Seatwork 7.1	
Using Sorting Algorithms	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09 - 16 - 25
Section: CPE21S4	**Date Submitted:** 09 - 16 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

Objectives: To create a program in C++ using sorting algorithms.

1. To be familiarized with the sorting algorithms.
2. To be able to differentiate the types of sorting algorithms.
3. To be able to create a program with sorting algorithms.

Answer the following questions:

1. What is a sorting algorithm?

   A sorting algorithm is a program used for rearranging data in a specific order. This is useful when our data has to be sifted through in a sequential order; like when using a binary search algorithm. For example, in a given array[5] = {1, 5, 2, 9, 4}, after using an increasing sorting algorithm, it will become array[5] = {1, 2, 4, 5, 9}. Afterwards, we can now apply a binary searching algorithm to search for a specific element. In this specific use-case, we can avoid the longer processing time of a linear searching algorithm by first sorting the array then using the faster binary searching algorithm.

2. Where can sorting algorithms be used?

   Sorting algorithms can be used in various applications; like data analysis, online shopping, item categorizing, scientific research, and many more. Rearranging it in sequential order makes the data more readable and usable. Additionally, they enhance data handling efficiency when the correct sorting algorithm is used thanks to the faster time complexity and effectiveness at sorting data. This faster and efficient handling of data improves user experience in real-life applications like social media algorithms. Overall, sorting algorithms improve efficiency in both the development side (e.g. data analysis) and the user experience side.

3. Explain the different types of sorting algorithms.
    a. Bubble Sorting
        - Bubble sorting involves comparing adjacent elements and moving the greater value higher and higher, starting at index 0. For example, for an increasing order in an array[3] = {0, 3, 1, 2 }, it first compares index 0 and 1 or elements 0 and 3. Because the right value is already greater than the left value, we keep it as is. Next, we compare index 1 and 2 or elements 3 and 1. In this case, the left value is greater than the next value so we swap their positions. Now the new arrangement is arr[3] = {0, 1, 3, 2}, we continue by comparing index 2 and 3 or elements 3 and 2 then swap them accordingly. Since the larger elements are continuously moving higher (element 3 moved from index 1, to 2, then to 3), it mimics the real life bubbles that float toward the surface, hence, its name.
    b. Selection Sorting
        - This is a comparison-based algorithm. It involves sorting an array by repeatedly choosing the smallest or largest element in an array then swapping it with the first element. This process is repeated until the element is properly sorted. For example, using the same array[3] = {0, 3, 1, 2 }, we first begin by comparing all the values to see which is the smallest, which is 0. Since it's already at the first index, we move on to the second smallest, which is 1, then move it next to 0 until we get the sorted array[3] = {0, 1, 2, 3}. This is generally faster and more memory efficient than bubble sorting as it involves less swapping albeit more comparisons.
    c. Insertion Sort
        - This algorithm involves iteratively inserting each element of an unsorted list into its correct position at the sorted portion of the list. The array gets split into two sections: the sorted portion and unsorted portion. It begins by comparing the second element with the first, if it is smaller, then we swap it to position it at the first position. Afterwards, we then compare the third element (in the unsorted) portion with the sorted portion (the first and second elements) and swap accordingly. This part repeats and ensures that with each iteration, the element to be inserted is placed at its correct position in the sorted section. For example, in the array[5] = {12, 11, 13, 5, 6}, we first compare 11 with 12. Since it is smaller, we insert it at the first position and move 12

ahead. Now, the sorted portion is in bold at array[5] = {**11, 12,** 13, 5, 6}. Next step is to compare 13 with the sorted portions, and since it is already greater than all elements, we keep it as is. The updated sorted portion is array[5] = {**11, 12, 13,** 5, 6}, and the next step is to compare 5 with the rest. Since it is actually less than all of them, it will be inserted at the beginning which makes the new sorted portion array[5] = {**5, 11, 12, 13,** 6}. Lastly, 6 is compared with the rest and inserted after 5 since it is greater than that value, finally returning the sorted array[5] = {5, 6, 11, 12, 13}.

4. Give sample programs in C++ that use sorting algorithms, specifically selection sort, insertion sort, and bubble sort. Explain how the programs work.
    a. Bubble sort:

```cpp
#include <iostream>

// perform bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size -1; ++step) {

    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping elements if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}
```

```cpp
// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    std::cout << array[i] << " ";
  }
  std::cout << "\n";
}

int main() {
  int data[] = {34, 44, 0, 10, -3};

  // find array's length
  int size = sizeof(data) / sizeof(data[0]);

  std::cout << "Unsorted array: \n";
  printArray(data, size);

  std::cout << "Sorted Array in Ascending Order using Bubble sort:\n";
  bubbleSort(data, size);
  printArray(data, size);

  return 0;
}
```

```
C:\Users\TIPQC\Downloads\S
Unsorted array:
34 44 0 10 -3
Sorted Array in Ascending Order using Bubble sort:
-3 0 10 34 44

--------------------------------
Process exited after 0.01056 seconds with return value 0
Press any key to continue . . .
```

Analysis:

This is a simple program that demonstrates bubble sorting. The first for loop accesses each element, and in each iteration a nested for loop executes, which serves as the comparison for loop. If the array[i] element (left) is greater than the right element array[i + 1], then the swap code block executes. After each comparison, we go to the next step which is the second iteration of the outer for loop and continues until we finish sorting the array.

b. Selection sort

```cpp
// Selection sort in C++

#include <iostream>

// function to swap the the position of two elements
void swap(int *a, int *b) { // pointers that point toward the
  int temp = *a;            // address of the two elements to be swapped
  *a = *b;
  *b = temp;
}

// function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    std::cout << array[i] << " ";
  }
  std::cout << '\n';
}

void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) { //loops through each element

      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.

      if (array[i] < array[min_idx])
        min_idx = i;
    }

    // put min at the correct position
    swap(&array[min_idx], &array[step]); // uses them as passby references to swap the elements
  }
}

// driver code
int main() {
  int data[] = {20, 12, 10, 15, 2};
  int size = sizeof(data) / sizeof(data[0]);

  std::cout << "Unsorted Array: \n";
  printArray(data, size);



  std::cout << "Sorted array in Acsending Order:\n";
  selectionSort(data, size);
  printArray(data, size);
}
```

```
Unsorted Array:
20 12 10 15 2
Sorted array in Acsending Order:
2 10 12 15 20

--------------------------------
Process exited after 0.01615 seconds with return value 0
Press any key to continue . . .
```

Analysis:

This program demonstrates the selection sorting algorithm. In this case, we create the swap function separately for modularity and ease of calling. Per movement, we indicate as the min index (or our point of comparison on the left) is the current step or iteration. Afterwards, we compare it to all of the elements to the right. If the array[i] is less than the min index, we update the min index to be the lesser value. This loop continues until we get the smallest value, after which we call the swap function to swap the min index and the current step, essentially moving the smallest element to the left.

c. Insertion sort

```cpp
// Insertion sort in C++

#include <iostream>

// Function to print an array
void printArray(int array[], int size) {

void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step]; // store the array to be compared on the key variable
    int j = step - 1;

    // Compare key with each element on the left of it until an element smaller than
    // it is found.
    // For descending order, change key<array[j] to key>array[j].

    while (j >= 0 && key < array[j]) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}
```

```
30  // Driver code
31  int main() {
32      int data[] = {14, -3, 0, 9, 8};
33      int size = sizeof(data) / sizeof(data[0]);
34
35      std::cout << "Unsorted array: \n";
36      printArray(data, size);
37
38      std::cout << "Sorted array in ascending order:\n";
39      insertionSort(data, size);
40      printArray(data, size);
41  }
```

```
Unsorted array:
14 -3 0 9 8
Sorted array in ascending order:
-3 0 8 9 14

--------------------------------
Process exited after 0.007229 seconds with return value 0
Press any key to continue . . .
```

Analysis:

This function is a little different from the selection and bubble sorting because it involves a key variable. The key will be the element that we will compare with the other values. The while loop essentially moves the key all the way to the left as long as the key is greater than the current and the j is within the index scope of the array. After each run of the loop, the key is updated to be the element at index j + 1 or the one at the right. This loop continues until we reach the end of the array, making the array now sorted.

References:

Algorithm Examples. (2024, October 31). 11 Everyday Life Uses of sorting Algorithms | Blog Algorithm Examples. *Algorithm Examples*. https://blog.algorithmexamples.com/sorting-algorithm/11-everyday-life-uses-of-sorting-algorithms/

GeeksforGeeks. (2025, July 23). *Selection Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/selection-sort-algorithm-2/

GeeksforGeeks. (2025b, August 24). *Sorting algorithms*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/sorting-algorithms/

Bubble Sort (With code in Python/C++/Java/C). (n.d.). https://www.programiz.com/dsa/bubble-sort

W3Schools.com. (n.d.). https://www.w3schools.com/dsa/dsa_algo_insertionsort.php

Source codes retrieved and analyzed from:

Bubble Sort (With code in Python/C++/Java/C). (n.d.-b). https://www.programiz.com/dsa/bubble-sort

Insertion Sort (With code in Python/C++/Java/C). (n.d.). https://www.programiz.com/dsa/insertion-sort

Selection Sort (With code in Python/C++/Java/C). (n.d.). https://www.programiz.com/dsa/selection-sort

7. Conclusion

This activity explores the three basic types of sorting algorithms: the bubble sort, selection sort, and the insertion sort. First, the bubble sort involves comparing two adjacent elements at a type and swapping them when the greater or lesser value is at the wrong position. This loop continues until the greater value is moved all the way to the right, then we return to the first value to repeat the process again. This poses an inefficient way of sorting due to the fact that we have to do multiple comparisons and swapping starting from the first all the way to the last element. Moving on, the selection sort is by far the most efficient sorting

algorithm due to the least amount of swapping involved. It first selects one element and compares it to all of the elements first before swapping it. This ensures that we only do one swap per element which is faster and more efficient. Despite this, it still has a flaw in which it cannot immediately recognize an already sorted array due to the need to do the comparisons per iteration. Lastly, the insertion sort involves a key variable that is compared toward the sorted values toward the left, like sorting cards at hand. It does this by comparing the key value toward the left, swapping it if it is less, then getting another key variable which is at the right. The key is compared to all of the values of the left, which makes its worst-case scenario very memory inefficient and yields a quadratic time complexity due to the fact that we have to compare and/or swap the key every iteration. Overall, learning about the workings and logic behind sorting algorithms made me appreciate this part of data analysis and structure learning much more.

8. Assessment Rubric