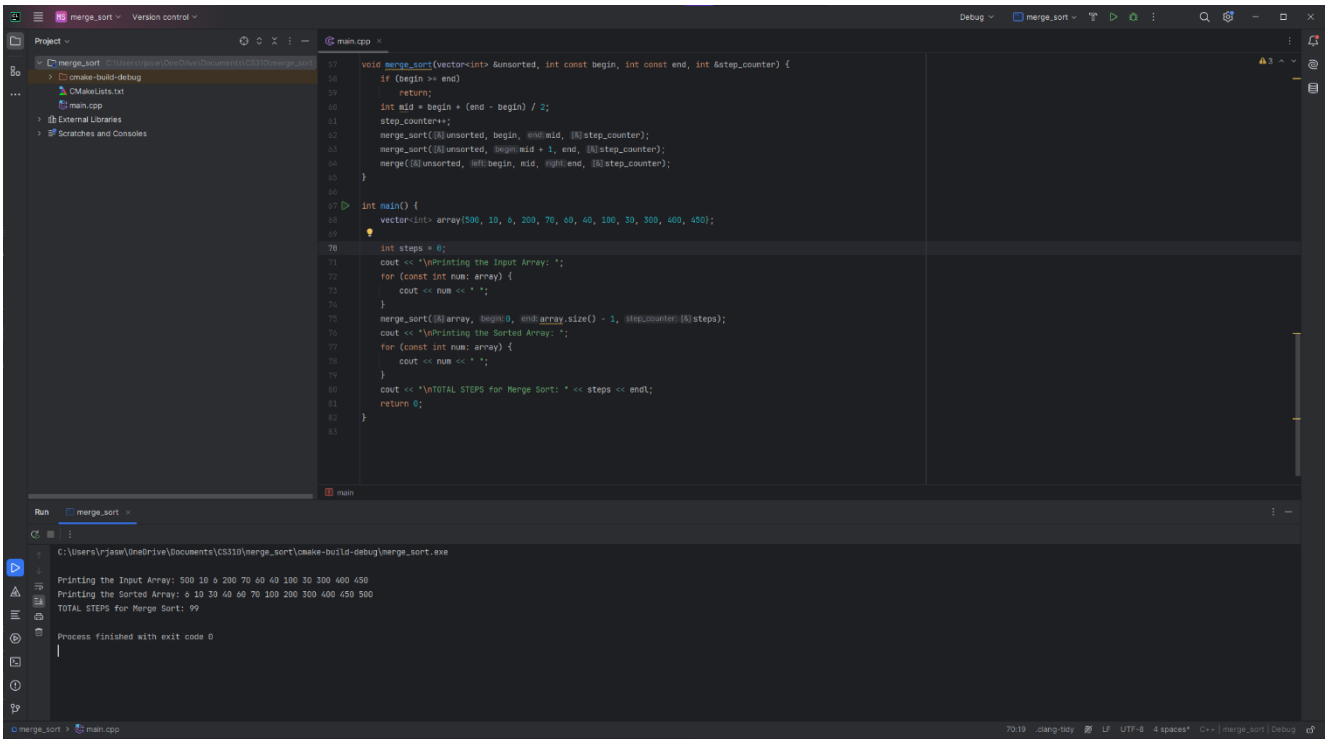
Deliverable 1

Sample Input & Output of Each Sorting Algorithm

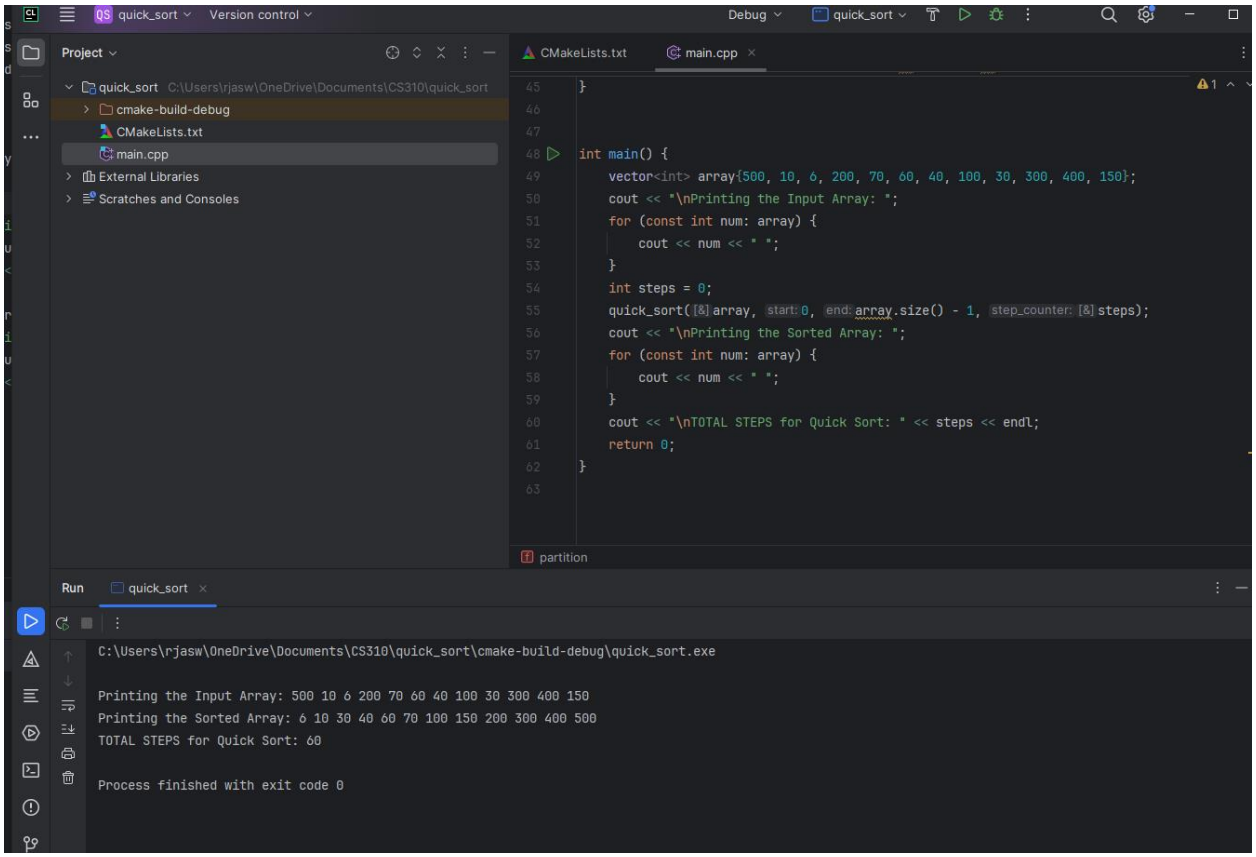
1. Insertion Sort



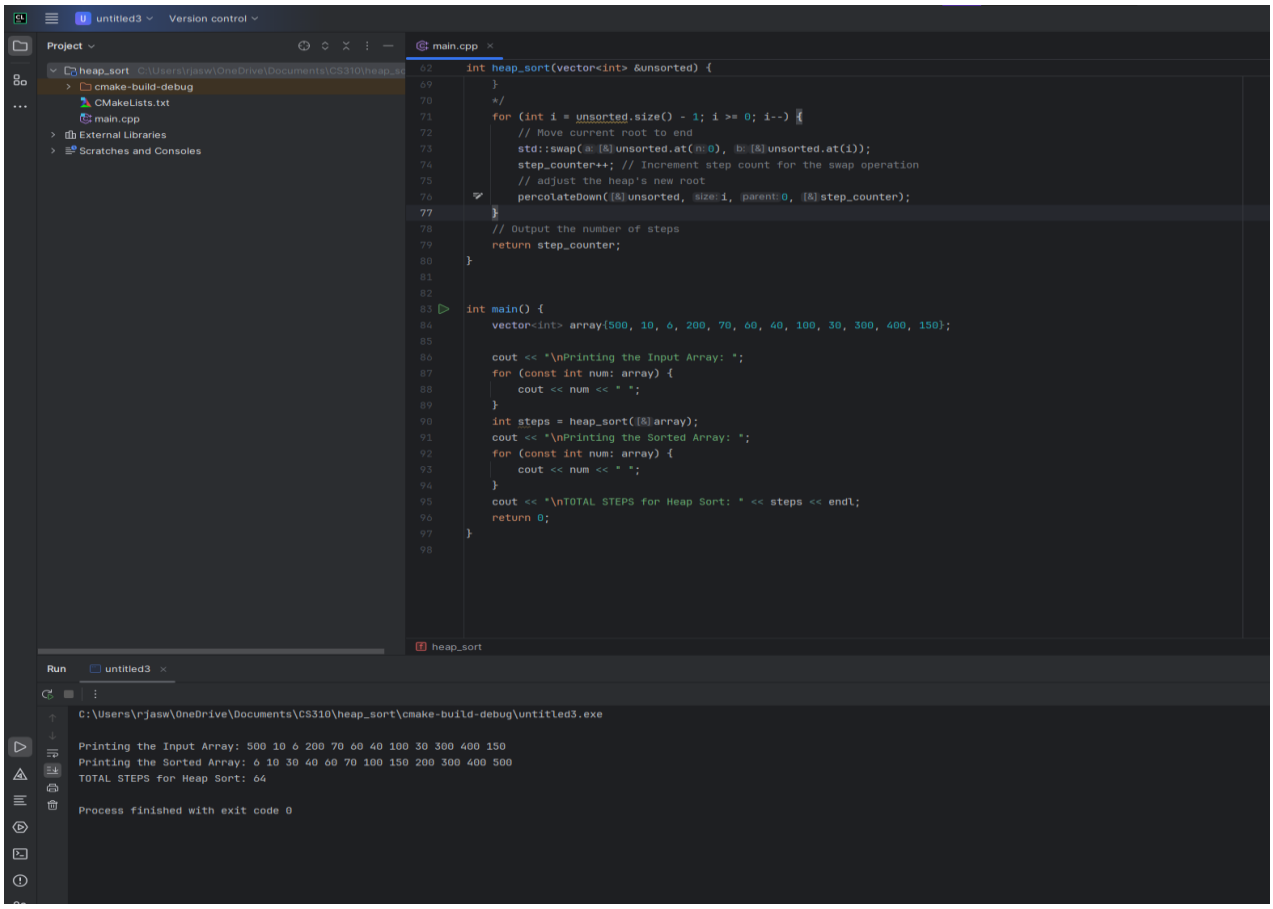
2. Merge Sort



3. Quick Sort

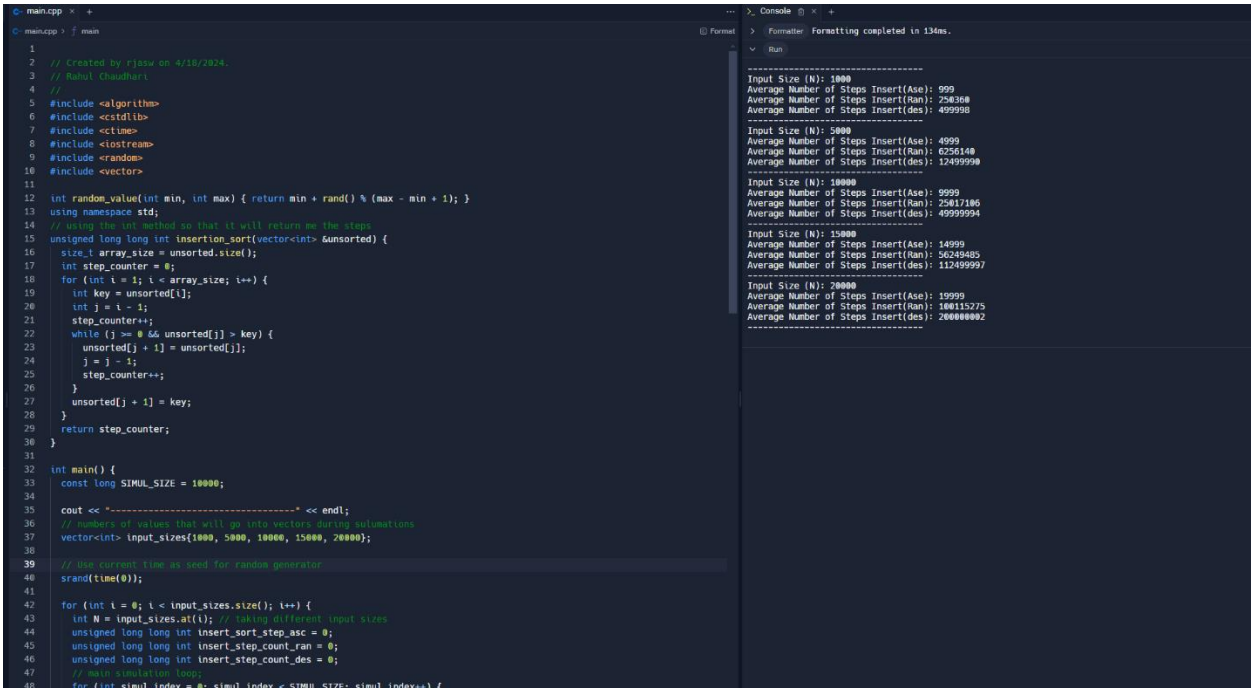


4. Heap Sort



Deliverable 2

1. Simulation of Insert Sort & Tabulated Data with Simulation Size of 10,000.
   (Note: Used online console compiler for faster processing, as it used their server CPU rather than laptop CPU. Website used: replit.com, can be used for any language projects)



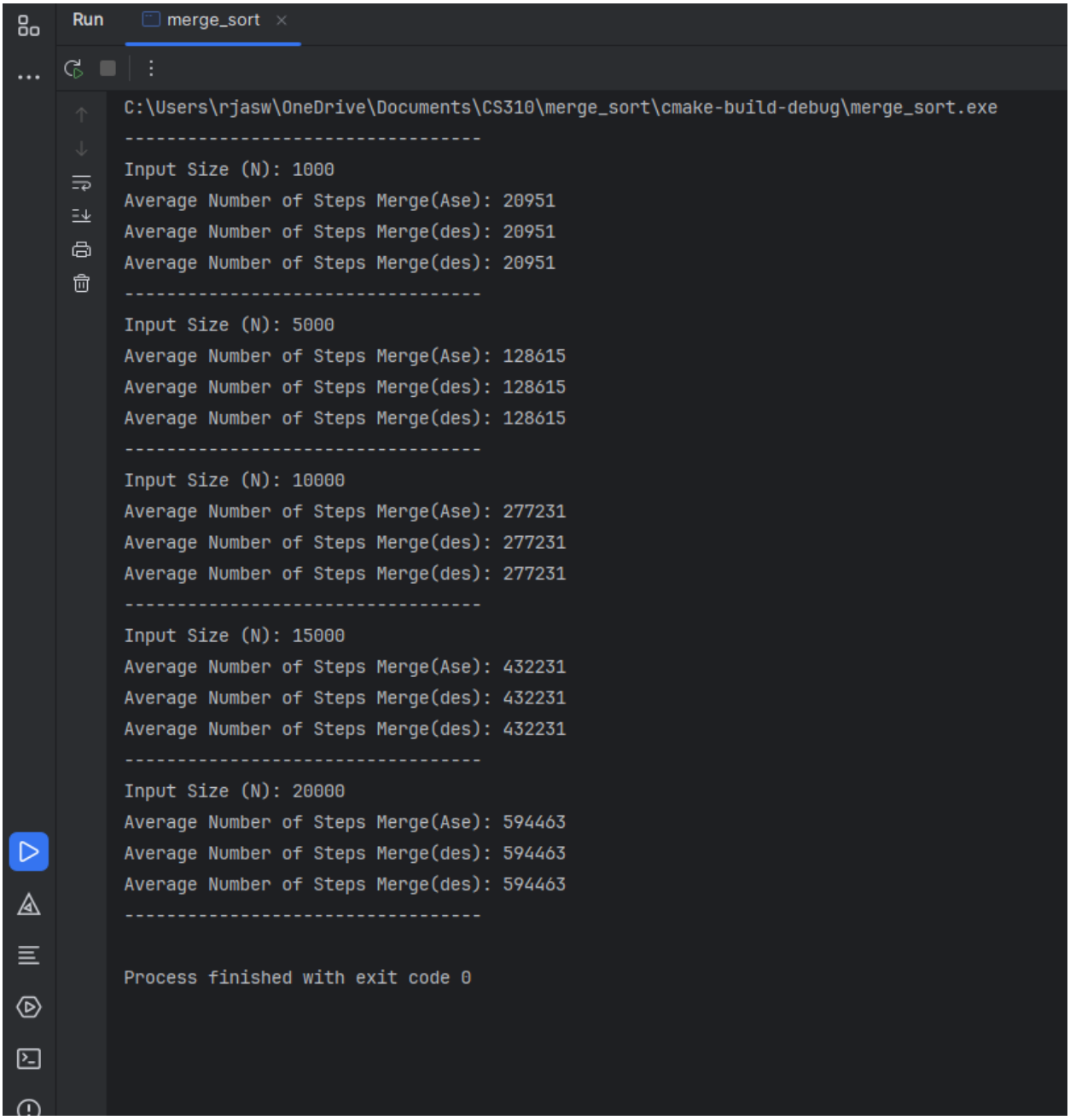
```cpp
// Created by rjasw on 4/18/2024.
// Rahul Chaudhari
//
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <random>
#include <vector>

int random_value(int min, int max) { return min + rand() % (max - min + 1); }
using namespace std;
// using the int method so that it will return me the steps
unsigned long long int insertion_sort(vector<int> &unsorted) {
  size_t array_size = unsorted.size();
  int step_counter = 0;
  for (int i = 1; i < array_size; i++) {
    int key = unsorted[i];
    int j = i - 1;
    step_counter++;
    while (j >= 0 && unsorted[j] > key) {
      unsorted[j + 1] = unsorted[j];
      j = j - 1;
      step_counter++;
    }
    unsorted[j + 1] = key;
  }
  return step_counter;
}

int main() {
  const long SIMUL_SIZE = 10000;

  cout << "-----------------------------------" << endl;
  // numbers of values that will go into vectors during sulumations
  vector<int> input_sizes{1000, 5000, 10000, 15000, 20000};

  // Use current time as seed for random generator
  srand(time(0));

  for (int i = 0; i < input_sizes.size(); i++) {
    int N = input_sizes.at(i); // taking different input sizes
    unsigned long long int insert_sort_step_asc = 0;
    unsigned long long int insert_step_count_ran = 0;
    unsigned long long int insert_step_count_des = 0;
    // main simulation loop;
```

```
Formatter: Formatting completed in 134ms.
Run
-----------------------------------
Input Size (N): 1000
Average Number of Steps Insert(Ase): 999
Average Number of Steps Insert(Ran): 250360
Average Number of Steps Insert(des): 499998
-----------------------------------
Input Size (N): 5000
Average Number of Steps Insert(Ase): 4999
Average Number of Steps Insert(Ran): 6256140
Average Number of Steps Insert(des): 12499990
-----------------------------------
Input Size (N): 10000
Average Number of Steps Insert(Ase): 9999
Average Number of Steps Insert(Ran): 25017106
Average Number of Steps Insert(des): 49999994
-----------------------------------
Input Size (N): 15000
Average Number of Steps Insert(Ase): 14999
Average Number of Steps Insert(Ran): 56249485
Average Number of Steps Insert(des): 112499997
-----------------------------------
Input Size (N): 20000
Average Number of Steps Insert(Ase): 19999
Average Number of Steps Insert(Ran): 100115275
Average Number of Steps Insert(des): 200000002
-----------------------------------
```

Insertion Sort				
Input Size	Elements already sorted	elements sorted in descending order	random values	average steps
1K	999	499998	250360	250452.3333
5K	4999	12499990	6256140	6253709.667
10K	9999	49999994	25017106	25009033
15K	14999	112499997	56249485	56254827
20K	19999	200000002	100115275	100045092

Note: Best for Already Sorted Array.

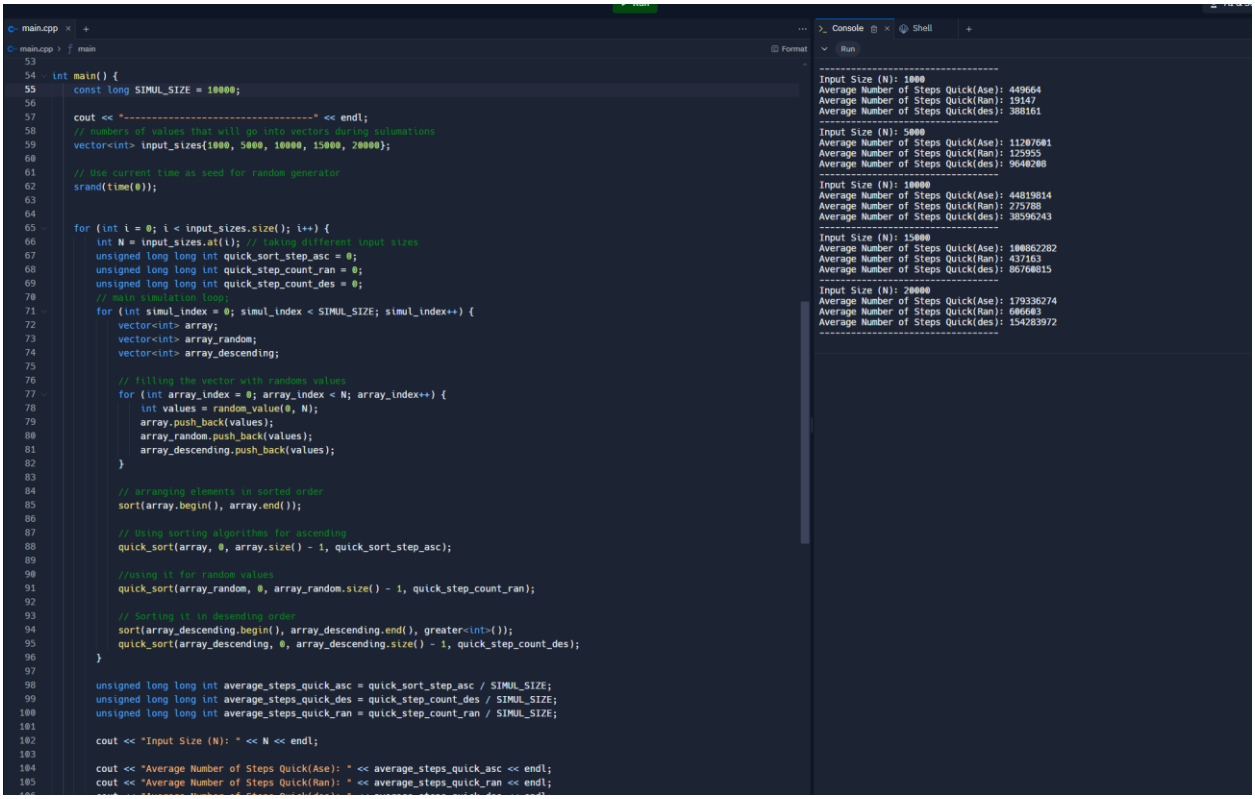
2. Simulation of Merge Sort & Tabulated Data with Simulation Size of 10,000.
(Note this is run on Clion IDE)



Merge Sort				
Input Size	Elements already sorted	elements sorted in descending order	random values	average steps
1K	20951	20951	20951	20951
5K	128615	128615	128615	128615
10K	277321	277321	277321	277321
15K	432231	43231	43231	43231
20K	594463	594463	594463	594463

Note: Same for All Cases

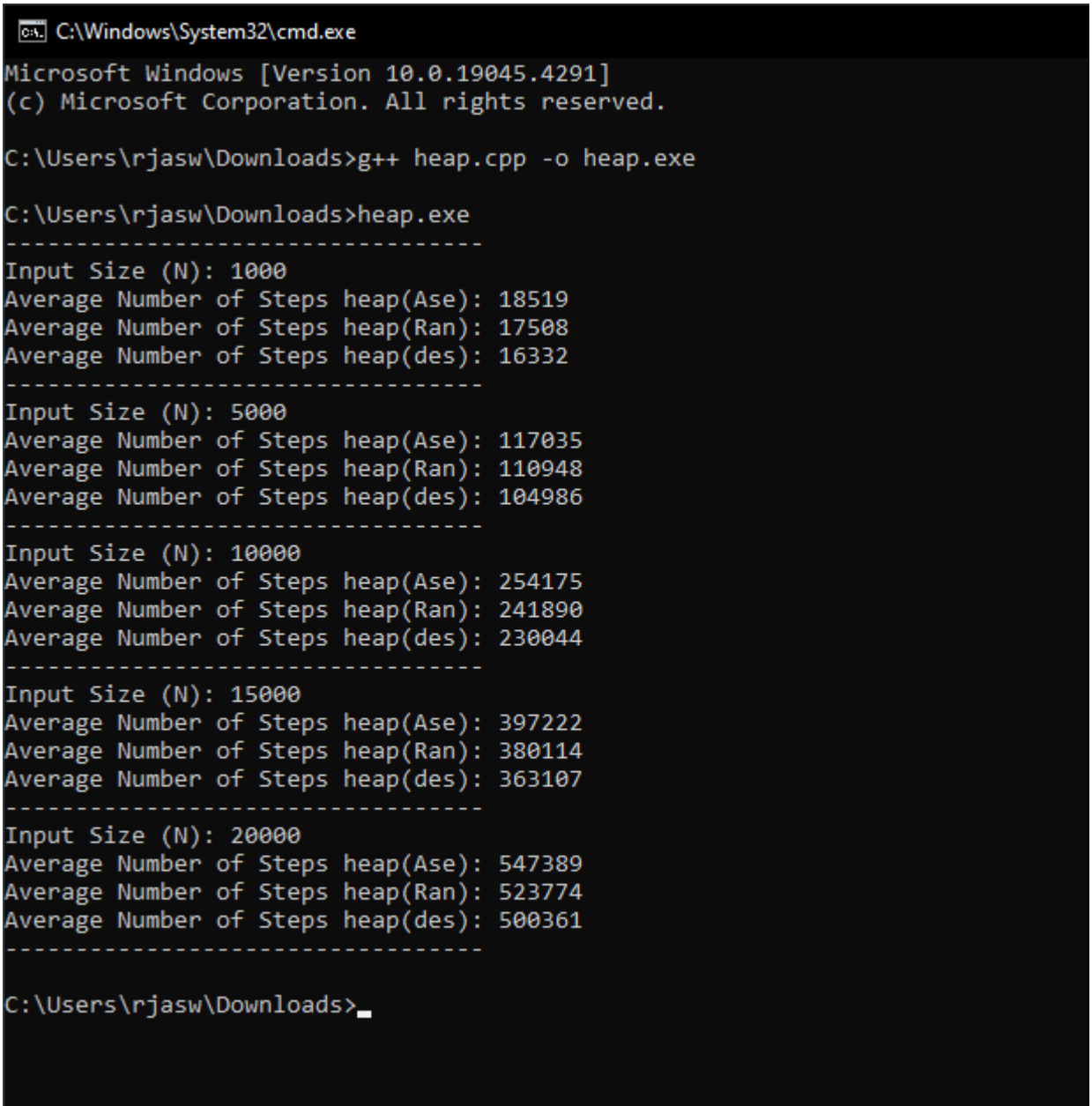
3. Simulation of Quick Sort & Tabulated Data with Simulation Size of 10,000.



Quick Sort				
Input Size	Elements already sorted	elements sorted in descending order	random values	average steps
1K	449664	388161	19147	285657.3333
5K	11207601	9640208	125955	6991254.667
10K	44819814	38596243	275788	27897281.67
15K	100862282	86760815	437163	62686753.33
20K	179336274	154283972	606603	111408949.7

Note: Good for Sorting Random Values

4. Simulation of Heap Sort & Tabulated Data with Simulation Size of 10,000.
   (Note: Made .exe file to run it using lower resources rather than using Clion)

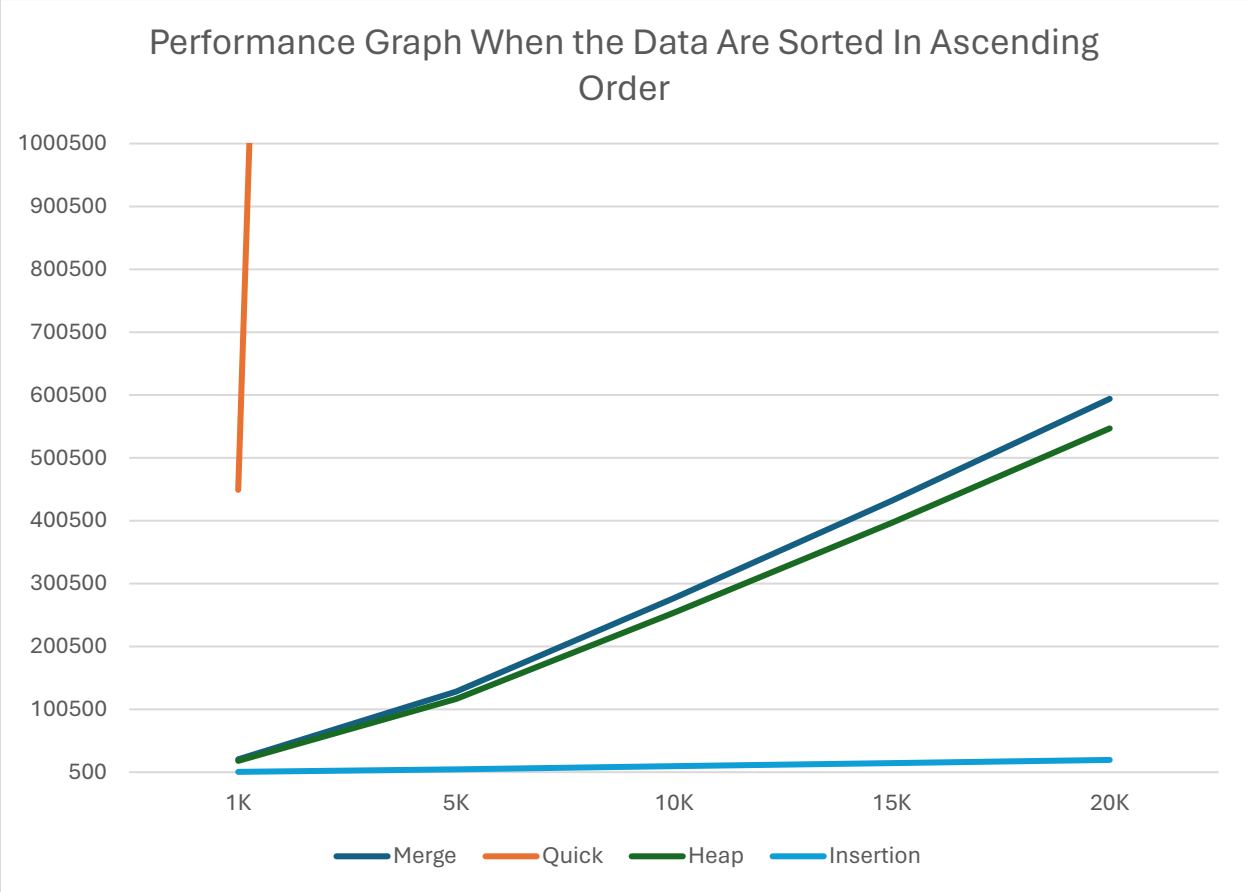


Heap Sort				
Input Size	Elements already sorted	elements sorted in descending order	random values	average steps
1K	18519	16332	17508	17453
5K	117035	104986	110948	110989.6667
10K	254175	230044	241890	242036.3333
15K	397222	363107	380114	380147.6667
20K	547389	500361	523774	523841.3333

Note: Best for Array in descending Order.

Deliverable 3

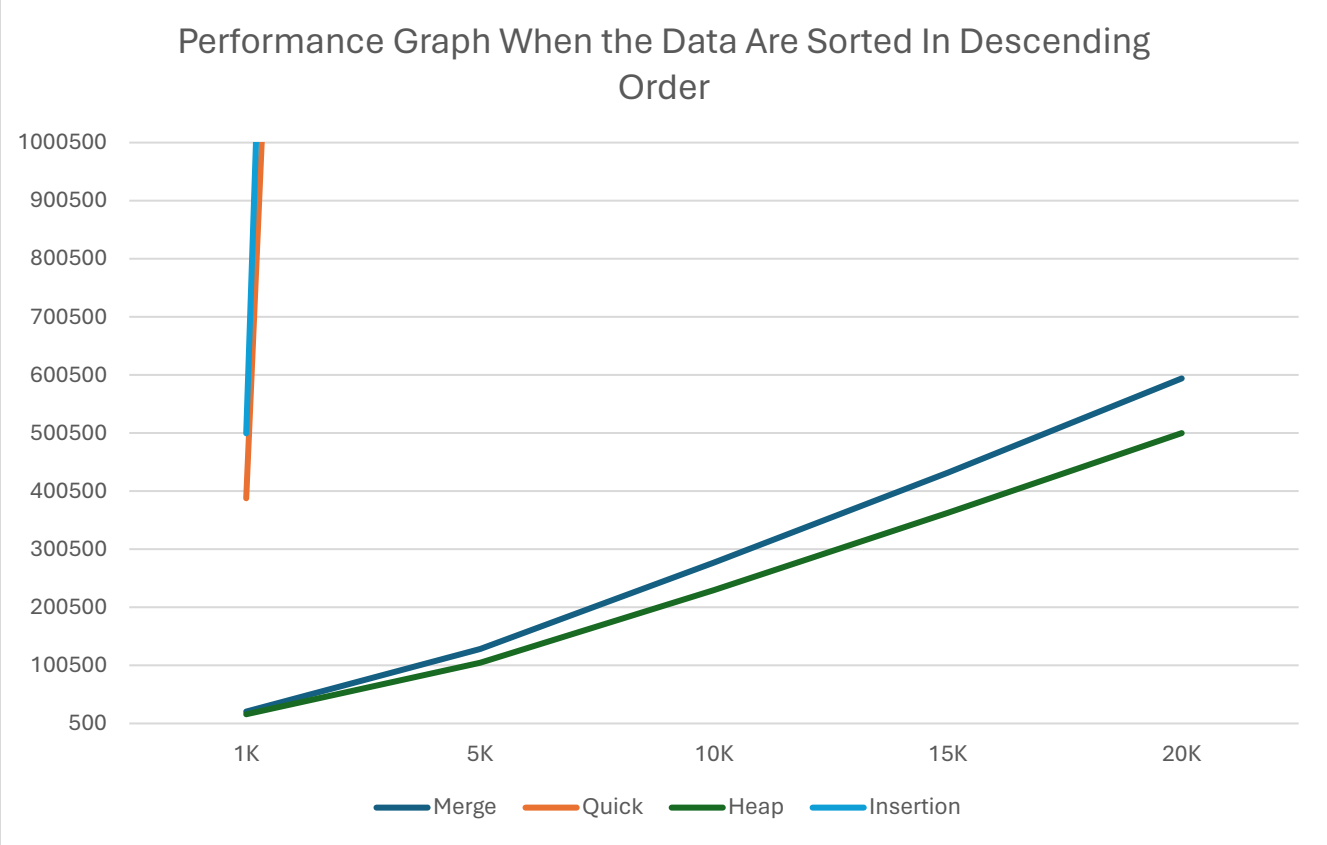
a. **Plot a graph showing the performance of each algorithm when the elements are already sorted. Determine which sorting algorithm's performance is the worst? What is the reason for this performance?**



Worst Performing Algorithm: Quick Sort

Reason: Quick Sort typically performs well on random data because it partitions the data efficiently. However, when the data is already sorted, the pivot selection process can lead to imbalanced partitions, resulting in unnecessary comparisons and swaps. In the worst case, Quick Sort can have a complexity of O(n^2) for sorted arrays.

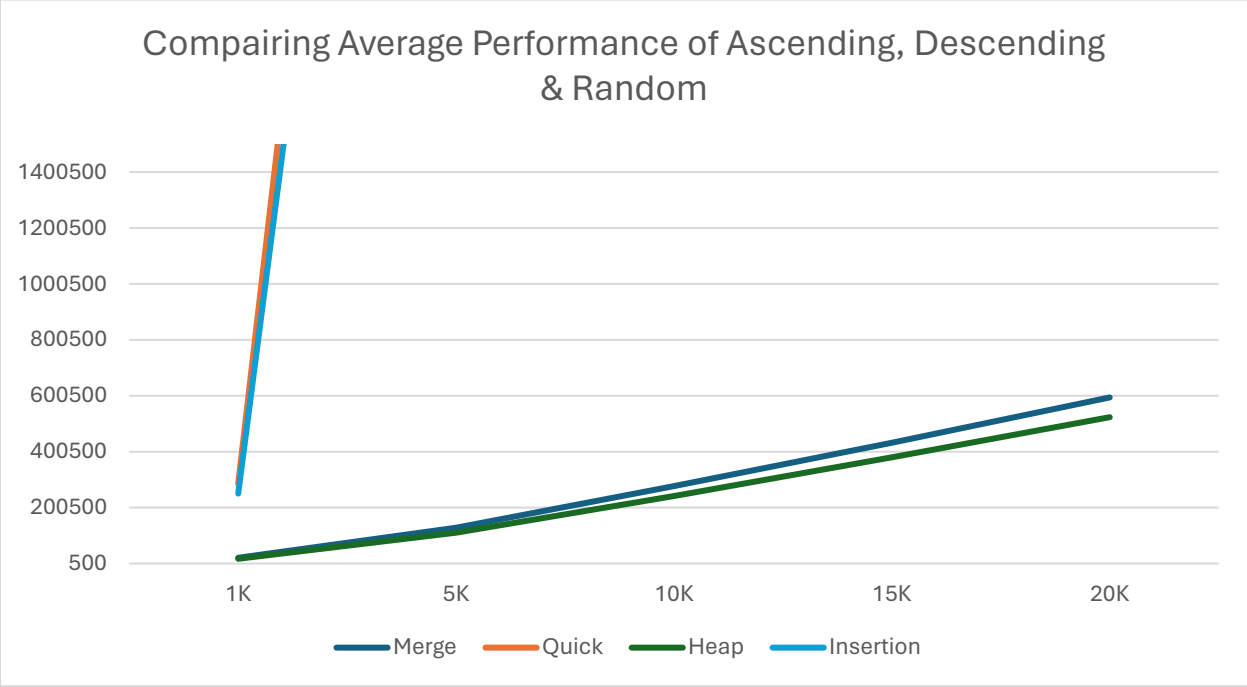
b. Plot a graph showing the performance of each algorithm when the elements are sorted in descending order. Determine which sorting algorithm's performance is the worst? What is the reason for this performance?



Worst Performing Algorithm: Insertion Sort (Side Note: Quick Sort is also kind of same and near)

Reason: Insertion Sort works by iterating through the list and inserting each element in its correct position among the previously sorted elements. When the data is sorted in descending order (opposite of its desired outcome), Insertion Sort needs to shift almost all elements one position to the right to insert each new element at the beginning. This shifting requires multiple comparisons and swaps for each element, leading to a quadratic time complexity of O(n^2) in the worst case for descending sorted data.

c. Considering all the tabular data that you have gathered for the four algorithms, which sorting algorithm's performance on average is better than the others? Why?



Considering the tabular data, Heap Sort emerges as the sorting algorithm with the best average performance. Its consistency across different scenarios, including already sorted, descending order, and random data, showcases its robustness. While Heap Sort may not always have the fastest performance with random data compared to Quick Sort, it still maintains a competitive edge in terms of average performance. Additionally, Heap Sort's worst-case time complexity of O (n log n) ensures that its performance remains reasonable even in the worst-case scenario.

d. Considering all the tabular data that you have gathered for the four algorithms, which sorting algorithm's performance on average is the worse than the others? Why? (Referring in the Graph from C)

Quick Sort emerges as the sorting algorithm with the worst average performance among the four algorithms analyzed based on the tabular data. While Quick Sort typically exhibits excellent average and best-case performance with random data, its average steps increase substantially when the data is already sorted or sorted in descending order (worst-case time complexity of O(n^2)). This degradation in performance is due to Quick Sort's reliance on pivot selection and partitioning, which can lead to unbalanced partitions and inefficient sorting in these scenarios.