FACULTY OF COMPUTING
SEMESTER 1 2024/2025

SECJ2013 DATA STRUCTURE AND ALGORITHM

SECTION 02

Assignment 1

LECTURER: Dr Zuraini binti Ali Shah

GROUP 9

Student Name	Matric No.
DHESHIEGHAN A/L SARAVANA MOORTHY	A23CS0072
LAU YAN KAI	A23CS0098
NURUL ADRIANA BINTI KAMAL JEFRI	A23CS0258
TEH RU QIAN	A23CS0191

Bubble Sort

Source code

```
#include <iostream>
using namespace std;

void bubbleSort(int array[], int n, int &comparisons, int &swaps) {
    comparisons = 0, swaps = 0;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            comparisons++;
            if (array[j] > array[j + 1]) {
                swap(array[j], array[j + 1]);
                swaps++;
            }
        }
    }
}

int main() {
    int marks[] = {75, 95, 60, 88, 70};
    int n = sizeof(marks) / sizeof(marks[0]);
    int comparisons, swaps;

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

    bubbleSort(marks, n, comparisons, swaps);

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

    cout << "Total comparisons: " << comparisons << endl;
    cout << "Total swaps: " << swaps << endl;

    return 0;
}
```

Sample Output for Bubble Sort

```
Original Array: 75 95 60 88 70

Comparison: 1
Compare 75 with 95 → No swap
Comparison: 2
Compare 95 with 60 → Swap
After swapped: 75 60 95 88 70
Comparison: 3
Compare 95 with 88 → Swap
After swapped: 75 60 88 95 70
Comparison: 4
Compare 95 with 70 → Swap
After swapped: 75 60 88 70 95
Comparison: 5
Compare 75 with 60 → Swap
After swapped: 60 75 88 70 95
Comparison: 6
Compare 75 with 88 → No Swap
Comparison: 7
Compare 88 with 70 → Swap
After swapped: 60 75 70 88 95
Comparison: 8
Compare 60 with 75 → No Swap
Comparison: 9
Compare 75 with 70 → Swap
After swapped: 60 70 75 88 95
Comparison: 10
Compare 60 with 70 → No Swap


Sorted student marks: 60 70 75 88 95
Total comparisons: 10
Total swaps: 6
```

Improved Bubble Sort

Source code

```cpp
#include <iostream>
#include <vector>
using namespace std;

void improvedBubbleSort(vector<int>& arr, int& comparisons, int& swaps) {
    bool swapped;
    int n = arr.size();

    for (int pass = 1; pass < n; pass++) {
        swapped = false;

        for (int j = 0; j < n - pass; j++) {
            comparisons++;
            cout << "\nComparison: " << comparisons << endl;

            if (arr[j] > arr[j + 1]) {
                swaps++;
                cout << arr[j] << " swap with " << arr[j+1] << endl;

                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

                cout << "After swapped: ";
                for (int i = 0; i < arr.size(); i++) {
                    cout << arr[i] << " ";
                }
                cout << endl;

                swapped = true;
            }
            else {
                cout << "No swap\n";
            }
        }

        if (!swapped) {
            break;
        }
    }
}

void printArray(const vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

```

```cpp
int main() {

    vector<int> marks = {75, 95, 60, 88, 70};
    int comparisons = 0, swaps = 0;

    cout << "Original Array: ";
    for (int i = 0; i < marks.size(); i++) {
        cout << marks[i] << " ";
    }
    cout << endl;

    improvedBubbleSort(marks, comparisons, swaps);

    cout << "\nSorted student marks: ";
    printArray(marks);

    cout << "Total comparisons: " << comparisons << endl;
    cout << "Total swaps: " << swaps << endl;

    return 0;
}
```

Sample Output for Improved Bubble Sort

```
Original Array: 75 95 60 88 70
```

Comparison: 1

No swap

Comparison: 2

95 swap with 60

After swapped: 75 60 95 88 70

Comparison: 3

95 swap with 88

After swapped: 75 60 88 95 70

Comparison: 4

95 swap with 70

After swapped: 75 60 88 70 95

Comparison: 5

75 swap with 60

After swapped: 60 75 88 70 95

Comparison: 6

No swap

Comparison: 7

88 swap with 70

After swapped: 60 75 70 88 95

Comparison: 8

No swap

Comparison: 9

75 swap with 70

After swapped: 60 70 75 88 95

Comparison: 10

No swap

Sorted student marks: 60 70 75 88 95

Total comparisons: 10

Total swaps: 6

Selection Sort

Source code

```cpp
#include <iostream>
using namespace std;

void printArray(int array[], int n) {
    cout << "[";
    for (int i = 0; i < n; i++) {
        cout << array[i];
        if (i < n-1) cout << ",";
    }
    cout << "]";
}

void selectionSort(int array[], int n, int &comparisons, int &swaps) {
    comparisons = 0, swaps = 0;

    for (int i = n - 1; i > 0; i--) {
        cout << "(" << n-i << ") Find the largest element for position " <<
i << "." << endl;

        int max_idx = 0;
        for (int j = 1; j <= i; j++) {
            comparisons++;
            if (array[j] > array[max_idx]) {
                max_idx = j;
            }
        }

        cout << "Largest is " << array[max_idx] << ". Move " <<
array[max_idx]
             << " to the position " << i << "." << endl;
        cout << "Initial: ";
        printArray(array, n);
        cout << endl;

        if (max_idx != i) {
            swap(array[max_idx], array[i]);
            swaps++;
        }

        cout << "After comparing and move largest to position " << i << ":
";
        printArray(array, n);
        cout << "\n" << endl;
    }

    cout << "Sorted array: ";
    printArray(array, n);
    cout << endl;
    cout << "Total comparisons: " << comparisons << endl;
```

```
    cout << "Total swaps: " << swaps << endl;
}

int main() {
    int marks[] = {75,  95, 60, 88, 70};
    int n = sizeof(marks) / sizeof(marks[0]);
    int comparisons, swaps;
    cout << "Original Array: ";
    printArray(marks, n);
    cout << "\n\n";

    selectionSort(marks, n, comparisons, swaps);

    return 0;
}
```

Sample Output for Selection Sort

Original Array: [75,95,60,88,70]

(1) Find the largest element for **position 4.**
Largest is 95. **Move** 95 to the position 4.
Initial: [75,95,60,88,70]
After comparing and move largest to position 4: [75,70,60,88,**95**]

(2) Find the largest element for **position 3.**
Largest is 88. Move 88 to the position 3. **Remain unchanged.**
Initial: [75,70,60,88,95]
After comparing and move largest to position 3: [75,70,60,**88**,95]

(3) Find the largest element for **position 2.**
Largest is 75. **Move** 75 to the position 2.
Initial: [75,70,60,88,95]
After comparing and move largest to position 2: [60,70,**75**,88,95]

(4) Find the largest element for **position 1.**
Largest is 70. Move 70 to the position 1. **Remain unchanged.**
Initial: [60,70,75,88,95]
After comparing and move largest to position 1: [60,**70**,75,88,95]

Sorted array: [60,70,75,88,95]
Total comparisons: 10
Total swaps: 2

Insertion Sort

```cpp
#include <iostream>
using namespace std;

void insertionSort(int marks[], int n, int& comparisons, int& swaps) {
    comparisons = 0;
    swaps = 0;

    for (int i = 1; i < n; i++) {
        int key = marks[i];
        int j = i - 1;

        cout << "Inserting " << key << " into sorted portion...\n";

        // Compare and shift elements
        while (j >= 0) {
            comparisons++; // Increment for each comparison
            cout << "Comparison " << comparisons << ": " << marks[j] << " > " << key;
            if (marks[j] > key) {
                cout << " (Swap)\n";
                marks[j + 1] = marks[j];
                swaps++; // Increment for each swap
                cout << "Swapped " << marks[j] << " with " << key << "\n";
            } else {
                cout << " (No Swap)\n";
                break;
            }
            j--;

            // Print the current array state
            cout << "Current array: ";
            for (int k = 0; k < n; k++) {
                cout << marks[k] << " ";
            }
            cout << "\n";
            cout<<endl;
        }

        // Insert the key
        marks[j + 1] = key;

        // Print the current array state after inserting the key
        cout << "Current array after inserting " << key << ": ";
        for (int k = 0; k < n; k++) {
            cout << marks[k] << " ";
```

```
        }
        cout << "\n";
        cout<<endl;
    }
}

int main() {
    int marks[] = {75, 95, 60, 88, 70}; // Sample data
    int n = sizeof(marks) / sizeof(marks[0]);
    int comparisons = 0, swaps = 0;

    cout << "Original Marks: ";
    for (int i = 0; i < n; i++)
    {
        cout << marks[i] << " ";
    }
    cout << endl;

    insertionSort(marks, n, comparisons, swaps);

    cout << "Sorted Marks: ";
    for (int i = 0; i < n; i++)
    {
        cout << marks[i] << " ";
    }
    cout << endl;

    cout << "Total Comparisons: " << comparisons << endl;
    cout << "Total Swaps: " << swaps << endl;

    system("pause");

    return 0;
}
```

Sample Output for Insertion Sort

```
Original Marks: 75 95 60 88 70
Inserting 95 into sorted portion...
Comparison 1: 75 > 95 (No Swap)
Current array after inserting 95: 75 95 60 88 70

Inserting 60 into sorted portion...
Comparison 2: 95 > 60 (Swap)
Swapped 95 with 60
```

```
Current array: 75 95 95 88 70

Comparison 3: 75 > 60 (Swap)
Swapped 75 with 60
Current array: 75 75 95 88 70

Current array after inserting 60: 60 75 95 88 70

Inserting 88 into sorted portion...
Comparison 4: 95 > 88 (Swap)
Swapped 95 with 88
Current array: 60 75 95 95 70

Comparison 5: 75 > 88 (No Swap)
Current array after inserting 88: 60 75 88 95 70

Inserting 70 into sorted portion...
Comparison 6: 95 > 70 (Swap)
Swapped 95 with 70
Current array: 60 75 88 95 95

Comparison 7: 88 > 70 (Swap)
Swapped 88 with 70
Current array: 60 75 88 88 95

Comparison 8: 75 > 70 (Swap)
Swapped 75 with 70
Current array: 60 75 75 88 95

Comparison 9: 60 > 70 (No Swap)
Current array after inserting 70: 60 70 75 88 95

Sorted Marks: 60 70 75 88 95
Total Comparisons: 9
Total Swaps: 6
```

Result

Sorting Algorithm	Comparisons	Swaps
Bubble Sort	10	6
Improved Bubble Sort	10	6
Selection Sort	10	2
Insertion Sort	9	6

Discussion

1. Bubble Sort vs. Improved Bubble Sort

Both Bubble Sort and Improved Bubble Sort have 10 comparisons and 6 swaps. Improved Bubble Sort usually performs better than Bubble Sort since it can terminate early if the list is already sorted. However, based on this data, both had the same swaps and comparisons, indicating that early termination didn't occur in here. Therefore, the performance is the same.

2. Selection Sort

Selection sort used exactly 10 comparisons and 2 swaps for our list [75,95,60,88,70]. Compared to Bubble Sort which needed 6 swaps, selection sort makes fewer swaps since it only moves each maximum number once to its final position. However, selection sort is always needed to complete all its comparisons, even if the list is particularly sorted. It means that it is more predictable but not faster. In real-world problem, selection sort might work better with small list or when we want to minimise the number of swaps, but might not be the best choice for large set of numbers.

3. Insertion Sort

Given the list [75,95,60,88,70], Insertion Sort does 9 comparisons and 6 swaps. This is in contrast to Selection Sort, which, for a similar list, did 10 comparisons and 3 swaps. Insertion Sort therefore, adapts better to partially sorted data; it requires fewer comparisons. It involved, however, more swaps since it shifts lots of elements in order to insert a key into the correct position. This makes Insertion Sort more efficient in scenarios where the data is nearly sorted but less optimal when minimizing swaps is important. This makes Insertion Sort suitable for small sets of data or nearly sorted data. For larger data sets, it is less efficient. In this case, Insertion Sort delivered the sorted list [60,70,75,88,95], which demonstrates its flexibility and simplicity but it is less efficient when fewer movements of elements are needed in a particular scenario.

As a conclusion, this project proved to be very beneficial in understanding the various sorting algorithms like Bubble Sort, Improved Bubble Sort, Selection Sort, Insertion Sort, etc. Each one had its own set of advantages and disadvantages: Improved Bubble Sort is good for partially-sorted arrays, Selection Sort minimizes the number of swaps done, whereas Insertion Sort is easy to implement but not practical for large datasets. Evaluating the performance of the algorithms helped to learn more about the efficiency and application of the algorithms in the real world. All these activities served to sharpen our programming skills and the skill of picking the right algorithms for a given dataset context.

Reflection

[Teh Ru Qian]

I learned that the Improved Bubble Sort's main advantage is its ability to stop early if the array is partially sorted, even while it performs similarly to the Bubble Sort in terms of comparisons and swaps. In this exercise, the improved version didn't reach its full potential because each pass required exchanges. However, I now understand how it can be more efficient in real world cases with nearly sorted data.

[Lau Yan Kai]

After learning about Selection Sort, I now understand why it is such an interesting way to sort numbers. It reminds me of arranging students by height, finding the tallest person and put them at the end, then repeat until everyone is in order. When I tried it with the numbers in the assignment, I was surprised to see that it only needed 2 swaps, which is much less than other sorting methods like Bubble Sort. Selection sort might not the fastest way to sort big list of numbers, but I learnt that it is simple and reliable that make it perfect for smaller list. It taught me that sometimes the simplest solution can be the best choice of the job.

[Nurul Adriana]

Bubble sort was the sorting technique that I developed and evaluated whilst in this role. This activity enlightened me on the fundamentals of comparing and swapping neighbouring items. It worked when the datasets were small, however, it proved to be a time-consuming approach as more comparisons and swaps were involved especially for larger arrays. This helped in improving my programming skills as well as understanding the limits of algorithms.

[Dheshieghan A/L Saravana Moorthy]

When I learned insertion sort, I gained a deeper understanding of how sorting algorithms work. I realized that insertion sort builds a sorted array one element at a time by comparing and shifting elements. Although it has a time complexity of $O(n^2)$, which makes it inefficient for large datasets, I saw how it performs well on smaller or nearly sorted arrays. By tracking comparisons and swaps, I learned to appreciate how these operations affect the algorithm's efficiency. Overall, insertion sort helped me develop problem-solving skills and provided a foundation for understanding more complex algorithms.

Appendix

Bubble Sort

1. When **i = 0**, inner loop runs from **j = 0** to **j < n – i – 1 = 4**.
   - **swapped = false**.

j	Compare	Action	Comparisons	Swaps	Array State
0	75 > 95	No	1	0	[75, 95, 60, 88, 70]
1	95 > 60	Swap	2	1	[75, 60, 95, 88, 70]
2	95 > 88	Swap	3	2	[75, 60, 88, 95, 70]
3	95 > 70	Swap	4	3	[75, 60, 88, 70, 95]

   - **swapped = true**

2. When **i = 1**, inner loop runs from **j = 0** to **j < n – i -1 = 3**.
   - **swapped = false.**

j	Compare	Action	Comparisons	Swaps	Array State
0	75 > 60	Swap	5	4	[60, 75, 88, 70, 95]
1	75 >88	No	6	4	[60, 75, 88, 70, 95]
2	88 > 75	Swap	7	5	[60. 75, 70, 88, 95]

   - **swapped = true.**

3. When **i = 2**, inner loop runs from **j = 0** to **j < n – i -1 = 2**.
   - **swapped = false.**

j	Compare	Action	Comparisons	Swaps	Array State
0	60 > 75	No	8	5	[60, 75, 70, 88, 95]
1	75 > 70	Swap	9	6	[60, 70, 75, 88, 95]

   - **swapped = true.**

4. When **i = 3**, inner loop runs from **j = 0** to **j < n – i -1 = 1**.
   - **swapped = false.**

j	Compare	Action	Comparisons	Swaps	Array State
0	60 > 70	No	10	6	[60, 70, 75, 88, 95]

   - **swapped = false.**

Figure 1 shows tracking by Bubble Sort

Improved Bubble Sort

① When pass = 1, swapped = false, inner loop runs from j = 0 to j < n - pass = 4

j	Compare	Action	comparisons	swaps	
0	75 < 95	No	1	0	
1	95 > 60	Swap	2	1	[75, 60, 95, 88, 70]
2	95 > 88	Swap	3	2	[75, 60, 88, 95, 70]
3	95 > 70	Swap	4	3	[75, 60, 88, 70, 95]

swapped = true

② when pass = 2, swapped = false, inner loop runs from j = 0 to j < 3.

j	Compare	Action	comparisons	swaps	
0	75 > 60	Swap	5	4	[60, 75, 88, 70, 95]
1	75 < 88	No	6	4	
2	88 > 70	Swap	7	5	[60, 75, 70, 88, 95]

swapped = true

③ When pass = 3, swapped = false, inner loop runs from j = 0 to j < 2

j	Compare	Action	comparisons	swaps	
0	60 < 75	No	8	5	
1	75 > 70	Swap	9	6	[60, 70, 75, 88, 95]

swapped = true

④ When pass = 4, swapped = false, inner loop runs from j = 0 to j < 1

j	Compare	Action	comparisons	swaps
0	60 < 70	No	10	6

swapped = false

∴ [60, 70, 75, 88, 95]

Total Comparisons : 10

Total swaps : 6

Figure 2 shows tracking by Improved Bubble Sort

Selection sort

[75 , 95 , 60 , 88 , 70]

① when i = 0 . Find the largest element - 95 .

j	compare	action	comparison	swap	array state
0	95 > 75	swap	1	1	[75 , 95 , 60 , 88 , 70]
1	95 > 60	swap	2	1	[75 , 60 , 95 , 88 , 70]
2	95 > 88	swap	3	1	[75 , 88 , 60 , 95 , 70]
3	95 > 70	swap	4	1	[75 , 70 , 60 , 88 , 95]

② when i = 1 . Find the second largest element - 88 .

j	compare	action	comparison	swap	array state
0	88 > 70	No	5	1	[75 , 70 , 60 , 88 , 95]
1	88 > 75	No	6	1	[75 , 70 , 60 , 88 , 95]
2	88 > 60	No	7	1	[75 , 70 , 60 , 88 , 95]

③ when i = 2 . Find the third largest element - 75 .

j	compare	action	comparison	swap	array state
0	75 > 70	swap	8	2	[70 , 75 , 60 , 88 , 95]
1	75 > 60	swap	9	2	[60 , 70 , 75 , 88 , 95]

④ when i = 3 . Find the futh largest element - 70 .

j	compare	action	comparison	swap	array state
0	70 > 60	NO	10	2	[60 , 70 , 75 , 88 , 95]

Total comparisons : 10

Total swaps : 2

sorted array : [60 , 70 , 75 , 88 , 95]

Figure 3 shows tracking by Selection Sort

Subject: Date:

Insertion Sort

[75, 95, 60, 88, 70]

When i = 0, key = 95

j	compare	action	comparison	swap	array state
0	95 > 75	No	1	0	[75, 95, 60, 88, 70]

When i = 1, key = 60

j	compare	action	comparison	swap	array state
0	60 < 95	Swap	2	1	[75, 95, 95, 88, 70]
1	60 < 75	Swap	3	2	[75, 75, 95, 88, 70]
2	60 > 60	No	-	-	[60, 75, 95, 88, 70]

When i = 2, key = 88

j	compare	action	comparison	swap	array state
0	88 < 95	swap	4	3	[60, 75, 95, 95, 70]
1	88 > 75	No	5	-	[60, 75, 88, 95, 70]

When i = 3, key = 70

j	compare	action	comparison	swap	array state
0	70 < 95	Swap	6	4	[60, 75, 88, 95, 95]
1	70 < 88	Swap	7	5	[60, 75, 88, 88, 95]
2	70 < 75	Swap	8	6	[60, 75, 75, 88, 95]
3	70 > 60	No	9	-	[60, 70, 75, 88, 95]

Total comparison = 9

Total Swaps : 6

Sorted array : [60, 70, 75, 88, 95]

Figure 4 shows tracking by Insertion Sort