# Department of Computing

# School of Electrical Engineering and Computer Science

**CS-250: Data Structure and Algorithms**

**Class: BESE 13A**

# 

# Lab 7: Implementation of Sorting Algorithms and Complexity Analysis

**Date: 3rd November, 2023**

**Time: 10 am - 12 pm**

# Lab Instructor: Anum Asif

# 

# Lab 7: Implementation of Sorting Algorithms and Complexity Analysis

**Lab Tasks**

**Task 1:**

Implement Bubble sort, Selection sort, Insertion sort and Merge sort algorithms in C++.

**Task 2 (average case complexity):**

The next step is to compare the two algorithms. Generate arrays of random numbers in the range 1 to 100 with sizes 100, 1000, 10000, 100000, and 1000000. Compare the running times of the three algorithms on each array. How do they compare? Are the results what you expected, and why? Answer the questions in the solution section.

**Task 3 (best and worst case complexity):**

Now sort the arrays using stl::sort, once in ascending order and then in descending order. Given both sorted arrays as inputs to all three algorithms and compute their running time. The running time of which algorithm shows most variations based on the structure of the input and why? Answer the questions in the solution section.

**Solution:**

|  |
| --- |
| Solution |
| Task 1:    Task 2 :    Task 3:    CPP File: #include <iostream>  #include <algorithm>  #include <ctime>  #include <cstdlib>  #include <functional>  using namespace std;  class Sorter {  public:      // Function to perform the Bubble Sort algorithm      void bubbleSort(int arr[], int n) {          for (int i = 0; i < n - 1; i++) {              for (int j = 0; j < n - 1 - i; j++) {                  if (arr[j] > arr[j + 1]) {                      std::swap(arr[j], arr[j + 1]);                  }              }          }      }      // Function to perform the Insertion Sort algorithm      void insertionSort(int arr[], int n) {          for (int i = 1; i < n; i++) {              int j = i;              while (j > 0 && arr[j - 1] > arr[j]) {                  std::swap(arr[j - 1], arr[j]);                  j--;              }          }      }      // Function to perform the Selection Sort algorithm      void selectionSort(int arr[], int n) {          for (int i = 0; i < n - 1; i++) {              int minIndex = i;              for (int j = i + 1; j < n; j++) {                  if (arr[j] < arr[minIndex]) {                      minIndex = j;                  }              }              std::swap(arr[i], arr[minIndex]);          }      }      // Function to merge two halves of an array      void merge(int arr[], int left, int middle, int right) {          int n1 = middle - left + 1;          int n2 = right - middle;          int L[n1];          int R[n2];          for (int i = 0; i < n1; i++) {              L[i] = arr[left + i];          }          for (int j = 0; j < n2; j++) {              R[j] = arr[middle + 1 + j];          }          int 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++;          }      }      // Function to perform the Merge Sort algorithm      void mergeSort(int arr[], int left, int right) {          if (left < right) {              int middle = left + (right - left) / 2;              mergeSort(arr, left, middle);              mergeSort(arr, middle + 1, right);              merge(arr, left, middle, right);          }      }      // Function to display an array      void displayArray(int arr[], int n) {          for (int i = 0; i < n; i++) {              std::cout << arr[i] << " ";          }          std::cout << std::endl;      }  };  int main() {      // Array 1      int a1[10] = {5, 9, 1, 3, 7, 2, 8, 4, 6, 10};      // Array 2      int a2[10] = {12, 45, 23, 9, 3, 31, 7, 19, 2, 8};      // Array 3      int a3[10] = {55, 12, 88, 17, 6, 33, 21, 4, 9, 28};      // Array 4      int a4[10] = {37, 14, 26, 59, 8, 43, 11, 31, 19, 67};      Sorter s;      s.bubbleSort(a1, 10);      cout<<"\nThe Result of bubble sort is:";      s.displayArray(a1, 10);      s.insertionSort(a2, 10);      cout<<"\nThe result of insertion sort is: ";      s.displayArray(a2, 10);      s.selectionSort(a3, 10);      cout<<"\nThe result of selection sort is: ";      s.displayArray(a3, 10);      cout<<"\nThe result of merge sort is:\n";      s.mergeSort(a4,0,9);      s.displayArray(a4,10);  cout<<"\n\n";      // Task 2: Measure and compare the running times of sorting algorithms on random arrays      int sizes[] = {100, 1000, 10000, 100000};      for (int size : sizes) {          int\* randomArray = new int[size];          // Fill the array with random numbers          for (int i = 0; i < size; i++) {              randomArray[i] = std::rand() % 100 + 1;          }          // Start the timer for Bubble Sort          std::clock\_t start\_time = std::clock();          // Test and measure Bubble Sort          s.bubbleSort(randomArray, size);          // Stop the timer          std::clock\_t end\_time = std::clock();          // Calculate elapsed time          double elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;          std::cout << "Bubble Sort for " << size << " elements took " << elapsed\_time << " seconds\n";          // Reset the array          std::random\_shuffle(randomArray, randomArray + size);          // Start the timer for Insertion Sort          start\_time = std::clock();          // Test and measure Insertion Sort          s.insertionSort(randomArray, size);          // Stop the timer          end\_time = std::clock();          // Calculate elapsed time          elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;          std::cout << "Insertion Sort for " << size << " elements took " << elapsed\_time << " seconds\n";          // Reset the array          std::random\_shuffle(randomArray, randomArray + size);          // Start the timer for Selection Sort          start\_time = std::clock();          // Test and measure Selection Sort          s.selectionSort(randomArray, size);          // Stop the timer          end\_time = std::clock();          // Calculate elapsed time          elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;          std::cout << "Selection Sort for " << size << " elements took " << elapsed\_time << " seconds\n";          // Reset the array          std::random\_shuffle(randomArray, randomArray + size);          // Start the timer for Merge Sort          start\_time = std::clock();          // Test and measure Merge Sort          s.mergeSort(randomArray, 0, size - 1);          // Stop the timer          end\_time = std::clock();          // Calculate elapsed time          elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;          std::cout << "Merge Sort for " << size << " elements took " << elapsed\_time << " seconds\n";          delete[] randomArray;      }      // Task 3: Sort the arrays using std::sort in ascending and descending order      // and then test sorting algorithms on these sorted arrays.      int array1[10] = {5, 9, 1, 3, 7, 2, 8, 4, 6, 10};      int array2[10] = {12, 45, 23, 9, 3, 31, 7, 19, 2, 8};      // Task 3: Best Case - Ascending Order      std::sort(std::begin(array1), std::end(array1));      // Start the timer for Bubble Sort on the sorted array      std::clock\_t start\_time = std::clock();      // Test and measure Bubble Sort      s.bubbleSort(array1, 10);      // Stop the timer      std::clock\_t end\_time = std::clock();      // Calculate elapsed time      double elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Bubble Sort on the sorted array (Best Case) took " << elapsed\_time << " seconds\n";      // Reset the array      std::copy(std::begin(array1), std::end(array1), std::begin(array1));      // Start the timer for Insertion Sort on the sorted array      start\_time = std::clock();      // Test and measure Insertion Sort      s.insertionSort(array1, 10);      // Stop the timer      end\_time = std::clock();      // Calculate elapsed time      elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Insertion Sort on the sorted array (Best Case) took " << elapsed\_time << " seconds\n";      // Reset the array      std::copy(std::begin(array1), std::end(array1), std::begin(array1));      // Start the timer for Selection Sort on the sorted array      start\_time = std::clock();      // Test and measure Selection Sort      s.selectionSort(array1, 10);      // Stop the timer      end\_time = std::clock();      // Calculate elapsed time      elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Selection Sort on the sorted array (Best Case) took " << elapsed\_time << " seconds\n";      // Start the timer for Merge Sort on the sorted array      start\_time = std::clock();      // Test and measure Merge Sort      s.mergeSort(array1, 0, 9);      // Stop the timer      end\_time = std::clock();      // Calculate elapsed time      elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Merge Sort on the sorted array (Best Case) took " << elapsed\_time << " seconds\n";      // Task 3: Worst Case - Descending Order      std::sort(std::begin(array2), std::end(array2), std::greater<int>());      // Start the timer for Bubble Sort on the sorted array      start\_time = std::clock();      // Test and measure Bubble Sort      s.bubbleSort(array2, 10);      // Stop the timer      end\_time = std::clock();      // Calculate elapsed time      elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Bubble Sort on the sorted array (Worst Case) took " << elapsed\_time << " seconds\n";      // Reset the array      std::copy(std::begin(array2), std::end(array2), std::begin(array2));      // Start the timer for Insertion Sort on the sorted array      start\_time = std::clock();      // Test and measure Insertion Sort      s.insertionSort(array2, 10);      // Stop the timer      end\_time = std::clock();      // Calculate elapsed time      elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Insertion Sort on the sorted array (Worst Case) took " << elapsed\_time << " seconds\n";      // Reset the array      std::copy(std::begin(array2), std::end(array2), std::begin(array2));      // Start the timer for Selection Sort on the sorted array      start\_time = std::clock();      // Test and measure Selection Sort      s.selectionSort(array2, 10);      // Stop the timer      end\_time = std::clock();      // Calculate elapsed time      elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Selection Sort on the sorted array (Worst Case) took " << elapsed\_time << " seconds\n";      // Start the timer for Merge Sort on the sorted array      start\_time = std::clock();      // Test and measure Merge Sort      s.mergeSort(array2, 0, 9);      // Stop the timer      end\_time = std::clock();      // Calculate elapsed time      elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;      std::cout << "Merge Sort on the sorted array (Worst Case) took " << elapsed\_time << " seconds\n";      return 0;  } |