Searching and Sorting Algorithms

3A. Binary Search: Implement a program to search and find the student details in an efficient manner. Reduce the number of comparisons as much as possible. Use the student's registration number as the key. Store the student's name, registration number, phone number and CGPA in the list of student details.

Algorithm:

Step 1: Create class Student to store the details of the students with attributes reg no, first name, last name, phone number, and CGPA.
Step 2: Input the number of students 'n'.
Step 3: Initialize an array of type Student of length n.
Step 4: Input the details of n students and store it in the array initialized above.
Step 5: Sort the given array using merge sort.
Step 6: For implementing merge sort, recursively divide the array into two sub arrays, and when the base case is reached, merge the sorted subarrays, leading up the sorted main array.
Step 7: Input the search query.
Step 8: Binary search the sorted main array for the query.
Step 9: Binary search is implemented by the following method:

```
function binary_search(Student Array[], int high, int low,
                          Student target) {

 if (low > high)  return false;

 int mid = (high + low) / 2;

  if (Array[mid] = target) return true;
  else if (Array[mid] < target)
          return binary_search(Array, high, mid, target);
  else return binary_search(Array, mid, low, target);
}
```

Program: [In next page]

```cpp
// Exp 3A: Binary search student details
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>
#include <string>

class Student {
public:
  //initialize student attributes
  int reg_no;
  std::string first_name;
  std::string last_name;
  std::string phone_number;
  float CGPA;

  // constructor for Student
  Student(int r_no, std::string f_name, std::string l_name,
          std::string ph_no, float cgpa)
  {
    reg_no = r_no;
    first_name = f_name;
    last_name = l_name;
    phone_number = ph_no;
    CGPA = cgpa;
  }
  // empty constructor for Student
  // used when initializing empty array
  Student() {}
};

int binary_search(Student* students, int target, int high, int
low) {
  // base case for recursion
  if (low > high) {
    std::cout << "Not found!\n\n";
    return -1;
  }
  // get index of middle element
  int mid = (high + low) / 2;
```

```
  // if found print the student details

if (students[mid].reg_no == target) {
    std::cout << "Found at " << mid << "\n";

    std::cout << "Registration No: " << students[mid].reg_no <<
        " | Name: " << students[mid].first_name <<
        " " << students[mid].last_name << " | Phone no: " <<
        students[mid].phone_number << " | CGPA: " <<
        students[mid].CGPA << "\n\n";

    return mid;
  } else if (students[mid].reg_no < target) {
    // check the right sub array
    return binary_search(students, target, high, mid+1);
  } else if (students[mid].reg_no > target) {
    // check the let sub array
    return binary_search(students, target, mid-1, low);
  }
}


// use merge sort for efficient sorting of student data
void merge(Student* students, int l, int mid, int r) {
  // compute the lengths of the right and the left subarray
  int len_1 = mid - l + 1;
  int len_2 = r - mid;

  // Create temp arrays
  Student L[len_1], R[len_2];

  // initialize and copy data to the right and left subarray
  // from the main array
  for (int i = 0; i < len_1; i++)
    L[i] = students[l + i];
  for (int j = 0; j < len_2; j++)
    R[j] = students[mid + 1 + j];
```

```
  /* i is the index for left subarray
  * j is the index for right subarray
  * k is the index of the main array
  */
  int i{0}, j{0}, k{l};

  while (i < len_1 && j < len_2) {
    if (L[i].reg_no <= R[j].reg_no) {
      students[k] = L[i];
      i++;
    }
    else {
      students[k] = R[j];
      j++;
    }
    k++;
  }

  while (i < len_1) {
    students[k] = L[i]; i++; k++;
  }
  while (j < len_2) {
    students[k] = R[j]; j++; k++;
  }
}


// main merge sort function
void merge_sort(Student* students, int left, int right) {
  if (left >= right ) return;
  int mid = (right + left) / 2;
  merge_sort(students, left, mid);
  merge_sort(students, mid+1, right);
  merge(students, left, mid, right);
}
```

```
int main() {
  int n; // initialize and input number of students
  std::cout << "Enter number of students: ";
  std::cin >> n;
  // initalize an array with n students
  Student* students = new Student[n];
  // initialize temporary variables for storing inputs
  std::string temp_f_name, temp_l_name, temp_ph_no;
  int temp_r_no;
  float temp_gpa;

  // input details of n students
  for (int i = 0; i < n; i++) {
    std::cout << "Enter first name, last name, phone number, "
<<
                  "registration number and the CGPA:\n";
    std::cin >> temp_f_name >> temp_l_name >> temp_ph_no >>
                temp_r_no >> temp_gpa;
    Student new_student(temp_r_no, temp_f_name,
                     temp_l_name, temp_ph_no,temp_gpa);
    students[i] = new_student;
  }
  // sort the students array
  merge_sort(students, 0, n-1);

  int query; // initialize variable for input query
  std::cout << "Enter a query: (0 to break)\n";

  while (std::cin >> query) {
    if (!query) return 0;
    std::cout << "Query: " << query << " | ";
    // search for the queried student reg no
    binary_search(students, query, n, 0);
  }
}
```

Output:

```
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Executables> .\students.exe
Enter number of students: 5
Enter first name, last name, phone number, registration number and the CGPA:
Pranjal Timsina 9999999999 392 9
Enter first name, last name, phone number, registration number and the CGPA:
Aarav Timsina 8888888888 393 9
Enter first name, last name, phone number, registration number and the CGPA:
John Wick 7777777777 7 7
Enter first name, last name, phone number, registration number and the CGPA:
David Gilmour 5555555555 5 5
Enter first name, last name, phone number, registration number and the CGPA:
Kirk Hammett 2222222222 8 8
Enter a query: (0 to break)
8
Query: 8 | Found at 2
Registration No: 8 | Name: Kirk Hammett | Phone no: 2222222222 | CGPA: 8

392
Query: 392 | Found at 3
Registration No: 392 | Name: Pranjal Timsina | Phone no: 9999999999 | CGPA: 9

393
Query: 393 | Found at 4
Registration No: 393 | Name: Aarav Timsina | Phone no: 8888888888 | CGPA: 9

394
Query: 394 | Not found!

7
Query: 7 | Found at 1
Registration No: 7 | Name: John Wick | Phone no: 7777777777 | CGPA: 7

0
```

Results:

Thus, the program to search and find student details in an efficient manner is implemented.

3B. Application of Binary Search: Implement a program to find the square root of a number. User can give the number randomly. Floor the result in case of floating point.

Algorithm:

Step 1: Input the number from the user.
Step 2: Call the square root function which is implemented in the following way:

```
Function square_root (int low, int high, int number) {
    int ans_sqrt = number;

    while (low <= high) {
        int mid = (high + low) / 2;

        if (mid*mid == number) {
            return mid;
        } else if (mid*mid > number) {
            high = mid - 1;
        } else {
            ans_sqrt = mid;
            low = mid + 1;
        }
    }
    return ans_sqrt;
}
```

Step 3: Print the value returned by the function.

Program: [In next page]

```cpp
// Exp 3B: Square root using binary search
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>

int get_square_root (int low, int high, int number) {
    // variable for storing the answer
    int ans_sqrt = number;

    // condition for exiting the loop
    while (low <= high) {
        int mid = (high + low) / 2;

        if (mid*mid == number) {
            return mid;
        } else if (mid*mid > number) {
            // if the product is greater than then number
            // reduce the set of possible numbers
            high = mid - 1;
        } else {
            // the answer will be the mid in this case
            ans_sqrt = mid;
            low = mid + 1;
        }
    }
    return ans_sqrt;
}
int main() {
    int number; // intialize and input the number
    std::cout << "Enter a number: ";
    std::cin >> number;
    // calculate the square_root and print it
    float square_root = get_square_root(1, number, number);
    std::cout << "The square root is " << square_root;
}
```

Output:

```
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Executables> .\square_root.exe
Enter a number: 55
The square root is 7
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Executables> .\square_root.exe
Enter a number: 25
The square root is 5
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Executables> .\square_root.exe
Enter a number: 1
The square root is 1
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Executables> .\square_root.exe
Enter a number: 0
The square root is 0
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Executables> .\square_root.exe
Enter a number: 36
The square root is 6
```

Results:

Thus, the program to find the square root of a number using binary search is implemented.

2C. Bubble Sort: Sort the given array of elements in ascending order and print out the number of comparisons performed.

Algorithm:

Step 1: Input the number of elements in an array.
Step 2: Initialize an array of appropriate data type.
Step 3: Input the elements of the array
Step 4: Call bubble sort on the array which is implement as follows:

```
bubble_sort(array)
   for i = 0 to len(array)
       for j = 0 to len(array) - i
            if array[j] > array[j+1]
                swap(array[j], array[j+1])
```

Step 5: Print the sorted array, the number of comparisons and the number of array accesses.

Program: [In next page]

```cpp
// Exp 3C: Bubble Sort
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>

// c++ header to calculate time taken
#include <chrono>

// number of comparisons and array accesses
int comparisons{0}, array_access{0};

void bubble_sort(int *numbers, int size) {
    int temp; // variable for swapping data

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size - i; j++) {
            // iterate till size - i as i elements will be
            // sorted in each pass
            array_access +=2;
            // swap if jth element is larget than
            // j+1 th element
            if (++comparisons && numbers[j] > numbers[j+1]) {
                temp = numbers[j];
                numbers[j] = numbers[j+1];
                numbers[j+1] = temp;
                array_access += 4;
            }
        }
    }
}
```

[Continued in next page]

```
int main() {
    // for calculating time taken
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    // initialize and input size of array
    int size, *numbers;
    std::cout << "Enter the number of elements: ";
    std::cin >> size;

    // initialize arary of req size
    numbers = new int[size];
    // input elements of array
    std::cout<<"\nEnter the elements seperated by spaces:\n\n";
    for (int i = 0; i < size; i++) {
        std::cin >> numbers[i];
    }

    // start timing and call bubble sort function
    auto t1 = high_resolution_clock::now();
        bubble_sort(numbers, size);
    auto t2 = high_resolution_clock::now();
    // compute the time duration taken
    duration<double, std::milli> ms_double = t2 - t1;

    std::cout << "\n\nThe sorted array is:\n\n";
    for (int i = 0; i < size; i++) {
        std::cout << numbers[i] << " ";
    }
    // print the time taken, array accesses and comparisons

  std::cout << "\n\nTime taken: " << ms_double.count() << "ms";
  std::cout << " | Array accesses: " << array_access <<
               " | Comparisons: " << comparisons << "\n\n";
}
```

Output:

```
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Code> .\out.exe
Enter the number of elements: 10

Enter the elements seperated by spaces:

43 8 23 9 5 8 4 16 4 7


The sorted array is:

4 4 5 7 8 8 9 16 23 43

Time taken: 0ms | Array accesses: 238 | Comparisons: 55
```

Results:

Thus, the program to sort the given array using bubble sort is implemented.

2D. Insertion Sort: Sort the given array of elements in ascending order and print out the number of comparisons performed.

Algorithm:

Step 1: Input the number of elements in an array.
Step 2: Initialize an array of appropriate data type.
Step 3: Input the elements of the array
Step 4: Call insertion sort on the array which is implement as follows:

```
InsertionSort(array)
    for j = 1 to len(array)-1
        key = array[j]
        i = j-1
        while i > 0 and array[i]>key
            array[i+1] = array[i]
            i = i -1
        array[i+1] = key
```

Step 5: Print the sorted array, the number of comparisons and the number of array accesses.

Program: [In next page]

```cpp
// Exp 3D: Insertion Sort
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>

// c++ header for time
#include <chrono>

// intialize number of comparisons and
// number of array accesses
int comparisons{0}, array_access{0};

void insertion_sort(int *numbers, int size) {
    int key;

    // start by assuming number[0 to i exclusive]
    // is sorted
    for (int i = 1; i < size; i++) {
        // use ith element as key
        key = numbers[i];
        // increment number of array accesses
        array_access++;
        // insert the key in its proper position
        // left of i (inclusive)
        int j = i-1;
        while (j >= 0 && ++comparisons && ++array_access
              && numbers[j] > key) {
            // shift elements to right
            // if condition above is met
            numbers[j+1] = numbers[j];
            array_access +=2;
            j--;
        }
        // insert the key at appropriate location
        numbers[j+1] = key;
        array_access+=1;
    }
}
```

```
int main() {
    // for calculating time taken by sort
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    // initialize and input array
    // and size of array
    int size, *numbers;
    std::cout << "Enter the elements: ";
    std::cin >> size;

    numbers = new int[size];
    std::cout << "Enter " << size <<
                "elements seperated by spaces: \n";
    // input array elements
    for (int i = 0; i < size; i++) {
        std::cin >> numbers[i];
    }

    // start timer, call insertion sort
    // then end timer
    auto t1 = high_resolution_clock::now();
        insertion_sort(numbers, size);
    auto t2 = high_resolution_clock::now();

    // compute time taken
    duration<double, std::milli> ms_double = t2 - t1;
    std::cout << "\nSorted array:\n\n";
    for (int i = 0 ; i < size; i++) {
        std::cout << numbers[i] << " ";
    }

    // display statistics of the sort
    std::cout << "\n\nTime taken: " << ms_double.count()<<"ms";
    std::cout << " | Array accesses: " << array_access <<
                 " | Comparisons: " << comparisons << "\n";
}
```

Output:

```
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Code> .\out.exe
Enter the number elements: 10
Enter 10 elements seperated by spaces:
12 4 65 7 2 9 8 8 23 4

Sorted array:

2 4 4 7 8 8 9 12 23 65

Time taken: 0ms | Array accesses: 94 | Comparisons: 30
```

Results:

Thus, the program to sort the given array using insertion sort is implemented.

2E. Selection Sort: Sort the given array of elements in ascending order and print out the number of comparisons performed.

Algorithm:

Step 1: Input the number of elements in an array.
Step 2: Initialize an array of appropriate data type.
Step 3: Input the elements of the array
Step 4: Call selection sort on the array which is implement as follows:

```
SelectionSort(array)
    for j = 0 to len(array)-1
        key = j
        for i = j+1 to len(array)-1
            if (array[i] < array[key])
                key = i

        swap(array[j], array[key])
```

Step 5: Print the sorted array, the number of comparisons and the number of array accesses.

Program: [In next page]

```cpp
// Exp 3E: Selection Sort
// Author: Pranjal Timsina
#include <iostream>

// c++ header for time
#include <chrono>

// initialize number of comparisons and
// number of array accesses
int comparisons{0}, array_access{0};

void selection_sort(int *numbers, int size) {

    for (int i = 0; i < size; i++) {
        // find the smallest element
        // and put it in its appropriate place
        // then move on to the next smallest element
        int key = i;
        for (int j = i+1; j < size; j++) {
            if (++comparisons&& ++++array_access && numbers[j] < numbers[key]) {
                // find the smallest element and set it as key
                key = j;
            }
        }
        // put the smallest element in its appropriate place
        int temp = numbers[key];
        numbers[key] = numbers[i];
        numbers[i] = temp;
        array_access += 3;
    }
}
```

```cpp
int main() {
    // headers for calculating time taken by sort
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    // initialize the size and the array
    int size, *numbers;
    std::cout << "Enter the number of elements: ";
    std::cin >> size;

    // input the required number of elements
    numbers = new int[size];
    std::cout << "Enter the elements seperated by spaces:\n";
    for (int i = 0; i < size; i++)
        std::cin >> numbers[i];

    // start clock, run selection sort
    // and stop the clock
    auto t1 = high_resolution_clock::now();
        selection_sort(numbers, size);
    auto t2 = high_resolution_clock::now();

    // compute time taken
    duration<double, std::milli> ms_double = t2 - t1;

    std::cout << "\nSorted array:\n\n";
    for (int i = 0; i < size; i++) {
        std::cout << numbers[i] << " ";
    }

    // give output statistics
    std::cout << "Time taken: " << ms_double.count() << "ms";
    std::cout << " | Array accesses: " << array_access <<
                 " | Comparisons: " << comparisons << "\n";
}
```

Output:

```
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Code> .\out.exe
Enter the number of elements: 10
Enter the elements seperated by spaces:
123 123 445 85 69 24 5 2 57 7

Sorted array:

2 5 7 24 57 69 85 123 123 445

Time taken: 0ms | Array accesses: 120 | Comparisons: 45
```

Results:

Thus, the program to sort the given array using selection sort is implemented.

2F. Quick Sort: Sort the given array of elements in ascending order and print out the number of comparisons performed.

Algorithm:

Step 1: Input the number of elements in an array.
Step 2: Initialize an array of appropriate data type.
Step 3: Input the elements of the array
Step 4: Call quick sort on the array which is implement as follows:

```
QuickSort(Array, p ,r)
    if p < r
        q = Partition(Array, p, r)
        QuickSort(Array, p, q-1)
        QuickSort(Array, q+1, r)

Partition(Array, p, r)
    x = Array[r]
    i = p-1
    for j = p to r-1
        if array[j] < = x
            i += 1
            swap(Array[i], Array[j])
    swap(Array[i+1], Array[r])
    return i+1
```

Step 5: Print the sorted array, the number of comparisons and the number of array accesses.

Program: [In next page]

```cpp
// Exp 3F: Quick Sort
// Author: Pranjal Timsina; 20BDS0392

#include <iostream>

// c++ header for time
#include <chrono>

// intialize number of comparisons and
// number of array accesses
int array_access{0}, comparisons{0};

int partition(int * numbers, int p, int r) {
    // selects the right most element as the pivot
    int x{numbers[r]}, i{p-1}, temp;
    array_access++;

    for (int j = p; j <= r-1; j++) {
        // check and puts the elements in the array to the
        // right or left of the pivot accordingly
        if (++array_access && ++comparisons && numbers[j] <= x)
          {
            i++;
            // swap jth and ith element
            temp = numbers[j];
            numbers[j] = numbers[i];
            numbers[i] = temp;
            array_access+=4;
        }
    }

    // put the pivot element back in place
    temp = numbers[i+1];
    numbers[i+1] = numbers[r];
    numbers[r] = temp;
    array_access+=4;
    return i+1;
}
```

```
void quick_sort(int *numbers, int left, int right) {
    if (left < right) {
        // find the index of pivot element
        int q = partition(numbers, left, right);
        // sort the subarrays recursively
        quick_sort(numbers, left, q-1);
        quick_sort(numbers, q+1, right);
    }
}

int main() {
    // for calculating time taken by sort
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    // initialize size and array
    int size, *numbers;
    std::cout << "Enter the size of array: ";
    std::cin >> size;
    // input elements of array
    numbers = new int[size];
    std::cout << "Enter the elements:\n\n";
    for (int i = 0; i < size; i++)
        std::cin >> numbers[i];

    // start timer, call quick sort
    // and end timer
    auto t1 = high_resolution_clock::now();
        quick_sort(numbers, 0, size-1);
    auto t2 = high_resolution_clock::now();

    // compute time taken by sort
    duration<double, std::milli> ms_double = t2 - t1;
```

```
     // print the sorted array
    std::cout << "The sorted array is:\n\n";
    for (int i = 0; i < size; i++) {
        std::cout << numbers[i] << " ";
    }
    // print sort statistics
    std::cout<<"\n\nTime taken: "<< ms_double.count() << "ms";
    std::cout << " | Array accesses: " <<  array_access <<
                 " | Comparisons: " << comparisons << "\n";
}
```

Output:

```
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Code> .\out.exe
Enter the size of array: 10
Enter the elements:

234 4325 1432 45 9 1 4 5 9
12
The sorted array is:

1 4 5 9 9 12 45 234 1432 4325

Time taken: 0ms | Array accesses: 116 | Comparisons: 22
```

Results:

Thus, the program to sort the given array using quick sort is implemented.

2G. Merge Sort: Sort the given array of elements in ascending order and print out the number of comparisons performed.

Algorithm:

Step 1: Input the number of elements in an array.
Step 2: Initialize an array of appropriate data type.
Step 3: Input the elements of the array
Step 4: Call merge sort on the array which is implement as follows:

```
MergeSort(Array, p ,r)
    if p < r
        q = (p+r) / 2
        MergeSort(Array, p, q)
        MergeSort(Array, q+1, r)
        Merge(Array, p, q, r)

Merge(Array, p , q ,r)
    len_1 = q - p + 1;
    len_2 = r - q;
    if Array[mid+1] > Array[mid]) return
    Left = Array[p:len_1-1]
    Right = Array[q+1: q + 1 + len_2]
    i = 0,j = 0,k = l

    while i < len_1 and j < len_2
        if  Left[i] <= Right[j]
            Array[k] = L[i]
            i++
        else
            Array[k] = Right[j]
            j++
        k++
    while i < len_1
        Array[k] = Left[i]
        i++, k++
    while j < len_2
        Array[k] = Right[j]
        j++, k++
```

Step 5: Print the sorted array, the number of comparisons and the number of array accesses.

Program:

```
// Exp 3G: Merge sort
// Author: Pranjal Timsina; 20BDS0392

#include <iostream>
// header for timing functions
#include <chrono>

// number of comparisons and array accesses
int array_access{0}, comparisons{0};

void merge(int * numbers, int l, int mid, int r) {
    // compute the lengths of the right and the left subarray
    int len_1 = mid - l + 1;
    int len_2 = r - mid;
    if (++comparisons && ++++array_access && numbers[mid+1] > n
umbers[mid]) return;
    // initialize and copy data to the right and left subarray
    // from the main array
    int L[len_1], R[len_2];
    for (int i = 0; i < len_1; i++)
        L[i] = numbers[l + i];
    for (int j = 0; j < len_2; j++)
        R[j] = numbers[mid + 1 + j];

    array_access = array_access + len_1 + len_2;
    /* i is the index for left subarray
     * j is the index for right subarray
     * k is the index of the main array
     */
    int i{0}, j{0}, k{l};
```

[Continued in next page]

```
     // copy elements from the right & left
    // sub arrays to the main array
    // in ascending order
    while (i < len_1 && j < len_2) {
        if (++++array_access && ++comparisons && L[i] <= R[j])
{
            numbers[k] = L[i];
            i++;
        } else {
            numbers[k] = R[j];
            j++;
        }
        array_access += 2;
        k++;
    }
    // if any element is left in either of the sub arrays
    // copy them to the main array
    while (i < len_1) {
        numbers[k] = L[i]; i++; k++; array_access+=2;
    }
    while (j < len_2) {
        numbers[k] = R[j]; j++; k++; array_access +=2;
    }
}

void merge_sort(int* numbers, int left, int right) {
    // base case for recursion
    if (left >= right ) return;
    int mid = (right + left) / 2;
    // recursively merge_sort the left and right subarrays
    merge_sort(numbers, left, mid);
    merge_sort(numbers, mid+1, right);
    // merge the sorted subarrays
    merge(numbers, left, mid, right);
}
```

[Continued in next page]

```cpp
int main() {
    // used for timing functions
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    // initialize size and the array
    int size, *numbers;
    std::cout << "Enter the size: ";
    std::cin >> size;

    numbers = new int[size];
    // input the elements of the array
    std::cout << "Enter the elements:\n";
    for (int i = 0; i < size; i++) {
        std::cin >> numbers[i];
    }

    // start timer, call merge_sort()
    // and stop the timer
    auto t1 = high_resolution_clock::now();
        merge_sort(numbers, 0, size);
    auto t2 = high_resolution_clock::now();

    // calculate the time taken
    duration<double, std::milli> ms_double = t2 - t1;

    std::cout << "The sorted array is:\n\n";

    for (int i = 0; i < size; i++) {
        std::cout << numbers[i] << " ";
    }
    // display the statistics of the sort
    std::cout << "\n\nTime taken: " <<
              ms_double.count() << "ms";
    std::cout << " | Array accesses: " << array_access <<
                 " | Comparisons: " << comparisons << "\n";
}
```

Output:

```
PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 3\Code> .\out.exe
Enter the size: 10
Enter the elements:
12 4 5 6 8 9 03 5 7 9
The sorted array is:

3 4 5 5 6 7 8 9 9 12

Time taken: 0ms | Array accesses: 122 | Comparisons: 28
```

Results:

Thus, the program to sort the given array using merge sort is implemented.

2H. Compare the number of comparisons of various sorting algorithms mentioned in above questions 3 to 6. Print a table which shows the input array, and number of comparisons performed by various algorithms. Reuse above sorting programs as functions in this new program.

[Note: Since there are up to 1,00,000 elements in the examples below, I have used a .txt file as an input and since it is not feasible to print out all 1,00,000 elements, I have not printed the sorted array.]

Algorithm:

Step 1: Implement the algorithms as in the above questions.
Step 2: Read inputs from a text file.
Step 3: Run each of the sorting algorithms on the array.
Step 4: Print the statistics of the time and operations done by the sorting algorithms

Program: [In next page]

```
// Exp 3H: Performance of different sorting algorithms
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>
#include <iomanip>
#include <chrono>

// number of comparisons and array accesses
unsigned long long comparisons{0}, array_access{0};

// functions are truncated because they are the same in the
// question above
void bubble_sort(int *numbers, int size) { … }
void insertion_sort(int *numbers, int size) { … }
void merge(int * numbers, int l, int mid, int r) { … }
void merge_sort(int* numbers, int left, int right) { … }
int partition(int * numbers, int p, int r) { … }
void quick_sort(int *numbers, int left, int right) { … }
void selection_sort(int *numbers, int size) { … }

// question above
int main() {
    // used for timing functions
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    // input the number of elements
    int size, *numbers, temp, *unsorted;
    std::cin >> size;
    // initialize 2 arrays
    numbers = new int[size];
    unsorted = new int[size];
    // input data in array
    for (int i = 0; i < size; i++) {
        std::cin >> temp;
        numbers[i] = temp;
        unsorted[i] = temp;
    }
```

```
 // print the number of elements
std::cout << "\nFor " << size << " elements:\n\n";

//-----------------------BUBBLE SORT-----------------------

// time the function bubble sort
auto t1 = high_resolution_clock::now();
    bubble_sort(numbers, size);
auto t2 = high_resolution_clock::now();

// compute the time taken
duration<double, std::milli> ms_double = t2 - t1;

// print the statistics
std::cout << "Bubble Sort    | Time taken | " <<
           std::setw(8) <<  ms_double.count() <<
           "ms | Array accesses: " << std::setw(15)
           << array_access << " | Comparisons: "
           << comparisons << "\n";

// ---------------------SELECTION SORT---------------------

// reset the array to unsorted state
for (int i = 0; i < size; i++)
    numbers[i] = unsorted[i];
// reset comparisons and array accesses
comparisons = 0; array_access = 0;

// time the function selection sort
t1 = high_resolution_clock::now();
    selection_sort(numbers, size);
t2 = high_resolution_clock::now();

// compute time taken
 ms_double = t2 - t1;

// print the statistics
std::cout << "Selection Sort | Time taken | "
      << std::setw(8) << ms_double.count() << "ms";
```

```
 std::cout << " | Array accesses: " << std::setw(15)
           << array_access << " | Comparisons: "
           << comparisons << "\n";

// ---------------------INSERTION SORT---------------------

// reset array to unsorted state
for (int i = 0; i < size; i++)
    numbers[i] = unsorted[i];

// reset comparisons and array accesses
comparisons = 0; array_access = 0;

// time the function insertion sort
t1 = high_resolution_clock::now();
    insertion_sort(numbers, size);
t2 = high_resolution_clock::now();

// compute time taken
ms_double = t2 - t1;

// print the statistics
std::cout << "Insertion Sort | Time taken | "
          << std::setw(8) << ms_double.count() << "ms";
std::cout << " | Array accesses: " << std::setw(15)
           << array_access << " | Comparisons: "
           << comparisons << "\n";

// ---------------------MERGE SORT------------------------

// reset array to unsorted state
for (int i = 0; i < size; i++)
    numbers[i] = unsorted[i];

// reset comparisons and array accesses
comparisons = 0; array_access = 0;
```

```
    // time the function merge sort
    t1 = high_resolution_clock::now();
        merge_sort(numbers, 0 , size);
    t2 = high_resolution_clock::now();

    // compute time taken
    ms_double = t2 - t1;

    // display the statistics
    std::cout << "Merge Sort      | Time taken | "
               << std::setw(8) << ms_double.count() << "ms";
    std::cout << " | Array accesses: " << std::setw(15)
               << array_access << " | Comparisons: "
               << comparisons << "\n";

    // --------------------QUICK SORT--------------------------

    // reset array to unsorted state
    for (int i = 0; i < size; i++)
        numbers[i] = unsorted[i];

    // reset comparisons and array accesses
    comparisons = 0; array_access = 0;

    // time the function quick sort
     t1 = high_resolution_clock::now();
        quick_sort(numbers, 0, size-1);
    t2 = high_resolution_clock::now();

    // calculate the time taken
    ms_double = t2 - t1;
    // display the statistics of the sort
    std::cout << "Quick Sort      | Time taken | "
               << std::setw(8) << ms_double.count() << "ms";
    std::cout << " | Array accesses: " << std::setw(15)
                << array_access  << " | Comparisons: "
                << comparisons << "\n\n";

}
```

Output:

Bubble Sort	Inputs 10	Time taken 0ms	Array accesses: 178	Comparisons: 55
Bubble Sort	Inputs 20	Time taken 0ms	Array accesses: 788	Comparisons: 210
Bubble Sort	Inputs 50	Time taken 0ms	Array accesses: 5106	Comparisons: 1275
Bubble Sort	Inputs 100	Time taken 0ms	Array accesses: 20420	Comparisons: 5050
Bubble Sort	Inputs 1000	Time taken 4.0011ms	Array accesses: 2019012	Comparisons: 500500
Bubble Sort	Inputs 5000	Time taken 103.024ms	Array accesses: 50006620	Comparisons: 12502500
Bubble Sort	Inputs 10000	Time taken 456.103ms	Array accesses: 200652392	Comparisons: 50005000
Bubble Sort	Inputs 20000	Time taken 2124.48ms	Array accesses: 801210872	Comparisons: 200010000
Bubble Sort	Inputs 50000	Time taken 15178.4ms	Array accesses: 4997920352	Comparisons: 1250025000
Bubble Sort	Inputs 100000	Time taken 62722.2ms	Array accesses: 19966850900	Comparisons: 5000050000
Insertion Sort	Inputs 10	Time taken 0ms	Array accesses: 77	Comparisons: 25
Insertion Sort	Inputs 20	Time taken 0ms	Array accesses: 331	Comparisons: 109
Insertion Sort	Inputs 50	Time taken 0ms	Array accesses: 2062	Comparisons: 686
Insertion Sort	Inputs 100	Time taken 0ms	Array accesses: 8034	Comparisons: 2676
Insertion Sort	Inputs 1000	Time taken 2.0003ms	Array accesses: 763501	Comparisons: 254497
Insertion Sort	Inputs 10000	Time taken 249.08ms	Array accesses: 75481783	Comparisons: 25160589
Insertion Sort	Inputs 20000	Time taken 986.223ms	Array accesses: 300893143	Comparisons: 100297709
Insertion Sort	Inputs 50000	Time taken 6173.4ms	Array accesses: 1873402753	Comparisons: 624467579
Insertion Sort	Inputs 100000	Time taken 24682.6ms	Array accesses: 7475063165	Comparisons: 2491687715
Merge Sort	Inputs 10	Time taken 0ms	Array accesses: 171	Comparisons: 36
Merge Sort	Inputs 20	Time taken 0ms	Array accesses: 384	Comparisons: 78
Merge Sort	Inputs 50	Time taken 0ms	Array accesses: 1264	Comparisons: 254
Merge Sort	Inputs 100	Time taken 0ms	Array accesses: 2950	Comparisons: 596
Merge Sort	Inputs 1000	Time taken 0ms	Array accesses: 46403	Comparisons: 9328
Merge Sort	Inputs 5000	Time taken 2.0007ms	Array accesses: 291021	Comparisons: 58188
Merge Sort	Inputs 10000	Time taken 2.9835ms	Array accesses: 632299	Comparisons: 126446
Merge Sort	Inputs 20000	Time taken 7.0032ms	Array accesses: 1365112	Comparisons: 273011
Merge Sort	Inputs 50000	Time taken 18.0041ms	Array accesses: 3740904	Comparisons: 747618
Merge Sort	Inputs 100000	Time taken 38.008ms	Array accesses: 7983460	Comparisons: 1595510
Quick Sort	Inputs 10	Time taken 0ms	Array accesses: 103	Comparisons: 24
Quick Sort	Inputs 20	Time taken 0ms	Array accesses: 347	Comparisons: 82
Quick Sort	Inputs 50	Time taken 0ms	Array accesses: 810	Comparisons: 240
Quick Sort	Inputs 100	Time taken 0ms	Array accesses: 2702	Comparisons: 716
Quick Sort	Inputs 1000	Time taken 1.0001ms	Array accesses: 32595	Comparisons: 10490
Quick Sort	Inputs 5000	Time taken 1.0002ms	Array accesses: 221167	Comparisons: 69633
Quick Sort	Inputs 10000	Time taken 3.0158ms	Array accesses: 528405	Comparisons: 158978
Quick Sort	Inputs 20000	Time taken 3.9828ms	Array accesses: 1091773	Comparisons: 325268
Quick Sort	Inputs 50000	Time taken 14.0025ms	Array accesses: 3327384	Comparisons: 984188
Quick Sort	Inputs 100000	Time taken 28.0064ms	Array accesses: 6686776	Comparisons: 2044507
Selection Sort	Inputs 10	Time taken 0ms	Array accesses: 120	Comparisons: 45
Selection Sort	Inputs 20	Time taken 0ms	Array accesses: 440	Comparisons: 190
Selection Sort	Inputs 50	Time taken 0ms	Array accesses: 2600	Comparisons: 1225
Selection Sort	Inputs 100	Time taken 0ms	Array accesses: 10200	Comparisons: 4950
Selection Sort	Inputs 1000	Time taken 5.0019ms	Array accesses: 1002000	Comparisons: 499500
Selection Sort	Inputs 5000	Time taken 122.028ms	Array accesses: 25010000	Comparisons: 12497500
Selection Sort	Inputs 10000	Time taken 489.089ms	Array accesses: 100020000	Comparisons: 49995000
Selection Sort	Inputs 20000	Time taken 1947.44ms	Array accesses: 400040000	Comparisons: 199990000
Selection Sort	Inputs 50000	Time taken 12178.7ms	Array accesses: 2500100000	Comparisons: 1249975000
Selection Sort	Inputs 100000	Time taken 48864.7ms	Array accesses: 10000200000	Comparisons: 4999950000

Results:

From the results above, we can verify the order of growth of different sorting algorithms. It is evident that for a very small number of inputs (< 500) the sorting algorithms do not make much difference; however, as the number of inputs grows, merge sort and quick sort prove to be much faster. Also, despite having a worst-case time complexity of n^2, Quick sort, always performs the best.