

## ACTIVITY NO. 7

SORTING ALGORITHMS: BUBBLE, SELECTION, AND INSERTION SORT	
<b>Course Code:</b> CPE010	<b>Program:</b> Computer Engineering
<b>Course Title:</b> Data Structures and Algorithms	<b>Date Performed:</b> 09/18/25
<b>Section:</b> CPE21S4	<b>Date Submitted:</b> 09/18/25
<b>Name:</b> Cruz, Axl Waltz E.	<b>Instructor:</b> Engr. Jimlord Quejado
<b>1. Objective(s)</b>	
Create C++ code to sort through data using sorting algorithms	
<b>2. Intended Learning Outcomes (ILOs)</b>	
After this activity, the student should be able to: <ul style="list-style-type: none"><li>• Create C++ code for implementation of selection, bubble, insertion and merge sort.</li><li>• Solve given data sorting problems using appropriate basic sorting algorithms</li></ul>	
<b>3. Discussion</b>	
Sorting is the process of placing elements from a collection in order. For example, a list of words could be sorted alphabetically or by length. A list of cities could be sorted by population, by area, or by zip code. We have already seen several algorithms that were able to benefit from having a sorted list (recall the final anagram example and the binary search).	
<b>4. Materials and Equipment</b>	
Personal Computer with C++ IDE Recommended IDE: <ul style="list-style-type: none"><li>• CLion (must use TIP email to download)</li><li>• DevC++ (use the embarcadero fork or configure to C++17)</li></ul>	
<b>5. Procedure</b>	

## **ILO A: Create C++ code for implementation of selection, bubble, insertion and merge sort.**

### **Preparation Task:**

- Create an array of elements with random values. The array must contain 100 elements that are not sorted. Use the created array for each sorting algorithm below. Show the output of the preparation task on table 7-1 in section 6.
- Create a header file for the implementation of the different sorting algorithms.
- Import this header file into your main.cpp.

### **A.1. Bubble Sort Technique**

Using the bubble sort technique, sorting is done in passes or iteration. Thus at the end of each iteration, the heaviest element is placed at its proper place in the list. In other words, the largest element in the list bubbles up.

#### **General Algorithm**

```
template <typename T>
void bubbleSort(T arr[], size_t arrSize){
    //Step 1: For i = 0 to N-1 repeat Step 2
    for(int i = 0; i < arrSize; i++){
        //Step 2: For J = i + 1 to N - I repeat
        for(int j = i+1; j < arrSize; j++){
```

```

        //Step 3: if A[J] > A[i]
        if(arr[j]>arr[i]){
            //Swap A[J] and A[i]
            std::swap(arr[j], arr[i]);
        }
        //[End of Inner for loop]
    }
    //[End if Outer for loop]
}
//Step 4: Exit
}

```

Include a screenshot of your code and output console along with your observations in table 7-2.

## A.2.Selection Sort Algorithm

Selection sort is quite a straightforward sorting technique as the technique only involves finding the smallest element in every pass and placing it in the correct position. Selection sort works efficiently when the list to be sorted is of small size but its performance is affected badly as the list to be sorted grows in size.

Hence we can say that selection sort is not advisable for larger lists of data.

```

template <typename T> int Routine_Smallest(T A[], int K, const int arrSize);

```

**//General Algorithm**

**//Selection Sort (Array, Size of Array)**

```

template <typename T>
void selectionSort(T arr[], const int N){
    int POS, temp, pass=0;
    //Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
    for(int i = 0; i < N; i++){
        //Step 2: Call routine smallest(A, K, N, POS)
        POS = Routine_Smallest(arr, i, N);
        temp = arr[i];
        //Step 3: Swap A[K] with A [POS]
        arr[i] = arr[POS];
        arr[POS] = temp;
        //Count
        pass++;
    }
    //[End of loop]
    //Step 4: EXIT
}

```

**//Routine smallest (Array, Current Position, Size of Array)**

```

template <typename T>
int Routine_Smallest(T A[], int K, const int arrSize){
    int position, j;
    //Step 1: [initialize] set smallestElem = A[K]
    T smallestElem = A[K];
    //Step 2: [initialize] set POS = K
    position = K;
    //Step 3: for J = K+1 to N -1,repeat
    for(int J=K+1; J < arrSize; J++){
        if(A[J] < smallestElem){

```

```
smallestElem = A[J];
```

```

        position = J;
    }
}
//Step 4: return POS
return position;
}

```

Include a screenshot of your code and output console along with your observations in table 7-3.

### A.3.Insertion Sort Algorithm

In the insertion sort technique, we start from the second element and compare it with the first element and put it in a proper place. Then we perform this process for the subsequent elements. We compare each element with all its previous elements and put or insert the element in its proper position. Insertion sort technique is more feasible for arrays with a smaller number of elements. It is also useful for sorting linked lists. Linked lists have a pointer to the next element (in case of a singly linked list) and a pointer to the previous element as well (in case of a doubly linked list). Hence it becomes easier to implement insertion sort for a linked list.

#### General Algorithm

```

//General Algorithm
//Insertion Sort
template <typename T>
void insertionSort(T arr[], const int N){
    int K = 0, J, temp;
    //Step 1: Repeat Steps 2 to 5 for K = 1 to N-1
    while(K < N){
        //Step 2: set temp = A[K]
        temp = arr[K];
        //Step 3: set J = K - 1
        J = K-1;
        //Step 4: Repeat while temp <=A[J]
        while(temp <= arr[J]){
            //set A[J + 1] = A[J]
            arr[J+1] = arr[J];
            //set J = J - 1
            J--;
            //[end of inner loop]
        }
        //Step 5: set A[J + 1] = temp
        arr[J+1] = temp;
        //[end of loop]
        K++;
    }
    //Step 6: exit
}

```

Include a screenshot of your code and output console along with your observations in table 7-4.

## 6. Output

Code + Console Screenshot

Table 7-1. Array of Values for Sort Algorithm Testing

```
1  #ifndef SORT_ALGORITHMS_H 50
2  #define SORT_ALGORITHMS_H 51
3  52
4  void copyArray(int source[], int destination[], int size) { 53
5      for (int i = 0; i < size; i++) { 54
6          destination[i] = source[i]; 55
7      } 56
8  } 57
9  58
10 void selectionSort(int arr[], int size) { 59
11     for (int i = 0; i < size - 1; i++) { 60
12         int minIndex = i; 61
13         for (int j = i + 1; j < size; j++) { 62
14             if (arr[j] < arr[minIndex]) { 63
15                 minIndex = j; 64
16             } 65
17         } 66
18         int temp = arr[i]; 67
19         arr[i] = arr[minIndex]; 68
20         arr[minIndex] = temp; 69
21     } 70
22 } 71
23 72
24 73
25 void bubbleSort(int arr[], int size) { 74
26     for (int i = 0; i < size - 1; i++) { 75
27         for (int j = 0; j < size - i - 1; j++) { 76
28             if (arr[j] > arr[j + 1]) { 77
29                 int temp = arr[j]; 78
30                 arr[j] = arr[j + 1]; 79
31                 arr[j + 1] = temp; 80
32             } 81
33         } 82
34     } 83
35 } 84
36 85
37 void insertionSort(int arr[], int size) { 86
38     for (int i = 1; i < size; i++) { 87
39         int key = arr[i]; 88
40         int j = i - 1; 89
41         while (j >= 0 && arr[j] > key) { 90
42             arr[j + 1] = arr[j]; 91
43             j = j - 1; 92
44         } 93
45         arr[j + 1] = key; 94
46     } 95
47 } 96
48 97
49 98
50 99
51 100
52 101
53 102
54 103
55 104
```

```

void merge(int arr[], int left, int middle, int right) {
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[100], R[100];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];

    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = (left + right) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }
}

#endif

```

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  #include "sort_algorithms.h"
5
6  using namespace std;
7
8  const int SIZE = 100;
9
10 void printArray(int arr[], int size) {
11     for (int i = 0; i < size; i++) {
12         cout << arr[i] << " ";
13     }
14     cout << endl;
15 }
16
17 int main() {
18     int original[SIZE];
19     int arr[SIZE];
20
21     srand(time(0));
22
23     for (int i = 0; i < SIZE; i++) {
24         original[i] = rand() % 1000;
25     }
26
27     cout << "Original Array:\n";
28     printArray(original, SIZE);
29
30     copyArray(original, arr, SIZE);
31     selectionSort(arr, SIZE);
32     cout << "\nArray after Selection Sort:\n";
33     printArray(arr, SIZE);
34
35     copyArray(original, arr, SIZE);
36     bubbleSort(arr, SIZE);
37     cout << "\nArray after Bubble Sort:\n";
38     printArray(arr, SIZE);
39
40     copyArray(original, arr, SIZE);
41     insertionSort(arr, SIZE);
42     cout << "\nArray after Insertion Sort:\n";
43     printArray(arr, SIZE);
44
45     copyArray(original, arr, SIZE);
46     mergeSort(arr, 0, SIZE - 1);
47     cout << "\nArray after Merge Sort:\n";
48     printArray(arr, SIZE);
49
50     return 0;
51 }

```



	<pre>Original Array: 950 490 424 224 558 949 41 937 696 731 201 302 361 171 879 681 382 736 346 313 587 870 816 388 497 682 314 169 729 625 735 973 21 40 460 821 331 980 250 199 465 854 908 100 801 744 827 388 342 49 149 480 856 446 290 602 241 756 361 379 672 691 721 468 630 74 1 437 71 800 309 358 255 340 391 794 313 463 23 852 141 118 913 340 944 812 564 929 682 267 773 299 983 886 57 70 800 247 2 444 758 837  Array after Selection Sort: 2 23 41 49 57 70 71 100 118 141 149 160 171 199 201 224 240 241 247 250 255 267 290 299 302 313 314 331 333 340 340 342 346 358 361 361 379 380 382 388 391 399 424 437 444 446 460 460 463 465 480 490 497 558 564 587 682 625 630 672 681 682 682 691 696 721 729 731 735 736 741 744 750 758 773 794 800 812 816 821 827 837 852 854 856 870 879 881 886 890 900 913 929 937 944 949 950 973 983 998  Array after Bubble Sort: 2 23 41 49 57 70 71 100 118 141 149 160 171 199 201 224 240 241 247 250 255 267 290 299 302 313 314 331 333 340 340 342 346 358 361 361 379 380 382 388 391 399 424 437 444 446 460 460 463 465 480 490 497 558 564 587 682 625 630 672 681 682 682 691 696 721 729 731 735 736 741 744 750 758 773 794 800 812 816 821 827 837 852 854 856 870 879 881 886 890 900 913 929 937 944 949 950 973 983 998  Array after Insertion Sort: 2 23 41 49 57 70 71 100 118 141 149 160 171 199 201 224 240 241 247 250 255 267 290 299 302 313 314 331 333 340 340 342 346 358 361 361 379 380 382 388 391 399 424 437 444 446 460 460 463 465 480 490 497 558 564 587 682 625 630 672 681 682 682 691 696 721 729 731 735 736 741 744 750 758 773 794 800 812 816 821 827 837 852 854 856 870 879 881 886 890 900 913 929 937 944 949 950 973 983 998  Array after Merge Sort: 2 23 41 49 57 70 71 100 118 141 149 160 171 199 201 224 240 241 247 250 255 267 290 299 302 313 314 331 333 340 340 342 346 358 361 361 379 380 382 388 391 399 424 437 444 446 460 460 463 465 480 490 497 558 564 587 682 625 630 672 681 682 682 691 696 721 729 731 735 736 741 744 750 758 773 794 800 812 816 821 827 837 852 854 856 870 879 881 886 890 900 913 929 937 944 949 950 973 983 998  ----- Process exited after 0.1331 seconds with return value 0 Press any key to continue . . .</pre>
Observations	<p>My observation on this code is compares four sorting algorithms using the same set of random values. It shows how each sort affects the array, making it easy to see differences in sorting results. The structure is clear and easy to follow for learning purposes.</p>

Table 7-2. Bubble Sort Technique

Code + Console Screenshot

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  #include "bubble_sort.h"
5
6  using namespace std;
7
8  const int SIZE = 100;
9
10 void printArray(int arr[], int size) {
11     for (int i = 0; i < size; i++) {
12         cout << arr[i] << " ";
13     }
14     cout << endl;
15 }
16
17 int main() {
18     int numbers[SIZE];
19
20     srand(time(0));
21
22     for (int i = 0; i < SIZE; i++) {
23         numbers[i] = rand() % 1000;
24     }
25
26     cout << "Original Array:\n";
27     printArray(numbers, SIZE);
28
29     bubbleSort(numbers, SIZE);
30
31     cout << "\nArray After Bubble Sort:\n";
32     printArray(numbers, SIZE);
33
34     return 0;
35 }

```

```

1  #ifndef BUBBLE_SORT_H
2  #define BUBBLE_SORT_H
3
4  void bubbleSort(int arr[], int size) {
5
6     for (int i = 0; i < size - 1; i++) {
7
8         for (int j = 0; j < size - i - 1; j++) {
9
10            if (arr[j] > arr[j + 1]) {
11
12                int temp = arr[j];
13                arr[j] = arr[j + 1];
14                arr[j + 1] = temp;
15            }
16        }
17    }
18 }
19
20 #endif

```

	<pre> Original Array: 573 398 546 281 137 342 699 315 217 372 379 299 321 848 452 266 496 764 329 420 949 949 990 799 699 541 272 813 390 29 2 51 651 590 282 747 777 579 357 476 658 794 135 342 230 911 364 283 343 987 208 971 608 249 567 571 277 867 945 921 630 7 86 822 721 470 446 551 848 692 459 46 974 87 68 950 718 359 804 197 710 794 667 704 837 884 469 280 12 203 506 637 603 4 26 241 200 954 571 862 5 33 987  Array After Bubble Sort: 5 12 20 33 46 68 87 135 137 197 200 203 203 208 217 230 241 249 251 266 272 277 280 281 282 290 315 321 329 342 342 343 357 359 364 372 379 390 398 420 426 446 452 459 469 470 476 496 506 541 546 551 567 571 571 573 579 590 603 608 630 637 651 658 667 692 699 699 704 706 710 718 731 747 764 777 794 794 799 804 813 822 837 848 848 862 867 884 911 921 945 949 949 950 954 971 974 987 987 990  ----- Process exited after 0.1222 seconds with return value 0 Press any key to continue . . . </pre>
Observations	<p>My observation in this code is uses bubble sort to arrange 100 random numbers in ascending order. It prints the original and sorted arrays, making it easy to understand how the sorting changed the data. While bubble sort is simple and not the fastest, it works well here for demonstrating how sorting algorithms function.</p>
Code + Console Screenshot	<p>Table 7-3. Selection Sort Algorithm</p> <pre> 1 #include &lt;iostream&gt; 2 #include "selectionSort.h" 3 using namespace std; 4 5 int main() { 6     int numbers[] = {20, 10, 50, 30, 40}; 7     int size = sizeof(numbers) / sizeof(numbers[0]); 8 9     cout &lt;&lt; "Original array: "; 10    for(int i = 0; i &lt; size; i++) { 11        cout &lt;&lt; numbers[i] &lt;&lt; " "; 12    } 13    cout &lt;&lt; endl; 14 15    selectionSort(numbers, size); 16 17    cout &lt;&lt; "Sorted array: "; 18    for(int i = 0; i &lt; size; i++) { 19        cout &lt;&lt; numbers[i] &lt;&lt; " "; 20    } 21    cout &lt;&lt; endl; 22 23    return 0; 24 } 25 </pre>

	<pre> 1  #ifndef SELECTIONSORT_H 2  #define SELECTIONSORT_H 3 4  #include &lt;iostream&gt; 5  using namespace std; 6 7  // Routine to find the smallest element position 8  template &lt;typename T&gt; 9  int Routine_Smallest(T A[], int K, const int arrSize) { 10     int position = K; 11     T smallestElem = A[K]; 12 13     for(int J = K + 1; J &lt; arrSize; J++) { 14         if(A[J] &lt; smallestElem) { 15             smallestElem = A[J]; 16             position = J; 17         } 18     } 19     return position; 20 } 21 22 // Selection Sort 23 template &lt;typename T&gt; 24 void selectionSort(T arr[], const int N) { 25     int POS, pass = 0; 26 27     for(int i = 0; i &lt; N; i++) { 28         POS = Routine_Smallest(arr, i, N); 29 30         // Swap arr[i] and arr[POS] 31         T temp = arr[i]; 32         arr[i] = arr[POS]; 33         arr[POS] = temp; 34 35         pass++; 36     } 37 } 38 39 #endif 40 41 10 20 30 40 50 42 ----- 43 Process exited after 0.09899 seconds with return value 0 44 Press any key to continue . . . </pre>
<p>Observations</p>	<p>My observation in this part is the program uses a header file that defines a routine to find the smallest element and a selection sort template function. It prints both the original and sorted array, allowing you to clearly see the effect of the sort. This example demonstrates clean modular programming by separating the sorting logic into its own header file, making the code easier to maintain and reuse.</p>

Code + Console Screenshot

Table 7-4. Insertion Sort Algorithm

```
1 #include <iostream>
2 #include "insertionSort.h"
3 using namespace std;
4
5 int main() {
6     int numbers[] = {20, 10, 50, 30, 40};
7     int size = sizeof(numbers) / sizeof(numbers[0]);
8
9     cout << "Original array: ";
10    for (int i = 0; i < size; i++) {
11        cout << numbers[i] << " ";
12    }
13    cout << endl;
14
15    // Call Insertion Sort
16    insertionSort(numbers, size);
17
18    cout << "Sorted array: ";
19    for (int i = 0; i < size; i++) {
20        cout << numbers[i] << " ";
21    }
22    cout << endl;
23
24    return 0;
25 }
```

```
1 #ifndef INSERTIONSORT_H
2 #define INSERTIONSORT_H
3
4 #include <iostream>
5 using namespace std;
6
7 template <typename T>
8 void insertionSort(T arr[], const int N) {
9     int K = 1;
10    while (K < N) {
11        T temp = arr[K];
12        int J = K - 1;
13
14        while (J >= 0 && arr[J] > temp) {
15            arr[J + 1] = arr[J];
16            J--;
17        }
18
19        arr[J + 1] = temp;
20        K++;
21    }
22 }
23
24 #endif
25
```

Original array: 20 10 50 30 40  
Sorted array: 10 20 30 40 50

-----  
Process exited after 0.1004 seconds with return value 0  
Press any key to continue . . .

Observations

I observe here was the program demonstrates sorting an array

using insertion sort, with the algorithm defined in a separate header file. It displays the original unsorted array, performs the sort, and then shows the sorted output. This version highlights a different algorithm while still keeping the modular structure, making the code easy to organize and understand.

## 7. Supplementary Activity

**ILO B: Solve given data sorting problems using appropriate basic sorting algorithms**

## List of Candidates

**Problem:** Generate an array A[0...100] of unsorted elements, wherein the values in the array are indicative of a vote to a candidate. This means that the values in your array must only range from 1 to 5. Using sorting and searching techniques, develop an algorithm that will count the votes and indicate the winning candidate.

**NOTE:** The sorting techniques you have the option of using in this activity can be either bubble, selection, or insertion sort. Justify why you chose to use this sorting algorithm.

## Deliverables:

- Pseudocode of Algorithm
- Screenshot of Algorithm Code
- Output Testing

Output Console Showing Sorted Array	Manual Count	Count Result of Algorithm
<pre>Sorted Array: 1111111111111111222222222222222222223333333333333333444 44444444444444444444445555555555555555555555555555555</pre>	<pre>Manual Count: Axolt:17 Jason:22 Kerwin:18 Peter:22 Griffin:21</pre>	<pre>Axolt:17 Jason:22 Kerwin:18 Peter:22 Griffin:21  Winner is Candidate 2 with 22 votes!</pre>

- The code uses selection sort to arrange votes and counts how many each candidate received. It clearly identifies the winner based on the highest vote count. The logic is simple and works well for small datasets like this.

Candidate 1	Bo Dalton Capistrano
Candidate 2	Cornelius Raymon Agustín
Candidate 3	Deja Jayla Bañaga
Candidate 4	Lalla Brielle Yabut
Candidate 5	Franklin Relano Castro

**Question:** Was your developed vote counting algorithm effective? Why or why not?

- Yes, the developed vote counting algorithm was effective. It correctly generated random votes, sorted them using a simple algorithm (insertion sort), and accurately counted each candidate's total using conditional checks. Sorting the array first made it easier to manually verify the results, and the logic was clear, efficient for the data size, and easy to understand and debug.

## 8. Conclusion

Provide the following:

- Summary of lessons learned
  - I learned how to apply basic sorting algorithms such as Bubble Sort, Selection Sort, and Insertion Sort in C++. These algorithms helped me understand how data can be arranged for easier analysis and comparison. I also learned how sorting can support other tasks like searching, counting, and organizing data clearly.
- Analysis of the procedure
  - The activity involved generating random data, applying multiple sorting techniques, and displaying results in a clear format. Using a header file made the code more organized and reusable across different parts of the program. Each sorting method showed different behaviors, which helped me compare their strengths and weaknesses.
- Analysis of the supplementary activity
  - The vote-counting task showed how sorting could assist in analyzing grouped data more easily. By sorting the votes first, I was able to count efficiently using simple conditional logic. The activity proved that even basic algorithms can solve real-world problems when applied properly.
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
  - I think I did well in completing the activity by writing clear, working code and understanding the logic behind each step. One area I can improve is optimizing the code and reducing repetition in certain sections. Overall, the activity helped me build confidence in using C++ for basic algorithm implementation and problem-solving.

## 9. Assessment Rubric