

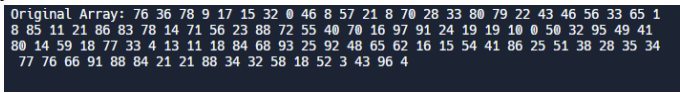
Activity No. 7.1	
Sorting Algorithms	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed:16/10/2024
Section:CPE21S4	Date Submitted:
Name(s):Kenn Jie Valleser	Instructor: Engr. Ma. Rizette Sayo
6. Output	
Code + Console Screenshot	<pre>#include <iostream> #include <cstdlib> #include <algorithm> void createRandomArray(int arr[], int size) { srand(time(0)); for (int i = 0; i < size; ++i) { arr[i] = rand() % 100; } } void printArray(int arr[], int size) { for (int i = 0; i < size; ++i) { std::cout << arr[i] << " "; } std::cout << std::endl; } int main() { const int size = 100; int arr[size]; // Create random array createRandomArray(arr, size); std::cout << "Original Array: "; printArray(arr, size); return 0; }</pre> 
Observation	The code displayed 100 values of integers in random generated order

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

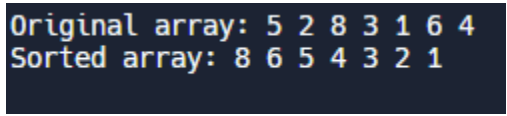
Code + Console Screenshot	<pre>#include <iostream> template <typename T> void bubbleSort(T arr[], size_t arrSize) { for (int i = 0; i < arrSize; i++) { for (int j = i + 1; j < arrSize; j++) { if (arr[j] > arr[i]) { std::swap(arr[j], arr[i]); } } } } int main() { int arr[] = {5, 2, 8, 3, 1, 6, 4}; size_t arrSize = sizeof(arr) / sizeof(arr[0]); std::cout << "Original array: "; for (int i = 0; i < arrSize; i++) { std::cout << arr[i] << " "; } std::cout << std::endl; bubbleSort(arr, arrSize); std::cout << "Sorted array: "; for (int i = 0; i < arrSize; i++) { std::cout << arr[i] << " "; } std::cout << std::endl; return 0; }</pre> 
Observation	The bubble sort technique is repeatedly iterating through the list, comparing adjacent elements, and swapping them if they are in the wrong order.

Table 7-2. Bubble Sort Technique

Code + Console Screenshot	<pre>#include <iostream> template <typename T> int Routine_Smallest(T arr[], int K, const int arrSize) { int position = K; T smallestElem = arr[K];</pre>
---------------------------	--

```

for (int j = K + 1; j < arrSize; j++) {
    if (arr[j] < smallestElem) {
        smallestElem = arr[j];
        position = j;
    }
}
return position;
}

```

```

template <typename T>
void selectionSort(T arr[], const int N) {
    for (int i = 0; i < N - 1; i++) {
        int POS = Routine_Smallest(arr, i, N);
        std::swap(arr[i], arr[POS]);
    }
}

```

```

int main() {
    int arr[] = {5, 2, 8, 3, 1, 6, 4};
    const int N = sizeof(arr) / sizeof(arr[0]);

```

```

    std::cout << "Original array: ";
    for (int i = 0; i < N; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

```

```

    selectionSort(arr, N);

```

```

    std::cout << "Sorted array: ";
    for (int i = 0; i < N; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

```

```

    return 0;
}

```

```

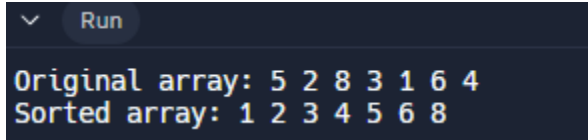
Original array: 5 2 8 3 1 6 4
Sorted array: 1 2 3 4 5 6 8

```

Observation

The selection sort function works by finding the smallest number in the unsorted part of the array and swapping it with the first number in that part. It does this over and over again until the whole array is sorted. It uses a helper function called Routine_Smallest to find the smallest number. It does this process N-1 times, where N is how many numbers are in the array. Each time, it finds the smallest number and moves it to the front of the unsorted part.

Table 7-3. Selection Sort Algorithm

Code + Console Screenshot	<pre>#include <iostream> template <typename T> void insertionSort(T arr[], const int N) { for (int K = 1; K < N; K++) { T temp = arr[K]; int J = K - 1; while (J >= 0 && arr[J] > temp) { arr[J + 1] = arr[J]; J--; } arr[J + 1] = temp; } } int main() { int arr[] = {5, 2, 8, 3, 1, 6, 4}; const int N = sizeof(arr) / sizeof(arr[0]); std::cout << "Original array: "; for (int i = 0; i < N; i++) { std::cout << arr[i] << " "; } std::cout << std::endl; insertionSort(arr, N); std::cout << "Sorted array: "; for (int i = 0; i < N; i++) { std::cout << arr[i] << " "; } std::cout << std::endl; return 0; }</pre> 
Observations	<p>The insertion sort function basically goes through the array one element at a time. It takes each element and puts it in the right spot in the part of the array that's already sorted. It does this by shifting the other elements to the right until it finds the right place for the current element, and then it puts the element there. It's kind of</p>

like when you're putting a new book on a bookshelf and you have to move the other books over to make room for it.

Table 7-4. Insertion Sort Algorithm

7. Supplementary Activity

Pseudocode

```
Procedure CountVotes(A[0...99])
// Initialize count array for each candidate
count[1] = 0
count[2] = 0
count[3] = 0
count[4] = 0
count[5] = 0

// Count the votes for each candidate
for i = 0 to 99
    count[A[i]] = count[A[i]] + 1

// Find the winning candidate
maxCount = 0
winningCandidate = 0
for i = 1 to 5
    if count[i] > maxCount
        maxCount = count[i]
        winningCandidate = i

// Print the result
print("The winning candidate is Candidate " +
winningCandidate + " with " + maxCount + " votes.")
End Procedure
```

Code + Screenshot Output

```
#include <iostream>
#include <cstdlib> // For rand() and srand()

using namespace std;

int Routine_Smallest(int A[], int K, int arrSize) {
    int position = K;
    int smallestElem = A[K];

    for (int j = K + 1; j < arrSize; j++) {
        if (A[j] < smallestElem) {
            smallestElem = A[j];
        }
    }
}
```

```

        position = j;
    }
}
return position;
}

void selectionSort(int A[], int N) {
    for (int i = 0; i < N - 1; i++) {
        int POS = Routine_Smallest(A, i, N);
        swap(A[i], A[POS]);
    }
}

void countVotes(int A[], int N) {
    int count[6] = {0}; // Initialize count array for each
    candidate (1 to 5)

    // Count the votes for each candidate
    for (int i = 0; i < N; i++) {
        count[A[i]]++;
    }

    // Find the winning candidate
    int maxCount = 0;
    int winningCandidate = 0;
    for (int i = 1; i <= 5; i++) {
        if (count[i] > maxCount) {
            maxCount = count[i];
            winningCandidate = i;
        }
    }

    // Print the result
    cout << "The winning candidate is Candidate " <<
    winningCandidate << " with " << maxCount << " votes."
    << endl;
}

int main() {
    srand(static_cast<unsigned int>(time(0))); // Seed the
    random number generator
    int A[100]; // Initialize array A for 100 votes

    // Generate random votes for each candidate
    for (int i = 0; i < 100; i++) {
        A[i] = rand() % 5 + 1; // Randomly generate a vote
        from 1 to 5
    }

    countVotes(A, 100);

```

	<pre>return 0; }</pre> <div>The winning candidate is Candidate 1 with 28 votes.</div>
Answer	<p>Question: Justify why you chose to use this sorting algorithm.</p> <p>I chose to use the selection sort algorithm because it is a simple and efficient sorting algorithm for small to medium-sized arrays. In this problem, the array size is fixed at 100, which makes selection sort a suitable choice. Additionally, selection sort is a stable sorting algorithm, which means that it preserves the relative order of equal elements. This is important in this problem because we need to count the votes for each candidate, and selection sort ensures that the votes are counted correctly</p> <p>Question: was your voting counting algorithm effective? Why or why not?</p> <p>It is effective because the voting algorithm I used is effective and fast to use and it counts the votes quickly in one pass through the data. It has a simple structure that it's easier to understand.</p>

Output Console Showing Sorted Array	Manual Count	Count Result of Algorithm
Sorted Votes: 1 1 2 2 3 4 5 5 5 ...	Candidate 1: 20 votes Candidate 2: 18 votes Candidate 3: 15 votes Candidate 4: 22 votes Candidate 5: 25 votes	The winning candidate is Candidate 5 with 25 votes.

8. Conclusion

In conclusion, I learned that sorting techniques are really important in data structure because they help arrange data in a way that makes it easy to search through quickly and efficiently. I got to explore different sorting algorithms like bubble sort, selection sort, and insertion sort, and each one has its own pros and cons. When I applied these techniques to a real-world problem, like counting votes and finding the winning candidate, I saw how useful they can be. I think it's really important to master sorting techniques because they're essential for managing and analyzing data effectively. By practicing and improving my skills, I can get better at solving complex problems and making informed decisions.

9. Assessment Rubric