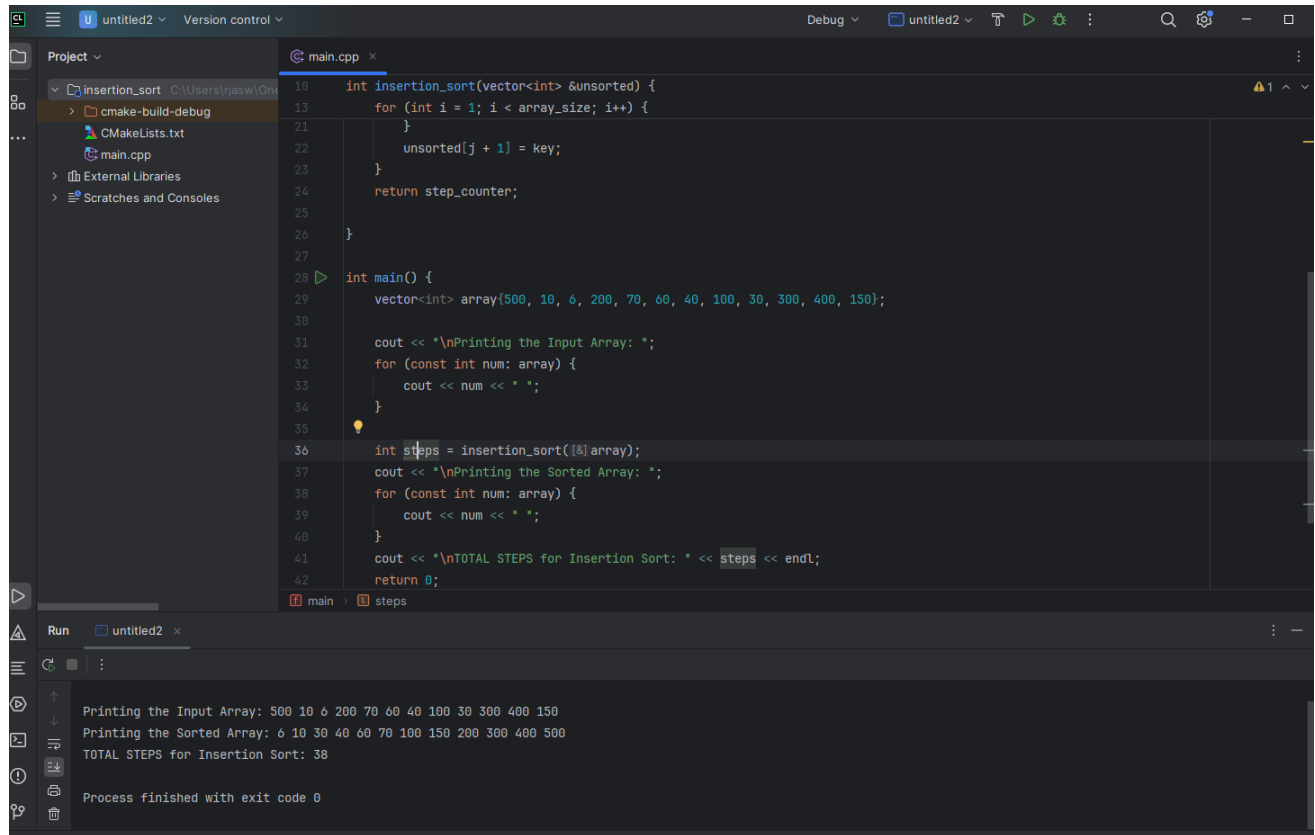


Deliverable 1

Sample Input & Output of Each Sorting Algorithm

1. Insertion Sort



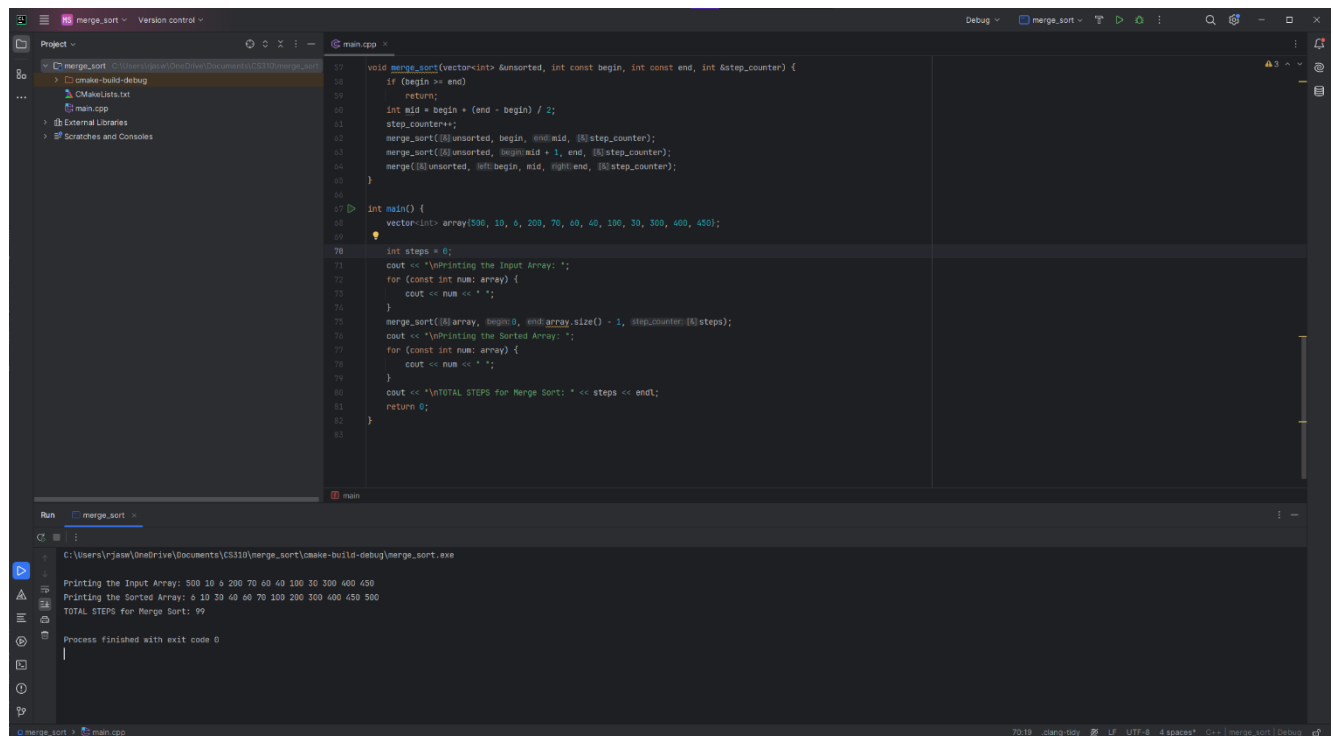
The screenshot shows a C++ IDE with a project named 'insertion_sort'. The code in 'main.cpp' defines an 'insertion_sort' function that takes a vector of integers and returns the number of steps. The 'main' function initializes an array of 15 numbers, prints it, calls 'insertion_sort', prints the sorted array, and outputs the total steps. The output window shows the input array, the sorted array, and the total steps for Insertion Sort: 38.

```
10 int insertion_sort(vector<int> &unsorted) {
11     for (int i = 1; i < array_size; i++) {
12     }
13     unsorted[j + 1] = key;
14 }
15 return step_counter;
16 }
17
18 int main() {
19     vector<int> array{500, 10, 6, 200, 70, 60, 40, 100, 30, 300, 400, 150};
20
21     cout << "\nPrinting the Input Array: ";
22     for (const int num: array) {
23         cout << num << " ";
24     }
25
26     int steps = insertion_sort([&]array);
27     cout << "\nPrinting the Sorted Array: ";
28     for (const int num: array) {
29         cout << num << " ";
30     }
31     cout << "\nTOTAL STEPS for Insertion Sort: " << steps << endl;
32     return 0;
33 }
```

Run untitled2

Printing the Input Array: 500 10 6 200 70 60 40 100 30 300 400 150
Printing the Sorted Array: 6 10 30 40 60 70 100 150 200 300 400 500
TOTAL STEPS for Insertion Sort: 38
Process finished with exit code 0

2. Merge Sort



The screenshot shows a C++ IDE with a project named 'merge_sort'. The code in 'main.cpp' defines a recursive 'merge_sort' function that takes a vector of integers and returns the number of steps. The 'main' function initializes an array of 15 numbers, prints it, calls 'merge_sort', prints the sorted array, and outputs the total steps. The output window shows the input array, the sorted array, and the total steps for Merge Sort: 99.

```
37 void merge_sort(vector<int> &unsorted, int const begin, int const end, int &step_counter) {
38     if (begin >= end)
39         return;
40     int mid = begin + (end - begin) / 2;
41     step_counter++;
42     merge_sort([&]unsorted, begin, mid, [&]step_counter);
43     merge_sort([&]unsorted, begin + 1, end, [&]step_counter);
44     merge([&]unsorted, [&]begin, mid, [&]end, [&]step_counter);
45 }
46
47 int main() {
48     vector<int> array{500, 10, 6, 200, 70, 60, 40, 100, 30, 300, 400, 150};
49
50     int steps = 0;
51     cout << "\nPrinting the Input Array: ";
52     for (const int num: array) {
53         cout << num << " ";
54     }
55     merge_sort([&]array, [&]begin, [&]end, [&]step_counter);
56     cout << "\nPrinting the Sorted Array: ";
57     for (const int num: array) {
58         cout << num << " ";
59     }
60     cout << "\nTOTAL STEPS for Merge Sort: " << steps << endl;
61     return 0;
62 }
```

Run merge_sort

C:\Users\rjase\OneDrive\Documents\CS310\merge_sort\cmake-build-debug\merge_sort.exe

Printing the Input Array: 500 10 6 200 70 60 40 100 30 300 400 150
Printing the Sorted Array: 6 10 30 40 60 70 100 200 300 400 450 500
TOTAL STEPS for Merge Sort: 99
Process finished with exit code 0

3. Quick Sort

The screenshot shows a Visual Studio Code editor with a project named 'quick_sort'. The file explorer on the left shows the project structure: 'quick_sort' (C:\Users\rjasw\OneDrive\Documents\CS310\quick_sort) containing 'cmake-build-debug', 'CMakeLists.txt', and 'main.cpp'. The 'main.cpp' file is open in the editor, showing the following code:

```

45 }
46
47
48 int main() {
49     vector<int> array{500, 10, 6, 200, 70, 60, 40, 100, 30, 300, 400, 150};
50     cout << "\nPrinting the Input Array: ";
51     for (const int num: array) {
52         cout << num << " ";
53     }
54     int steps = 0;
55     quick_sort(&array, start:0, end:array.size() - 1, step_counter: [&steps]);
56     cout << "\nPrinting the Sorted Array: ";
57     for (const int num: array) {
58         cout << num << " ";
59     }
60     cout << "\nTOTAL STEPS for Quick Sort: " << steps << endl;
61     return 0;
62 }
63

```

The Run and Debug console at the bottom shows the output of the program:

```

C:\Users\rjasw\OneDrive\Documents\CS310\quick_sort\cmake-build-debug\quick_sort.exe

Printing the Input Array: 500 10 6 200 70 60 40 100 30 300 400 150
Printing the Sorted Array: 6 10 30 40 60 70 100 150 200 300 400 500
TOTAL STEPS for Quick Sort: 60

Process finished with exit code 0

```

4. Heap Sort

The screenshot shows a Visual Studio Code editor with a project named 'heap_sort'. The file explorer on the left shows the project structure: 'heap_sort' (C:\Users\rjasw\OneDrive\Documents\CS310\heap_sort) containing 'cmake-build-debug', 'CMakeLists.txt', and 'main.cpp'. The 'main.cpp' file is open in the editor, showing the following code:

```

62 int heap_sort(vector<int> &unsorted) {
63     // Output the number of steps
64     return step_counter;
65 }
66
67
68 // Output the number of steps
69 return step_counter;
70
71
72 for (int i = unsorted.size() - 1; i >= 0; i--) {
73     // Move current root to end
74     std::swap(&unsorted.at(0), &unsorted.at(i));
75     step_counter++; // Increment step count for the swap operation
76     // adjust the heap's new root
77     percolateDown(&unsorted, size:1, parent:0, [&step_counter]);
78 }
79
80
81
82
83 int main() {
84     vector<int> array{500, 10, 6, 200, 70, 60, 40, 100, 30, 300, 400, 150};
85
86     cout << "\nPrinting the Input Array: ";
87     for (const int num: array) {
88         cout << num << " ";
89     }
90     int steps = heap_sort(&array);
91     cout << "\nPrinting the Sorted Array: ";
92     for (const int num: array) {
93         cout << num << " ";
94     }
95     cout << "\nTOTAL STEPS for Heap Sort: " << steps << endl;
96     return 0;
97 }
98

```

The Run and Debug console at the bottom shows the output of the program:

```

C:\Users\rjasw\OneDrive\Documents\CS310\heap_sort\cmake-build-debug\untitled3.exe

Printing the Input Array: 500 10 6 200 70 60 40 100 30 300 400 150
Printing the Sorted Array: 6 10 30 40 60 70 100 150 200 300 400 500
TOTAL STEPS for Heap Sort: 64

Process finished with exit code 0

```

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)

```

1 // Created by rjshaw on 4/10/2024.
2 // Rahul Chaudhari
3 //
4 #include <algorithm>
5 #include <cstdlib>
6 #include <ctime>
7 #include <iostream>
8 #include <random>
9 #include <vector>
10
11 int random_value(int min, int max) { return min + rand() % (max - min + 1); }
12 using namespace std;
13 // using the in-built sort that it will return the steps
14 unsigned long long int insertion_sort(vector<int> &unsorted) {
15     size_t array_size = unsorted.size();
16     int step_counter = 0;
17     for (int i = 1; i < array_size; i++) {
18         int key = unsorted[i];
19         int j = i - 1;
20         step_counter++;
21         while (j >= 0 && unsorted[j] > key) {
22             unsorted[j + 1] = unsorted[j];
23             j = j - 1;
24             step_counter++;
25         }
26         unsorted[j + 1] = key;
27     }
28     return step_counter;
29 }
30
31 int main() {
32     const long SIMUL_SIZE = 10000;
33
34     cout << "-----" << endl;
35     // numbers of values that will go into vectors during simulations
36     vector<int> input_sizes{1000, 5000, 10000, 15000, 20000};
37
38     // Use current time as seed for random generator
39     srand(time(0));
40
41     for (int i = 0; i < input_sizes.size(); i++) {
42         int N = input_sizes.at(i); // taking different input sizes
43         unsigned long long int insert_sort_step_asc = 0;
44         unsigned long long int insert_step_count_ran = 0;
45         unsigned long long int insert_step_count_des = 0;
46         // main simulation loop
47         for (int simul_index = 0; simul_index < SIMUL_SIZE; simul_index++) {

```

```

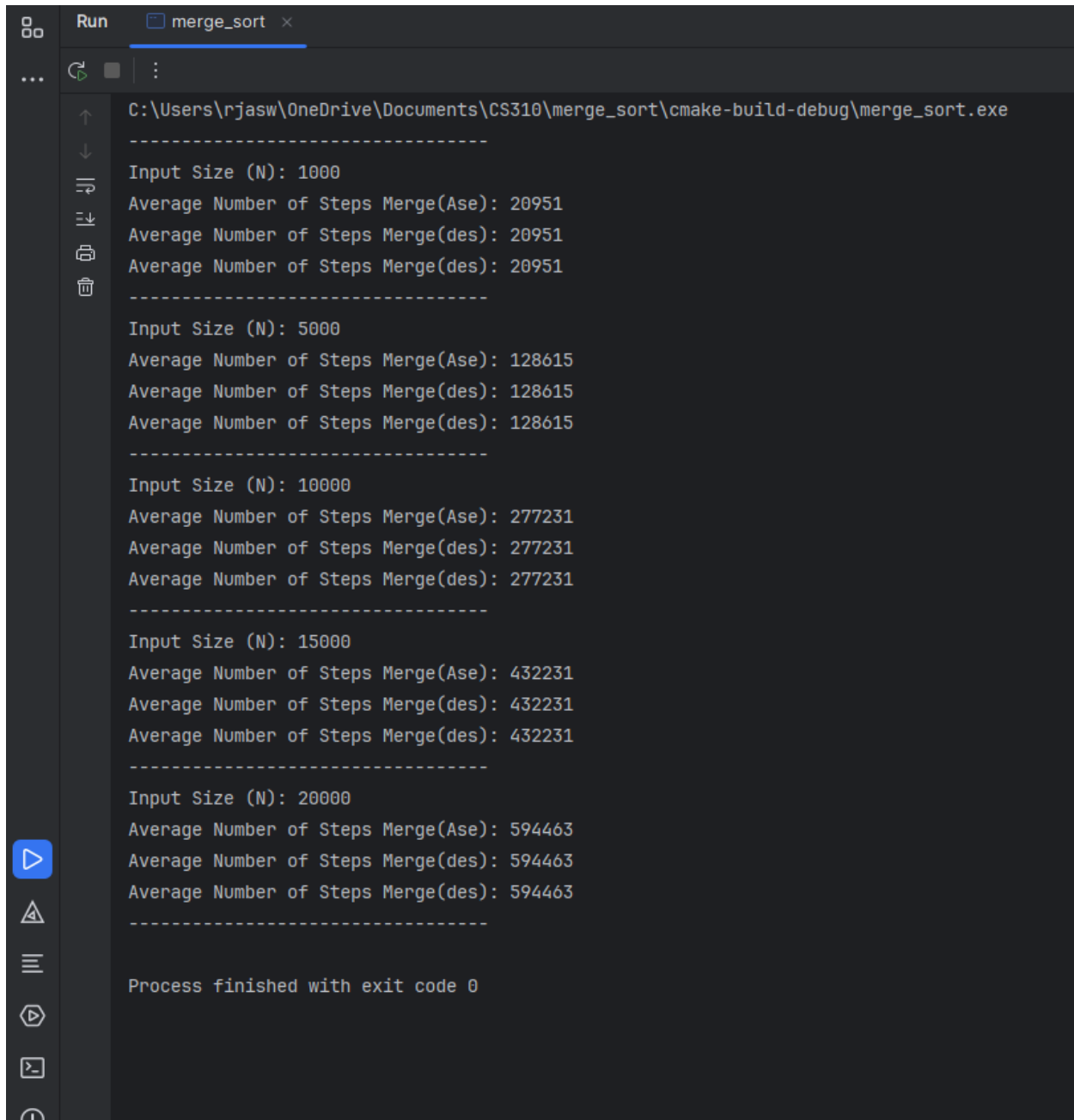
-----
Input Size (N): 1000
Average Number of Steps Insert(Asc): 999
Average Number of Steps Insert(Ran): 250360
Average Number of Steps Insert(Des): 499998
-----
Input Size (N): 5000
Average Number of Steps Insert(Asc): 4999
Average Number of Steps Insert(Ran): 6256140
Average Number of Steps Insert(Des): 12499990
-----
Input Size (N): 10000
Average Number of Steps Insert(Asc): 9999
Average Number of Steps Insert(Ran): 25017106
Average Number of Steps Insert(Des): 49999994
-----
Input Size (N): 15000
Average Number of Steps Insert(Asc): 14999
Average Number of Steps Insert(Ran): 56249485
Average Number of Steps Insert(Des): 112499997
-----
Input Size (N): 20000
Average Number of Steps Insert(Asc): 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)



```
Run merge_sort x
C:\Users\rjasw\OneDrive\Documents\CS310\merge_sort\cmake-build-debug\merge_sort.exe
-----
Input Size (N): 1000
Average Number of Steps Merge(Ase): 20951
Average Number of Steps Merge(des): 20951
Average Number of Steps Merge(des): 20951
-----
Input Size (N): 5000
Average Number of Steps Merge(Ase): 128615
Average Number of Steps Merge(des): 128615
Average Number of Steps Merge(des): 128615
-----
Input Size (N): 10000
Average Number of Steps Merge(Ase): 277231
Average Number of Steps Merge(des): 277231
Average Number of Steps Merge(des): 277231
-----
Input Size (N): 15000
Average Number of Steps Merge(Ase): 432231
Average Number of Steps Merge(des): 432231
Average Number of Steps Merge(des): 432231
-----
Input Size (N): 20000
Average Number of Steps Merge(Ase): 594463
Average Number of Steps Merge(des): 594463
Average Number of Steps Merge(des): 594463
-----
Process finished with exit code 0
```

| 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.

```

1  main.cpp x +
2  C: main.cpp > f main
3
4  int main() {
5      const long SIMUL_SIZE = 10000;
6
7      cout << "-----" << endl;
8      // numbers of values that will go into vectors during simulations
9      vector<int> input_sizes(1000, 5000, 10000, 15000, 20000);
10
11      // use current time as seed for random generator
12      srand(time(0));
13
14      for (int i = 0; i < input_sizes.size(); i++) {
15          int N = input_sizes.at(i); // taking different input sizes
16          unsigned long long int quick_sort_step_asc = 0;
17          unsigned long long int quick_step_count_ran = 0;
18          unsigned long long int quick_step_count_des = 0;
19          // main simulation loop
20          for (int simul_index = 0; simul_index < SIMUL_SIZE; simul_index++) {
21              vector<int> array;
22              vector<int> array_random;
23              vector<int> array_descending;
24
25              // filling the vector with random values
26              for (int array_index = 0; array_index < N; array_index++) {
27                  int values = random_value(0, N);
28                  array.push_back(values);
29                  array_random.push_back(values);
30                  array_descending.push_back(values);
31              }
32
33              // reversing elements in sorted order
34              sort(array.begin(), array.end());
35
36              // using sorting algorithm for ascending
37              quick_sort(array, 0, array.size() - 1, quick_sort_step_asc);
38
39              // using it for random values
40              quick_sort(array_random, 0, array_random.size() - 1, quick_step_count_ran);
41
42              // sorting it in descending order
43              sort(array_descending.begin(), array_descending.end(), greater<int>());
44              quick_sort(array_descending, 0, array_descending.size() - 1, quick_step_count_des);
45          }
46
47          unsigned long long int average_steps_quick_asc = quick_sort_step_asc / SIMUL_SIZE;
48          unsigned long long int average_steps_quick_des = quick_step_count_des / SIMUL_SIZE;
49          unsigned long long int average_steps_quick_ran = quick_step_count_ran / SIMUL_SIZE;
50
51          cout << "Input Size (N): " << N << endl;
52
53          cout << "Average Number of Steps Quick(Asc): " << average_steps_quick_asc << endl;
54          cout << "Average Number of Steps Quick(Ran): " << average_steps_quick_ran << endl;
55      }
56  }

```

```

Input Size (N): 1000
Average Number of Steps Quick(Asc): 449664
Average Number of Steps Quick(Ran): 19147
Average Number of Steps Quick(des): 308161
-----
Input Size (N): 5000
Average Number of Steps Quick(Asc): 11207081
Average Number of Steps Quick(Ran): 129955
Average Number of Steps Quick(des): 9648288
-----
Input Size (N): 10000
Average Number of Steps Quick(Asc): 44819814
Average Number of Steps Quick(Ran): 275788
Average Number of Steps Quick(des): 30590243
-----
Input Size (N): 15000
Average Number of Steps Quick(Asc): 108802282
Average Number of Steps Quick(Ran): 437163
Average Number of Steps Quick(des): 86768015
-----
Input Size (N): 20000
Average Number of Steps Quick(Asc): 17930674
Average Number of Steps Quick(Ran): 686983
Average Number of Steps Quick(des): 154283972
-----

```

| 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

- 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)

```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\rjasw\Downloads>g++ heap.cpp -o heap.exe

C:\Users\rjasw\Downloads>heap.exe
-----
Input Size (N): 1000
Average Number of Steps heap(Ase): 18519
Average Number of Steps heap(Ran): 17508
Average Number of Steps heap(des): 16332
-----
Input Size (N): 5000
Average Number of Steps heap(Ase): 117035
Average Number of Steps heap(Ran): 110948
Average Number of Steps heap(des): 104986
-----
Input Size (N): 10000
Average Number of Steps heap(Ase): 254175
Average Number of Steps heap(Ran): 241890
Average Number of Steps heap(des): 230044
-----
Input Size (N): 15000
Average Number of Steps heap(Ase): 397222
Average Number of Steps heap(Ran): 380114
Average Number of Steps heap(des): 363107
-----
Input Size (N): 20000
Average Number of Steps heap(Ase): 547389
Average Number of Steps heap(Ran): 523774
Average Number of Steps heap(des): 500361
-----

C:\Users\rjasw\Downloads>_

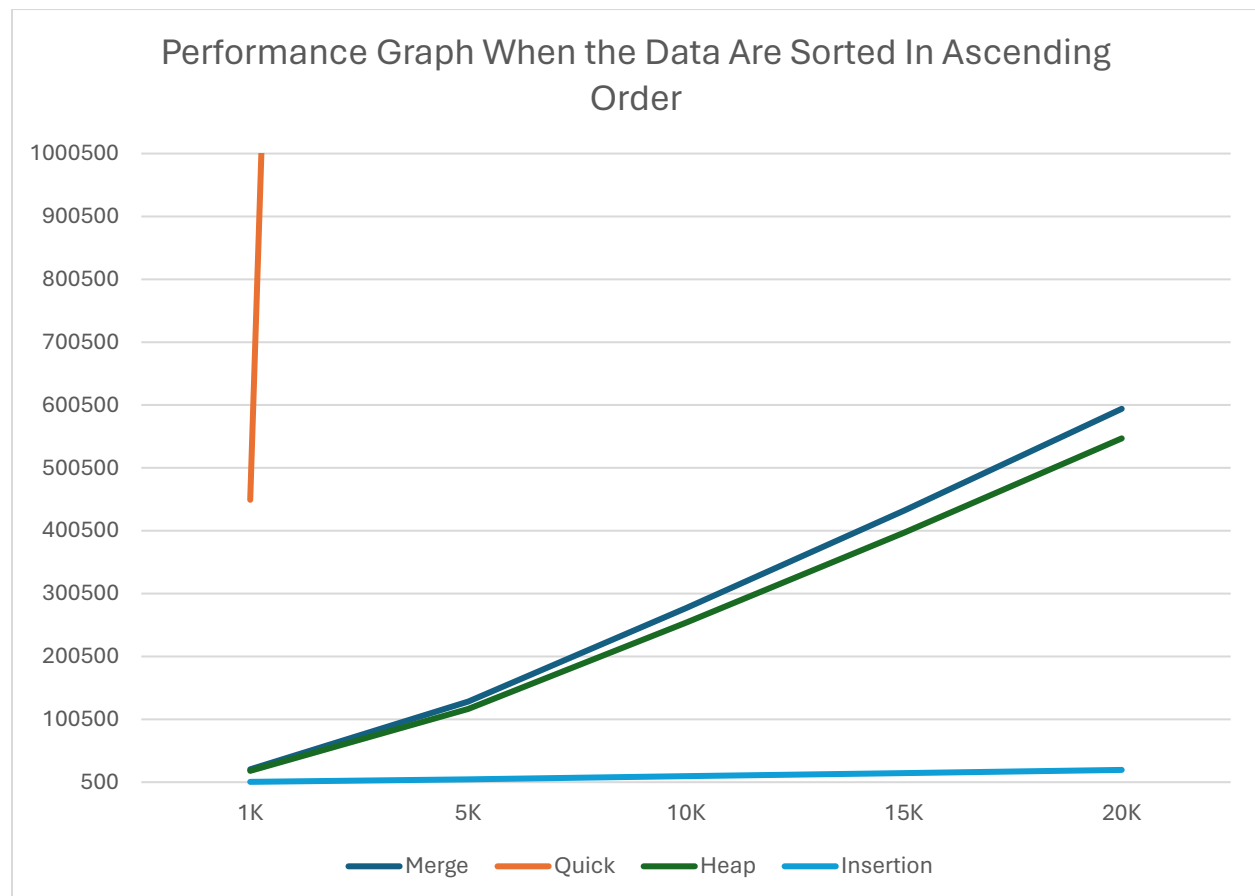
```

| 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

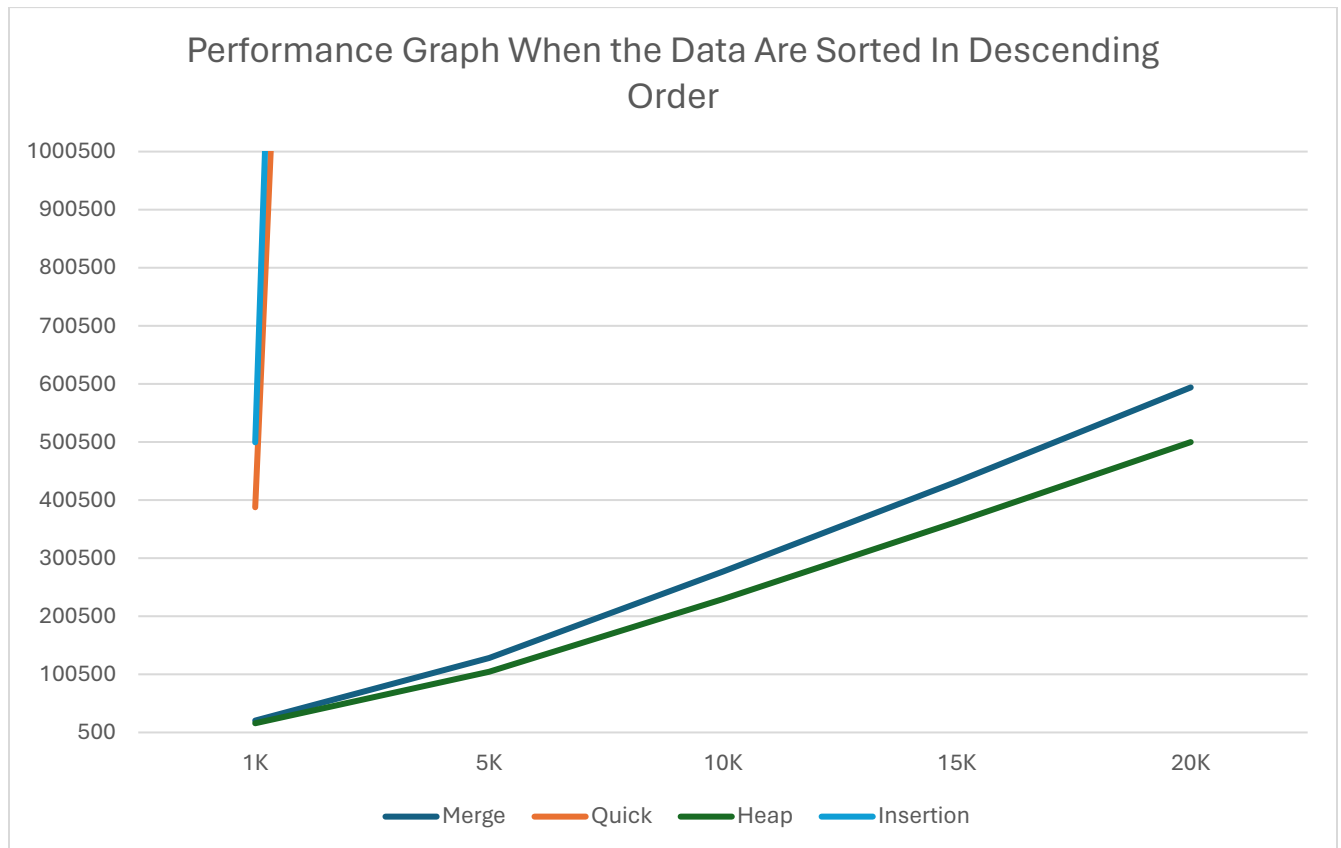
- a. Plot a graph showing the performance of each algorithm when the elements are already sorted. Determine which sorting algorithm's performance is the worst? What is the reason for this performance?



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.

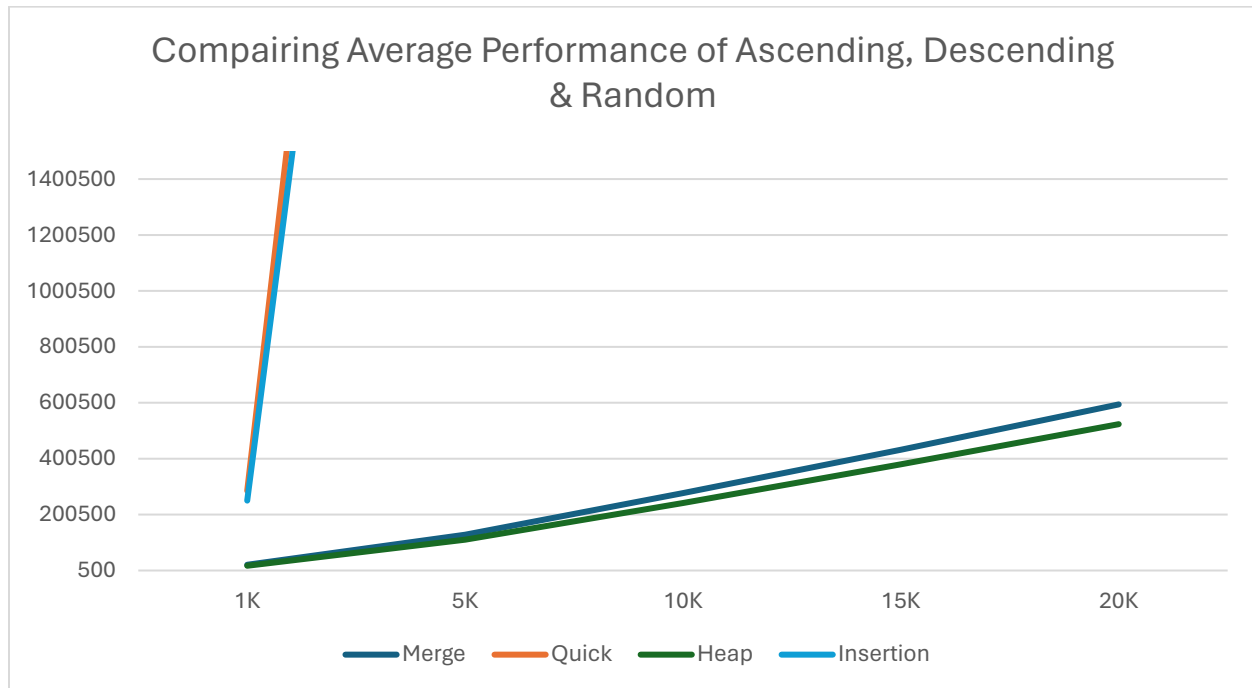
- b. Plot a graph showing the performance of each algorithm when the elements are sorted in descending order. Determine which sorting algorithm's performance is the worst? What is the reason for this performance?**



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.