SECJ2013 2526-1 LAB
2

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

void bubbleSort(int data[], int listSize)
{
    // implement the bubble sort algorithm


} // end bubble sort

void iBubbleSort(int data[], int listSize)
{
    // implement the improved bubble sort algorithm


} // end improved bubble sort

void selectionSort(int data[], int n)
{
    // implement the selection sort algorithm

} // end selection sort
void swap(int& x, int& y)
{

} // end swap

void insertionSort(int data[])
{
    // implement the insertion sort algorithm

} // end insertion sort
int main(){
    int dataArrayA[25] = { };
    int dataArrayB[25] = { };

    // Implement your code to call bubble sort, improved bubble sort,
    // selection sort, and insertion sort functions seperately.


}
```

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

void bubbleSort(int data[], int listSize)
{
    // implement the bubble sort algorithm

    int pass, tempValue;

    int comparisons = 0;
    int swaps = 0;
    int totalPasses = 0;

    for (pass = 1; pass < listSize; pass++) {

        totalPasses++;

        // moves the largest element to the
        // end of the array

        for (int x = 0; x < listSize - pass; x++) {

            comparisons++;

            // compare adjacent elements
            if (data[x] > data[x + 1])
            { // swap elements

                swaps++;

                tempValue = data[x];
                data[x] = data[x + 1];
                data[x + 1] = tempValue;
            }
```

```
        }
    }

    cout << "\n--- Conventional Bubble Sort Results ---" << endl;
    cout << "Total Passes: " << totalPasses << endl;
    cout << "Total Comparisons: " << comparisons << endl;
    cout << "Total Swaps: " << swaps << endl;
    cout << "---------------------------------------" << endl;

} // end bubble sort

void iBubbleSort(int data[], int listSize) {
    int temp;
    bool sorted = false; //False when swaps occur

    int comparisons = 0;
    int swaps = 0;
    int totalPasses = 0;

    for (int pass = 1; (pass < listSize) && !sorted; ++pass) {

        totalPasses++;

        sorted = true; //Assume sorted

        for (int x = 0; x < listSize - pass; ++x) {

            comparisons++; //

            if (data[x] > data[x + 1]) {

                swaps++; //

                //Swap elements
                temp = data[x];
```

```
                data[x] = data[x + 1];
                data[x + 1] = temp;
                sorted = false; //A swap occurred
            } // end if
        } // end inner for
    } // end outer for loop

    // 2. CETAK HASIL KIRAAN
    cout << "\n--- Improved Bubble Sort Results ---" << endl;
    cout << "Total Passes: " << totalPasses << endl;
    cout << "Total Comparisons: " << comparisons << endl;
    cout << "Total Swaps: " << swaps << endl;
    cout << "---------------------------------------" << endl;

} // end improved bubble sort

void selectionSort(int data[], int n)
{
    int comparisons = 0;
    int swaps = 0;
    int totalPasses = 0;

    // implement the selection sort algorithm
    for (int last = n - 1; last >= 1; --last) {

        totalPasses++;

        // select largest item in the Array
        int largestIndex = 0;
        // largest item is assumed start at index 0
        for (int p = 1; p <= last; ++p) {

            comparisons++;

            if (data[p] > data[largestIndex])
```

```
                largestIndex = p;
        } // end for

        swaps++;
        swap(data[largestIndex], data[last]);

    } // end for

    cout << "\n--- Selection Sort Results ---" << endl;
    cout << "Total Passes: " << totalPasses << endl;
    cout << "Total Comparisons: " << comparisons << endl;
    cout << "Total Swaps: " << swaps << endl;
    cout << "----------------------------------------" << endl;

} // end selection sort

void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
} // end swap

void insertionSort(int data[])
{
    const int n = 25;

    int comparisons = 0;
    int swaps = 0;
    int totalPasses = 0;

// implement the insertion sort algorithm

    int item;
    int pass, insertIndex;
```

```
    for (pass=1; pass<n; pass++)
    {
        totalPasses++;

        item = data[pass];
        insertIndex = pass;

        if (insertIndex > 0) {
            comparisons++;
        }

        while((insertIndex > 0) && (data[insertIndex -1] > item))
        {
            data[insertIndex] = data[insertIndex -1];
            swaps++;
            insertIndex --;

            if (insertIndex > 0) {
                comparisons++;
            }
        }
        data[insertIndex] = item;
    } // end for

    // Output the counts
    cout << "\n--- Insertion Sort Results ---" << endl;
    cout << "Total Passes: " << totalPasses << endl;
    cout << "Total Comparisons: " << comparisons << endl;
    cout << "Total Swaps (Data Movements): " << swaps << endl;
    cout << "---------------------------------------" << endl;

} // end insertion sort

int main(){
int dataArrayA[25] = { }; int dataArrayB[25] = { };
```

```
// Implement your code to call bubble sort, improved bubble sort,
// selection sort, and insertion sort functions seperately.

    // Data 'Worst Case' from
    int dataArrayA[25] = {
        100, 50, 88, 30, 60, 45, 25, 12, 10, 5,
        98, 15, 65, 55, 45, 70, 20, 90, 66, 22,
        120, 48, 35, 85, 5
    };

    // Data 'Best Case' from
    int dataArrayB[25] = {
        5, 8, 30, 25, 35, 40, 42, 50, 55, 22,
        24, 66, 70, 75, 78, 80, 85, 90, 100, 118,
        98, 120, 122, 121, 121
    };

    int workingArray[25];
    const int SIZE = 25;

// Implement your code to call bubble sort, improved bubble sort,
// selection sort, and insertion sort functions seperately.

    cout << "=============================================" << endl;
    cout << "  ANALYSIS FOR WORST CASE (dataArrayA) " << endl;
    cout << "=============================================" << endl;

    // --- 1. Bubble Sort (Worst Case) ---
    memcpy(workingArray, dataArrayA, sizeof(dataArrayA));
    bubbleSort(workingArray, SIZE);

    // --- 2. Improved Bubble Sort (Worst Case) ---
    memcpy(workingArray, dataArrayA, sizeof(dataArrayA));
    iBubbleSort(workingArray, SIZE);
```

```
    // --- 3. Selection Sort (Worst Case) ---
    memcpy(workingArray, dataArrayA, sizeof(dataArrayA));
    selectionSort(workingArray, SIZE);

    // --- 4. Insertion Sort (Worst Case) ---
    memcpy(workingArray, dataArrayA, sizeof(dataArrayA));
    insertionSort(workingArray);



    cout << "\n=============================================" << endl;
    cout << "   ANALYSIS FOR BEST CASE (dataArrayB) " << endl;
    cout << "=============================================" << endl;

    // --- 1. Bubble Sort (Best Case) ---
    memcpy(workingArray, dataArrayB, sizeof(dataArrayB));
    bubbleSort(workingArray, SIZE);

    // --- 2. Improved Bubble Sort (Best Case) ---
    memcpy(workingArray, dataArrayB, sizeof(dataArrayB));
    iBubbleSort(workingArray, SIZE);

    // --- 3. Selection Sort (Best Case) ---
    memcpy(workingArray, dataArrayB, sizeof(dataArrayB));
    selectionSort(workingArray, SIZE);

    // --- 4. Insertion Sort (Best Case) ---
    memcpy(workingArray, dataArrayB, sizeof(dataArrayB));
    insertionSort(workingArray);

    system("pause");
    return 0;
}
```

1. Complete program above to sort 25 integers below into ascending order using simple sort algorithms: Bubble Sort, Insertion Sort and Selection Sort. For each sorting technique learned in class, modify the algorithms so that the programs are able to count and print the number of passes, the number of data comparisons and the number of data swapping that take place in the sorting process.

a. Run the programs on the following list of integer values. You can initialize the data in the array declarations.

int dataArrayA[25] = { };

int dataArrayB[25] = { };

100 50 88 30 60 45 25 12 10 5 98 15 65 55 45 70 20 90 66 22 120 48 35 85 5

dataArrayA

5 8 30 25 35 40 42 50 55 22 24 66 70 75 78 80 85 90 100 118 98 120 122 121 121

dataArrayB

b. dataArrayA is an example of a worst case data — totally unsorted list, while dataArrayB is an example of a best case data — almost sorted list. Analyze the output displayed from each sorting technique to find the total number of passes, number of data comparisons and number of data swapping. Make conclusions on which sorting technique is the best for the worst case data and which technique is the best for best case data. Which technique has performance that does not depend on the initial arrangement of data?

c. Fill in the following table to help you discuss the performance analysis.

Technique	Case	No of Comparisons	No of Swaps	No of Passes
Conventional Bubble Sort	Worst Case	300	171	24
	Best Case	300	19	24
Improved Bubble Sort	Worst Case	300	171	24
	Best Case	164	19	8
Selection Sort	Worst Case	300	24	24
	Best Case	300	24	24
Insertion Sort	Worst Case	195	171	24
	Best Case	43	19	24

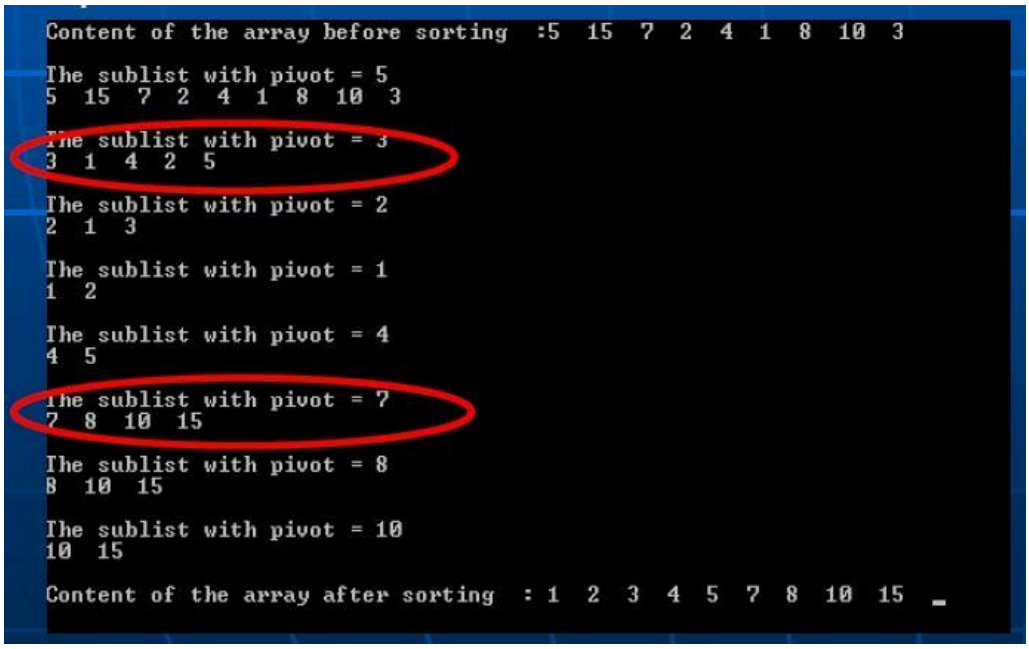
When dealing with the worst-case data (dataArrayA), the "best" technique really depends on what you're trying to optimize. If your main goal is to reduce the number of comparisons, then Insertion Sort was the most efficient. However, if your priority is to minimize the number of swaps (which can be a more costly operation), then Selection Sort was the clear winner, performing significantly fewer swaps than any other method.

For the best-case data (dataArrayB), which was an almost-sorted list, Insertion Sort was the undisputed champion. Its adaptive nature allowed it to recognize that the list was already in good order, resulting in a dramatically lower number of comparisons and swaps. This makes it an excellent choice for lists that are already partially sorted.

Finally, the one technique whose performance does not depend on the initial arrangement of data is Selection Sort. As the results clearly show, it performed the exact same number of passes, comparisons, and swaps for both the worst-case and best-case scenarios. It chugs along doing the same predictable amount of work, regardless of how messy or tidy the data is to begin with.

2. Write a C++ program that implements the Quick Sort algorithm to sort a list of integers [5 15 7 2 4 1 8 10 3].
   Your program must display the tracing output exactly in the format shown below, including:
   a) Unsorted data
   b) Content of partitioned sublists showing
      - pivot value
      - elements in the sublist
   c) Content of sublists processed inrecursive cals, showing
      - elements in the sublist
   d) Final sorted data

   The expected output format is illustrated as follows (your program must produce the same style of output from the slide):



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

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void printSublist(int arr[], int first, int last) {
    for (int i = first; i <= last; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int partition(int arr[], int first, int last) {
    int pivot = arr[first];
    int p = first;

    swap(arr[first], arr[last]);

    int i = first;

    for (int j = first; j < last; j++) {
        if (arr[j] <= pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }

    swap(arr[i], arr[last]);

    return i;
}

void quickSort(int arr[], int first, int last) {

    if (first < last) {

        cout << "The sublist with pivot = " << arr[first] << endl;
        printSublist(arr, first, last);
```

```
        int splitPoint = partition(arr, first, last);

        quickSort(arr, first, splitPoint - 1);

        quickSort(arr, splitPoint + 1, last);
    }

    else if (first <= last && first >= 0) {

        if (first == last || (last - first == 1) ) {

             if(arr[first] != arr[last]){
                cout << "The sublist with pivot = " << arr[first] << endl;
                printSublist(arr, first, last);
             }
        }
    }
}


int main() {
    int data[] = {5, 15, 7, 2, 4, 1, 8, 10, 3};
    int n = sizeof(data) / sizeof(data[0]);

    cout << "Content of the array before sorting: ";
    printSublist(data, 0, n - 1);

    quickSort(data, 0, n - 1);

    cout << "Content of the array after sorting: ";
    printSublist(data, 0, n - 1);

    system("pause");
    return 0;
}
```