

CS221-L Data Structures and Algorithms Lab



Lab # 07

Submitted by:

- Shayan Rizwan [2024585]

Submitted to: Sir Adnan Haider

Semester: 3rd

Faculty of Computer Science and Engineering
GIK Institute of Engineering Sciences and Technology

Task:

Write a C++ program that demonstrates the working of multiple sorting algorithms on integer arrays. The program must allow the user to choose a sorting technique from a menu and display the sorting progress after each pass or partition step.

You are required to implement the following sorting algorithms in C++:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort

Each sorting function must:

- Take an array of integers and its size as parameters.
- Modify the original array in-place.
- Return nothing.
- Display the array after each major step:
 - After each pass (Bubble, Insertion, Selection)
 - After each merge (Merge Sort)
 - After each partition (Quick Sort)

Code:

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

// The return type of all of the sorting functions must be set to void, as
they are to return nothing.

void DisplayArr(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
}

void BubbleSort(int arr[], int size) {
    int temp;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size - i - 1; j++) {
```

```

        if (arr[j] > arr[j+1]) {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }

    } // corresponds to the inner for loop
    // Display after each pass
    cout << "After pass " << i + 1 << ": ";
    DisplayArr(arr, size);
    cout << endl;
} // corresponds to the outer for loop
}

void InsertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        int key = arr[i];
        int j = i-1;

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

        arr[j+1] = key;
        // Display after each insertion
        cout << "After inserting element at position " << i << ": ";
        DisplayArr(arr, size);
        cout << endl;
    }
}

void SelectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        // int first = arr[i];
        // int swapVar = arr[i+1];
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[minIndex];
    }
}

```

```

        arr[minIndex] = temp;
        // Display after each selection
        cout << "After selection " << i + 1 << " (min = " << arr[i] << "): ";
        DisplayArr(arr, size);
        cout << endl;
    }
}

// Implementing the MergeSort function;

// l = left
// m = mid
// r = right

void Merge(int arr[], int l, int m, int r) {
    // Calculate sizes of two subarrays to be merged
    // n1 = size of left subarray (from left to mid inclusive)
    // n2 = size of right subarray (from mid+1 to right inclusive)
    int n1 = m - l + 1;
    int n2 = r - m;

    int left[n1];
    int right[n2];

    // Copy data from main array to temporary left subarray left[]
    // Copy elements from arr[l] to arr[m] into left[0] to left[n1-1]
    for (int i = 0; i < n1; i++)
        left[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        right[j] = arr[m + 1 + j];

    int i = 0; // Pointer for left subarray left[]
    int j = 0; // Pointer for right subarray right[]
    int k = l;

    // Compare elements from both subarrays and place smaller element in main
    array
    while (i < n1 && j < n2) {
        // Compare current elements of both subarrays
        if (left[i] <= right[j]) {
            // Left subarray element is smaller or equal
            arr[k] = left[i]; // Place left element in main array
            i++; // Move pointer in left subarray forward
        }
    }
}

```

```

    }
    else {
        // Right subarray element is smaller
        arr[k] = right[j]; // Place right element in main array
        j++; // Move pointer in right subarray forward
    }
    k++; // Always move main array pointer forward after each placement
}

// Copy the remaining elements of left[], if any
while (i < n1) {
    arr[k] = left[i];
    i++;
    k++;
}

// Copy the remaining elements of right[], if any
while (j < n2) {
    arr[k] = right[j];
    j++;
    k++;
}

// Display after each merge
cout << "After merging [" << l << "-" << m << "] and [" << m+1 << "-" << r
<< "]: ";
DisplayArr(arr + l, r - l + 1);
cout << endl;
}

void MergeSort(int arr[], int l, int r) {
    if (l < r) { // Check if the smaller sub array has one element only or not;
        (if FALSE, recursive calls stop)
        int m = (l + r)/2;
        MergeSort(arr, l, m);
        MergeSort(arr, m+1, r);
        Merge(arr, l, m, r);
    }
}

// Partition function: rearranges array and places pivot in correct position
// All elements smaller than pivot go to left, larger to right
// Returns the final index of pivot element

```

```

int partition(int arr[], int low, int high) {
    // Choose the pivot element - using last element (arr[high])
    // Pivot selection strategy affects performance but last element is common
    int pivot = arr[high];

    // Index of smaller element - indicates the right boundary of elements <
    // pivot
    // Initialized to (low - 1) because we haven't found any smaller elements
    // yet
    // i will always point to the last element that is smaller than pivot
    int i = low - 1;

    // Traverse through all elements from low to high-1 (exclude pivot at high)
    // Compare each element with pivot and rearrange accordingly
    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than pivot
        if (arr[j] < pivot) {
            i++; // Move boundary of smaller elements forward
            // Move smaller element to left partition
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            // After swap: arr[i] contains element < pivot, arr[j] contains what was
            // at arr[i]
        }
        // If arr[j] >= pivot, do nothing - it will stay in right partition
    }

    // After loop:
    // - Elements from low to i are all < pivot
    // - Elements from i+1 to high-1 are all >= pivot
    // - arr[high] still contains pivot

    // Place pivot in its correct final position
    // i+1 is the position where pivot should be placed
    // All elements before i+1 are smaller, all after are larger or equal
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    // Display after partition
    cout << "After partition (pivot " << pivot << " at position " << i + 1 <<
    "): ";
    DisplayArr(arr, high - low + 1);
}

```

```

    cout << endl;

    // Return pivot's final position - this index divides the array into two
    parts
    return i + 1;
}

// Main QuickSort recursive function
// Uses divide-and-conquer strategy to sort the array
void QuickSort(int arr[], int low, int high) {
    // Base case: if low >= high, subarray has 0 or 1 element (already sorted)
    // Recursion stops when subarray size becomes 1 or empty
    if (low < high) {
        // Display current subarray being processed
        cout << "Processing subarray: ";
        DisplayArr(arr + low, high - low + 1);
        cout << endl;

        // Partition the array and get pivot index (pi)
        // After partition:
        // - arr[pi] is in its final sorted position
        // - All elements left of pi are smaller
        // - All elements right of pi are larger
        int pi = partition(arr, low, high);

        // Recursively sort elements before partition (left subarray)
        // Elements from low to pi-1 are all smaller than pivot but unsorted
        QuickSort(arr, low, pi - 1);

        // Recursively sort elements after partition (right subarray)
        // Elements from pi+1 to high are all larger than pivot but unsorted
        QuickSort(arr, pi + 1, high);

        // Note: We don't sort pi because it's already in correct position
    }
}

void DisplayInterface() {
    int arr[5] = {9, 4, 7, 1, 3};
    int size = 5;
    cout << "1. Bubble Sort" << endl;
    cout << "2. Insertion Sort" << endl;
}

```

```

cout << "3. Selection Sort" << endl;
cout << "4. Merge Sort" << endl;
cout << "5. Quick Sort" << endl;
cout << "6. Exit" << endl;
cout << "Enter your choice = ";
int userChoice;
cin >> userChoice;
switch (userChoice) {
    case 1:
        cout << "BEFORE: " << endl;
        DisplayArr(arr, size);

        BubbleSort(arr, size);

        cout << "AFTER: " << endl;
        DisplayArr(arr, size);
        break;

    case 2:
        cout << "BEFORE: " << endl;
        DisplayArr(arr, size);

        InsertionSort(arr, size);

        cout << "AFTER: " << endl;
        DisplayArr(arr, size);
        break;

    case 3:
        cout << "BEFORE: " << endl;
        DisplayArr(arr, size);

        SelectionSort(arr, size);

        cout << "AFTER: " << endl;
        DisplayArr(arr, size);
        break;

    case 4:
        cout << "BEFORE: " << endl;
        DisplayArr(arr, size);

        MergeSort(arr, 0, size - 1);

```

```

        cout << "AFTER: " << endl;
        DisplayArr(arr, size);
        break;

    case 5:
        cout << "BEFORE: " << endl;
        DisplayArr(arr, size);

        QuickSort(arr, 0, size - 1);

        cout << "AFTER: " << endl;
        DisplayArr(arr, size);
        break;

    case 6:
        cout << "Thank you for using this Sorting Program!" << endl;
        exit(0);
        break;
    }
}

int main() {

    DisplayInterface();

    return 0;
}

```

Output:

```
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Quick Sort
6. Exit
Enter your choice = 3
BEFORE:
9 4 7 1 3 After selection 1 (min = 1): 1 4 7 9 3
After selection 2 (min = 3): 1 3 7 9 4
After selection 3 (min = 4): 1 3 4 9 7
After selection 4 (min = 7): 1 3 4 7 9
AFTER:
1 3 4 7 9
```