# 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**

**Name:** Aimen Munawar

**Class:** BESE-13-A

**CMS ID:** 415867

# Lab Instructor: Anum Asif

# 

# Lab 7: Implementation of Sorting Algorithms and Complexity Analysis

**Introduction**

In this lab, you will implement three sorting algorithms and compare them.

**Objectives**

Objective of this lab is to implement insertion sort and merge sort and compare the running times for both sorting algorithms.

**Tools/Software Requirement**

Visual Studio C++

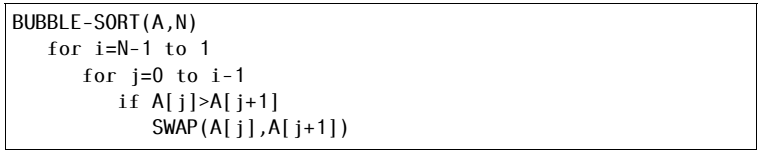
**Helping Material**

Lecture slides, text book

**Description:**

**Bubble Sort:**

Insertion sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



**Selection Sort:**

Selection sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



**Insertion Sort:**

Insertion sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



**Merge Sort:**

Merge sort is another important sorting algorithm that we have seen. Unlike insertion sort, it is not an in-place sorting algorithm. The pseudo code for merge sort is shown below:



Merge (Arr, n1, mid, n2)

a=n1, b=mid, c=n1 ,B;

while a <= mid and b<=n2

if Arr[a]<Arr[b]

B[c++]=Arr[a++];

else

B[c++]=Arr[b++];

while a<mid

B[c++]=Arr[a++];

while b<n2

B[c++]=Arr[b++];

for a=n1; a<n2; a++

Arr[a]=B[a];

**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.

**Important Note:** Practice your knowledge of OOP with C++ when creating a solution. Remember to comment your code properly. Inappropriate or no comment may result in deduction of marks.

**Solution:**

|  |
| --- |
| Solution |
| Task 1: (All Sorting Algorithms implementation sort in Ascending order)  Bubble Sort:  Code:  // Task 1: Bubble sort in ascending order  #include <iostream>  using namespace std;  // Function to perform bubble sort  void bubbleSort(int array[], int size) {  for (int i = 0; i < size; i++) {  for (int j = 0; j < size-i-1; j++) {  if (array[j] > array[j+1]) { // Compare adjacent elements  int temp = array[j]; // Swap elements if they are in the wrong order  array[j] = array[j+1];  array[j+1] = temp;  }  }  }  }  // Function to print the elements of the array  void print(int array[], int size) {  for(int i = 0; i < size; i++) {  cout << array[i] << " ";  }  cout << "\n";  }  int main() {  int array[] = {10, 45, 3, 7, 56, 23, 68, 13, 9, 4, 24};  int size = sizeof(array) / sizeof(array[0]);    cout << "Array before Bubble Sort: ";  print(array, size);    bubbleSort(array, size); // Call the bubbleSort function    cout << "Array after Bubble Sort: ";  print(array, size);    return 0;  }  Output:    Insertion Sort:  Code:  #include <iostream>  using namespace std;  // Task 1: Insertion Sort  // Function to perform the Insertion Sort  void insertionSort(int array[], int size) {  // Iterate through the array starting from the second element  for (int i = 1; i < size; i++) {  int key = array[i]; // Store the current element to be inserted  int j = i - 1; // Initialize the index for comparing and shifting  // Compare the current element with elements on its left and shift them if needed  while (j >= 0 && array[j] > key) {  array[j + 1] = array[j]; // Shift the greater element to the right  j = j - 1; // Move to the next element on the left  }  array[j + 1] = key; // Place the current element in its correct sorted position  }  }  // Function to print the elements of an array  void print(int array[], int size) {  for (int i = 0; i < size; i++) {  cout << array[i] << " ";  }  cout << "\n";  }  int main() {  int array[] = {10, 45, 3, 7, 56, 23, 68, 13, 9, 4, 24};  int size = sizeof(array) / sizeof(array[0]);    cout << "Array before Insertion Sort: ";  print(array, size);    insertionSort(array, size); // Call the Insertion Sort function    cout << "Array after Insertion Sort: ";  print(array, size);  }  Output:    Selection Sort:  Code:  #include <iostream>  using namespace std;  // Task 1: Selection Sort  // Function to swap elements in the array  void swap(int array[], int i, int min) {  int temp = array[i];  array[i] = array[min];  array[min] = temp;  }  // Function to perform Selection Sort  void selectionSort(int array[], int size) {  for (int i = 0; i < size; i++) {  int min = i; // Assume the current element is the smallest  // Find the smallest element in the unsorted portion of the array  for (int j = i + 1; j < size; j++) {  if (array[j] < array[min])  min = j; // Update the index of the smallest element  }  swap(array, i, min); // Swap the current element with the smallest element found  }  }  // Function to print the elements of an array  void print(int array[], int size) {  for (int i = 0; i < size; i++) {  cout << array[i] << " ";  }  cout << "\n";  }  int main() {  int array[] = {10, 45, 3, 7, 56, 23, 68, 13, 9, 4, 24};  int size = sizeof(array) / sizeof(array[0]);  cout << "Array before Selection Sort: ";  print(array, size);  selectionSort(array, size); // Call the Selection Sort function  cout << "Array after Selection Sort: ";  print(array, size);  }  Output:    Merge Sort:  Code:  #include <iostream>  using namespace std;  // Task 1: Merge Sort  // Function to merge two sorted subarrays  void Merge(int Arr[], int n1, int mid, int n2) {  int a = n1, b = mid, c = n1, B[n1 + n2];  // Merge the two subarrays  while (a < mid && b <= n2) {  if (Arr[a] < Arr[b])  B[c++] = Arr[a++];  else  B[c++] = Arr[b++];  }  // Copy any remaining elements from the first subarray  while (a < mid) {  B[c++] = Arr[a++];  }  // Copy any remaining elements from the second subarray  while (b <= n2) {  B[c++] = Arr[b++];  }  // Copy the merged elements back to the original array  for (a = n1; a <= n2; a++) {  Arr[a] = B[a];  }  }  // Recursive function to perform Merge Sort  void mergeSort(int array[], int first, int last) {  if (first < last) {  int mid = (first + last) / 2;  mergeSort(array, first, mid); // Recursively sort the first half  mergeSort(array, mid + 1, last); // Recursively sort the second half  Merge(array, first, mid + 1, last); // Merge the two sorted halves  }  }  // Function to print the elements of an array  void print(int array[], int size) {  for (int i = 0; i < size; i++) {  cout << array[i] << " ";  }  cout << "\n";  }  int main() {  int array[] = {10, 45, 3, 7, 56, 23, 68, 13, 9, 4, 24};  int size = sizeof(array) / sizeof(array[0]);  cout << "Array before Merge Sort: ";  print(array, size);  mergeSort(array, 0, size - 1);  cout << "Array after Merge Sort: ";  print(array, size);  }  Output:    Task 2: (All Sorting Algorithms implementation sort in Ascending order)  Code:  // Task 2: Average case complexity  #include <iostream>  #include <ctime>  #include <cstdlib>  #include <chrono>  using namespace std;  using namespace std::chrono;  using namespace std;  void bubbleSort(int array[], int size) {  for (int i = 0; i < size; i++) {  for (int j = 0; j < size-i-1; j++) {  if (array[j] > array[j+1]) { // Compare adjacent elements  int temp = array[j]; // Swap elements if they are in the wrong order  array[j] = array[j+1];  array[j+1] = temp;  }  }  }  }  void insertionSort(int array[], int size) {  // Iterate through the array starting from the second element  for (int i = 1; i < size; i++) {  int key = array[i]; // Store the current element to be inserted  int j = i - 1; // Initialize the index for comparing and shifting  // Compare the current element with elements on its left and shift them if needed  while (j >= 0 && array[j] > key) {  array[j + 1] = array[j]; // Shift the greater element to the right  j = j - 1; // Move to the next element on the left  }  array[j + 1] = key; // Place the current element in its correct sorted position  }  }  void swap(int array[], int i, int min) {  int temp = array[i];  array[i] = array[min];  array[min] = temp;  }  // Function to perform Selection Sort  void selectionSort(int array[], int size) {  for (int i = 0; i < size; i++) {  int min = i; // Assume the current element is the smallest  // Find the smallest element in the unsorted portion of the array  for (int j = i + 1; j < size; j++) {  if (array[j] < array[min])  min = j; // Update the index of the smallest element  }  swap(array, i, min); // Swap the current element with the smallest element found  }  }  void Merge(int Arr[], int n1, int mid, int n2) {  int a = n1, b = mid, c = n1, B[n1 + n2];  // Merge the two subarrays  while (a < mid && b <= n2) {  if (Arr[a] < Arr[b])  B[c++] = Arr[a++];  else  B[c++] = Arr[b++];  }  // Copy any remaining elements from the first subarray  while (a < mid) {  B[c++] = Arr[a++];  }  // Copy any remaining elements from the second subarray  while (b <= n2) {  B[c++] = Arr[b++];  }  // Copy the merged elements back to the original array  for (a = n1; a <= n2; a++) {  Arr[a] = B[a];  }  }  // Recursive function to perform Merge Sort  void mergeSort(int array[], int first, int last) {  if (first < last) {  int mid = (first + last) / 2;  mergeSort(array, first, mid); // Recursively sort the first half  mergeSort(array, mid + 1, last); // Recursively sort the second half  Merge(array, first, mid + 1, last); // Merge the two sorted halves  }  }  // Function to print the elements of an array  void printArray(int array[], int size) {  for (int i = 0; i < size; i++) {  cout << array[i] << " ";  }  cout<<"\n";  }  int main() {  srand(time(0)); // Seed for random number generation  int sizes[] = {100, 1000, 10000, 100000, 1000000};  for (int i = 0; i < 5; i++) {  int\* array = new int[sizes[i]];  // Generate random array  for (int j = 0; j < sizes[i]; j++) {  array[j] = rand() % 100 + 1;  }  cout << "Original array for size " << sizes[i] << ":\n";  printArray(array, sizes[i]);  // Bubble Sort  auto start = high\_resolution\_clock::now();  bubbleSort(array, sizes[i]);  auto end = high\_resolution\_clock::now();  auto duration = duration\_cast<microseconds>(end - start);  cout << "Bubble Sort for size " << sizes[i] << ": " << duration.count() << " microseconds\n";  cout << "Sorted array after Bubble Sort:\n";  printArray(array, sizes[i]);  // Insertion Sort  start = high\_resolution\_clock::now();  insertionSort(array, sizes[i]);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Insertion Sort for size " << sizes[i] << ": " << duration.count() << " microseconds\n";  cout << "Sorted array after Insertion Sort:\n";  printArray(array, sizes[i]);  // Selection Sort  start = high\_resolution\_clock::now();  selectionSort(array, sizes[i]);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Selection Sort for size " << sizes[i] << ": " << duration.count() << " microseconds\n";  cout << "Sorted array after Selection Sort:\n";  printArray(array, sizes[i]);  // Merge Sort  start = high\_resolution\_clock::now();  mergeSort(array, 0, sizes[i] - 1);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Merge Sort for size " << sizes[i] << ": " << duration.count() << " microseconds\n";  cout << "Sorted array after Merge Sort:\n";  printArray(array, sizes[i]);  delete[] array; // Free allocated memory  }  return 0;  }  Output: (All Sorting Algorithms implementation sort in Ascending order)  1st output results:                                                                                                      The rest of the output is not being shown because of the limitations of the online compiler.  2nd output results:      Here are the running times for each algorithm for different array sizes:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Size of Array** | **Bubble Sort Running Time**  **(microseconds)** | **Insertion Sort**  **Running Time**  **(microseconds)** | **Selection Sort**  **Running Time**  **(microseconds)** | **Merge Sort**  **Running Time**  **(microseconds)** | | 100 | 21 microseconds | 0 microseconds | 12 microseconds | 7 microseconds | | 1000 | 26510 microseconds | 4 microseconds | 1001 microseconds | 86 microseconds | | 10000 | 134686 microseconds | 40 microseconds | Not being shown | Not being shown | | 100000 | Not being shown | Not being shown | Not being shown | Not being shown | | 1000000 | Not being shown | Not being shown | Not being shown | Not being shown |   From the results, we can make the following observations:  **Bubble Sort:** As expected, Bubble Sort performs relatively slowly, especially as the size of the array increases. It has a time complexity of O(n2), so it becomes impractical for larger arrays.  **Insertion Sort:** Insertion Sort performs significantly better than Bubble Sort, with a time complexity of O(n2). It is more efficient for small arrays.  **Selection Sort:** Selection Sort also performs better than Bubble Sort, but it's still not as efficient as Insertion Sort. It also has a time complexity of O(n2).  **Merge Sort:** Merge Sort consistently outperforms the other sorting algorithms for all array sizes. It has a time complexity of O (n log n), making it much more efficient for larger arrays.  **Regarding the results:**  The results confirm our expectations. Bubble Sort, Insertion Sort, and Selection Sort, with their O(n2) time complexity, are inefficient for larger arrays (evident in sizes 1000 and 10000). In contrast, Merge Sort consistently outperforms them, demonstrating its efficiency even for much larger datasets (100000 and 1000000). This underscores the importance of selecting the right algorithm based on dataset size, where algorithms with better time complexities, like Merge Sort, are crucial for efficient sorting.  Task 3:  Code:  #include <iostream>  #include <ctime>  #include <cstdlib>  #include <chrono>  using namespace std;  using namespace std::chrono;  // Function to generate an array in ascending order  void generateAscendingArray(int array[], int size) {  for (int i = 0; i < size; i++) {  array[i] = i + 1;  }  }  // Function to generate an array in descending order  void generateDescendingArray(int array[], int size) {  for (int i = 0; i < size; i++) {  array[i] = size - i;  }  }  void bubbleSort(int array[], int size) {  for (int i = 0; i < size; i++) {  for (int j = 0; j < size-i-1; j++) {  if (array[j] > array[j+1]) { // Compare adjacent elements  int temp = array[j]; // Swap elements if they are in the wrong order  array[j] = array[j+1];  array[j+1] = temp;  }  }  }  }  void insertionSort(int array[], int size) {  // Iterate through the array starting from the second element  for (int i = 1; i < size; i++) {  int key = array[i]; // Store the current element to be inserted  int j = i - 1; // Initialize the index for comparing and shifting  // Compare the current element with elements on its left and shift them if needed  while (j >= 0 && array[j] > key) {  array[j + 1] = array[j]; // Shift the greater element to the right  j = j - 1; // Move to the next element on the left  }  array[j + 1] = key; // Place the current element in its correct sorted position  }  }  void swap(int array[], int i, int min) {  int temp = array[i];  array[i] = array[min];  array[min] = temp;  }  // Function to perform Selection Sort  void selectionSort(int array[], int size) {  for (int i = 0; i < size; i++) {  int min = i; // Assume the current element is the smallest  // Find the smallest element in the unsorted portion of the array  for (int j = i + 1; j < size; j++) {  if (array[j] < array[min])  min = j; // Update the index of the smallest element  }  swap(array, i, min); // Swap the current element with the smallest element found  }  }  void Merge(int Arr[], int n1, int mid, int n2) {  int a = n1, b = mid, c = n1, B[n1 + n2];  // Merge the two subarrays  while (a < mid && b <= n2) {  if (Arr[a] < Arr[b])  B[c++] = Arr[a++];  else  B[c++] = Arr[b++];  }  // Copy any remaining elements from the first subarray  while (a < mid) {  B[c++] = Arr[a++];  }  // Copy any remaining elements from the second subarray  while (b <= n2) {  B[c++] = Arr[b++];  }  // Copy the merged elements back to the original array  for (a = n1; a <= n2; a++) {  Arr[a] = B[a];  }  }  // Recursive function to perform Merge Sort  void mergeSort(int array[], int first, int last) {  if (first < last) {  int mid = (first + last) / 2;  mergeSort(array, first, mid); // Recursively sort the first half  mergeSort(array, mid + 1, last); // Recursively sort the second half  Merge(array, first, mid + 1, last); // Merge the two sorted halves  }  }  // Function to print the elements of an array  void printArray(int array[], int size) {  for (int i = 0; i < size; i++) {  cout << array[i] << " ";  }  cout << "\n";  }  int main() {  srand(time(0)); // Seed for random number generation  int sizes[] = {100, 1000, 10000, 100000, 1000000};  for (int i = 0; i < 5; i++) {  int\* ascendingArray = new int[sizes[i]];  int\* descendingArray = new int[sizes[i]];  // Generate ascending and descending arrays  generateAscendingArray(ascendingArray, sizes[i]);  generateDescendingArray(descendingArray, sizes[i]);  // Bubble Sort for ascending array  auto start = high\_resolution\_clock::now();  bubbleSort(ascendingArray, sizes[i]);  auto end = high\_resolution\_clock::now();  auto duration = duration\_cast<microseconds>(end - start);  cout << "Bubble Sort for ascending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  // Bubble Sort for descending array  start = high\_resolution\_clock::now();  bubbleSort(descendingArray, sizes[i]);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Bubble Sort for descending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  // Insertion Sort for ascending array  start = high\_resolution\_clock::now();  insertionSort(ascendingArray, sizes[i]);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Insertion Sort for ascending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  // Insertion Sort for descending array  start = high\_resolution\_clock::now();  insertionSort(descendingArray, sizes[i]);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Insertion Sort for descending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  // Selection Sort for ascending array  start = high\_resolution\_clock::now();  selectionSort(ascendingArray, sizes[i]);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Selection Sort for ascending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  // Selection Sort for descending array  start = high\_resolution\_clock::now();  selectionSort(descendingArray, sizes[i]);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Selection Sort for descending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  // Merge Sort for ascending array  start = high\_resolution\_clock::now();  mergeSort(ascendingArray, 0, sizes[i] - 1);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Merge Sort for ascending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  // Merge Sort for descending array  start = high\_resolution\_clock::now();  mergeSort(descendingArray, 0, sizes[i] - 1);  end = high\_resolution\_clock::now();  duration = duration\_cast<microseconds>(end - start);  cout << "Merge Sort for descending array of size " << sizes[i] << ": " << duration.count() << " microseconds\n";  cout << endl;  // Free allocated memory  delete[] ascendingArray;  delete[] descendingArray;  }  return 0;  }  Output:    The rest of the output is not shown because of the limitations of the online compiler.   |  |  |  | | --- | --- | --- | | **Array Size** | **Running time of Bubble Sort for array in Ascending Order(microseconds)** | **Running time of Bubble Sort for array in Descending Order(microseconds)** | | 100 | 16 microseconds | 23 microseconds | | 1000 | 1602 microseconds | 2668 microseconds | | 10000 | 1199483 microseconds | 1402195 microseconds | | 100000 | Not been Shown | Not been Shown | | 1000000. | Not been Shown | Not been Shown |  |  |  |  | | --- | --- | --- | | **Array Size** | **Running time of Insertion Sort for array in Ascending Order(microseconds)** | **Running time of Insertion Sort for array in Descending Order(microseconds)** | | 100 | 0 microseconds | 0 microseconds | | 1000 | 3 microseconds | 3 microseconds | | 10000 | 59 microseconds | 93 microseconds | | 100000 | Not been Shown | Not been Shown | | 1000000. | Not been Shown | Not been Shown |  |  |  |  | | --- | --- | --- | | **Array Size** | **Running time of Selection Sort for array in Ascending Order (microseconds)** | **Running time of Selection Sort for array in Descending Order (microseconds)** | | 100 | 12 microseconds | 12 microseconds | | 1000 | 85483 microseconds | 2033 microseconds | | 10000 | 495799 microseconds | 600800 microseconds | | 100000 | Not been Shown | Not been Shown | | 1000000. | Not been Shown | Not been Shown |  |  |  |  | | --- | --- | --- | | **Array Size** | **Running time of Merge Sort for array in Ascending Order(microseconds)** | **Running time of Merge Sort for array in Descending Order(microseconds)** | | 100 | 6 microseconds | 5 microseconds | | 1000 | 113 microseconds | 98 microseconds | | 10000 | 940 microseconds | 1165 microseconds | | 100000 | Not been Shown | Not been Shown | | 1000000. | Not been Shown | Not been Shown |   Conclusion:  The algorithm that shows the most significant variations in running time based on the input structure is "Bubble Sort." This is because Bubble Sort has a time complexity of O(n2) in the worst case and is highly sensitive to the initial order of elements in the array. When the array is in ascending order, Bubble Sort has the best-case scenario, with a running time close to linear. However, when the array is in descending order, it experiences the worst-case scenario, leading to much higher running times.  In contrast, other sorting algorithms like Insertion Sort, Selection Sort, and Merge Sort have more consistent performance regardless of the initial order of the array. Insertion Sort and Selection Sort also have quadratic time complexities but don't show as much variation because they have different inner workings and optimizations compared to Bubble Sort. Merge Sort, on the other hand, has a consistent time complexity of O(n\*log(n)) for any input, making it less sensitive to the input structure. |

### Deliverables

Compile a single word document by filling in the solution part and submit this Word file on LMS. Insert the solution/answer in this document. You must show the implementation of the tasks in the designing tool, along with your complete Word document to get your work graded.