|  |
| --- |
| **Data Structures and Algorithms**  ***Section*: BSCE2021 Assignment # 3 *Total marks*: 100**  ***Name*** : ***\_\_NIMRA MAQBOOL\_ Roll number* : BSCE21012\_** |

***Submission:***

• *Email instructor or TA if there are any questions. You cannot look at others’ solutions or use others’ solutions, however, you can discuss it with each other. Plagiarism will be dealt with according to the course policy.*

*• Submission after due time will not be accepted.*

**There should be a Report explaining your code and highlighting the results. Follow this naming convention for your report RollNumber\_Assignment#.pdf e.g BSCE21001\_Assignment3.pdf.**

**TASK:**

In the previous lab, you implemented the following function using Arraylist

Selection Sort

Insertion Sort

Merge Sort

Now write an in-depth time complexity analysis of the above function. write this analysis in this word document.

|  |
| --- |
| **FUNCTION.H:**  #include <iostream>  using namespace std;  class mergeArray {  public:  int sizeOfArray; //declaring  int NoOfElement; //declaring  int \*array; public:   mergeArray(int s) {  sizeOfArray = s; //copying  NoOfElement = 0; //placing it to zero  array = new int[sizeOfArray];  for (int i = 0; i < sizeOfArray; i++) {  array[i] = 0; //setting values to zero  }  }   void add() {  cout << "ENTER NUMBERS IN ARRAY TO BE SORTED = ";  for (int i = 0; i < sizeOfArray; i++) {  cin >> array[i]; //taking elements in the array  }  }   void display() {  cout << "THE SORTED ARRAY =";  for (int i = 0; i < sizeOfArray; i++) {  cout << array[i] << " "; //displaying the array  }  cout << endl;  }   void selectionSort(){  int temp;  for(int i=0;i<sizeOfArray-1;i++){ //applying loop till the size-1 starting from 0  int min=i;  for(int j=i+1;j<sizeOfArray;j++){ //applying inner loop starting from i to greater than zero  if(array[j]<array[min]){ //comparing  min=j;  }  }  if(min!=i){  temp=array[i]; //swapping  array[i]=array[min];  array[min]=temp;  }  }  }  void insertionSort() {  int temp;  for (int i = 1;  i < sizeOfArray; i++) { //applying loop till the size starting from 1  for (int j = i; j >0; j--) { //applying inner loop starting from i to greater than zero  if (array[j - 1] > array[j]) { //comparing  temp = array[j - 1]; //swapping  array[j - 1] = array[j];  array[j] = temp;  }  }  }  }   void merge(int \*arr, int lowerBound, int upperBound, int midTerm) {  int i = lowerBound;; //I have copied the lowerbound in i  int j = midTerm + 1; //I have copied the midTErm +1 in j  int z = lowerBound; //I have copied the lowerbound in z  int arr1[100];  while (j <= upperBound && i <=  midTerm) { //then applied an loop to check that if j and i are between the midterm and upperbound  if (arr[i] < arr[j]) { //checking which element is greater  arr1[z] = arr[i]; //putting the value  i++; //iterating i  } else {  arr1[z] = arr[j]; //putting this element  j++;  }  z++;  }  while (i <= midTerm) { //checking the i <= midterm  arr1[z] = arr[i]; //putting  z++;  i++;  }  while (j <= upperBound) { //checking that j is less than upper bound  arr1[z] = arr[j];  z++;  j++;  }  for (i = lowerBound; i < z; i++) {  arr[i] = arr1[i]; //copying  }  }   void mergeSort(int \*arr, int lowerBound, int upperBound) {  int midTerm; //declaring  if (lowerBound < upperBound) { //checking  midTerm = (lowerBound + upperBound) / 2; //calculating the mid of the array  mergeSort(arr, lowerBound, midTerm); //calling the function  mergeSort(arr, midTerm + 1, upperBound); //calling the function itself  merge(arr, lowerBound, upperBound, midTerm); //calling  }  }  };  **MAIN.CPP:**  #include <iostream> #include "Functions.h" using namespace std; // int main() {    // Call all functions one by one here to test the output  int opt;  cout << "PLEASE ENTER YOUR CHOICE." << endl;  cout << "1.SELECTION SORT." << endl;  cout << "2.INSERTION SORT." << endl; //displaying options  cout << "3.MERGE SORT." << endl;  cout << "4.EXIT." << endl;  cin >> opt;  if (opt == 1) {  mergeArray M(5); //making an object  M.add();  M.selectionSort(); //calling  M.display();  }  if (opt == 2) {  mergeArray M(5); //making an object  M.add();  M.insertionSort();  M.display();  }  if (opt == 3) {  mergeArray M(5);  int size; //declaring  int \*array1 = new int[size]; //making an  cout << "ENTER SIZE OF ARRAY = ";  cin >> size; //taking size of array  cout << "ENTER ELEMENTS OF ARRAY = ";  for (int i = 0; i < size; i++) {  cin >> array1[i]; //taking input  }  M.mergeSort(array1, 0, size - 1); //calling function  cout << "SORTED ARRAY AFTER MERGE SORT = ";  for (int i = 0; i < size; i++) {  cout << array1[i] << " "; //displaying  }  cout << endl << endl;  }  if (opt == 4) {  cout << "YOU CHOOSE TO EXIT..." << endl;  exit(2);  }  return 0; }  **SELECTION SORT:**  void selectionSort(){  int temp;  for(int i=0;i<sizeOfArray-1;i++){   int min=i;  for(int j=i+1;j<sizeOfArray;j++){   if(array[j]<array[min]){   min=j;  }  }  if(min!=i){  temp=array[i];   array[i]=array[min];  array[min]=temp; }}}  **COST REPITITION TOTAL**  **1 1(for temp) 1**  **For outer loop:**  **1 1(for int i) 1**  **1 n-1 (for s-1) n-1**  **1 n (increment i++) n**  **For inner loop:**  **1 n (declaration of j) n**  **1 n\*(n+1)/2 (for j<s) n\*(n+1)/2**  **1 n\*n (increment i++) n\*n**  **For swapping:**  **1 n-1(swapping) n-1**  **1 n-1(swapping) n-1**  **Total avg case time complexity is O(n^2)**   * **Which occurs when the array is neither sorted nor reversed.**   **Best-case time complexity is O(n^2).**   * **Which occurs when the array is sorted.**   **Worst-case time complexity is O(n^2).**   * **Which occurs when the array is reversed.**   **The space complexity is O(1) as they are independent of the input size given.**  **Insertion Sort:**  void insertionSort() {  int temp;  for (int i = 1;i < sizeOfArray; i++) {   for (int j = i; j >0; j--) { if (array[j - 1] > array[j]) { temp = array[j - 1];   array[j - 1] = array[j];  array[j] = temp;  }  }  } }  **COST REPITITION TOTAL**  **1 1(for temp ) 1**  **For outer loop:**  **1 1(for int i) 1**  **1 n-1 (for s-1) n-1**  **1 n (increment i++) n**  **For inner loop:**  **1 n (declaration of j) n**  **1 n(n+1)/2 n(n+1)/2**  **1 n\*n (increment i++) n\*n**  **For swapping:**  **1 (n-1) (n-1)**  **Total avg case time complexity is O(n^2)**   * **Which occurs when the array is neither sorted nor reversed.**   **Best-case time complexity is O(n).**   * **Which occurs when the array is sorted.**   **Worst-case time complexity is O(n^2).**   * **Which occurs when the array is reversed.**   **The space complexity is O(1) as they are independent of the input size given.**  **Merge Sort:**  void merge(int \*arr, int lowerBound, int upperBound, int midTerm) {  int i = lowerBound;;  int j = midTerm + 1;   int z = lowerBound;   int arr1[100];  while (j <= upperBound && i <=midTerm) {   if (arr[i] < arr[j]) {   arr1[z] = arr[i];   i++;   } else {  arr1[z] = arr[j];   j++;  }  z++;  }  while (i <= midTerm) {   arr1[z] = arr[i];   z++;  i++;  }  while (j <= upperBound) {   arr1[z] = arr[j];  z++;  j++;  }  for (i = lowerBound; i < z; i++) {  arr[i] = arr1[i];   }}  void mergeSort(int \*arr, int lowerBound, int upperBound) {  int midTerm;   if (lowerBound < upperBound) {   midTerm = (lowerBound + upperBound) / 2;   mergeSort(arr, lowerBound, midTerm);   mergeSort(arr, midTerm + 1, upperBound);   merge(arr, lowerBound, upperBound, midTerm);   }}  **COST REPITITION TOTAL**  **Declaration:**  **1 1 1**  **For first loop:**  **1 n-1 n-1**  **For the second loop:**  **1 n-1 n-1**  **For the third loop:**  **1 n-1 n-1**  **For for loop:**  **1 1 1**  **1 n-1 (for s-1) n-1**  **1 n (increment i++) n**  **For callings in if condition:**  **1 nlog(n) nlog(n)**  **1 nlog(n) nlog(n)**  **1 n n**  **Total avg case time complexity is O(nlog(n))**   * **Which occurs when the array is neither sorted nor reversed.**   **Best-case time complexity is O(nlog(n)).**   * **Which occurs when the array is sorted.**   **Worst-case time complexity is O(nlog(n)).**   * **Which occurs when the array is reversed.**   **The space complexity is O(n) as they are dependent of the input size given.** |