A Laboratory Manual for

Analysis and Design of

Algorithms

(3150703)

B.E. Semester 5

(Information Technology)

Directorate of Technical Education, Gandhinagar, Gujarat

Government Engineering College, Modasa

Certificate

This is to certify that Mr./Ms. _____ _____ Enrollment No. _____ of B.E. Semester _____ Information Technology of this Institute (GTU Code: _____) has satisfactorily completed the Practical / Tutorial work for the subject **Analysis and Design of Algorithms(3150703)** for the academic year 2022-23.

Place: _____

Date: _____

Name Faculty member Head of the Department

Preface

Main motto of any laboratory/practical/field work is for enhancing required skills as well as creating ability amongst students to solve real time problem by developing relevant competencies in psychomotor domain. By keeping in view, GTU has designed competency focused outcome-based curriculum for engineering degree programs where sufficient weightage is given to practical work. It shows importance of enhancement of skills amongst the students and it pays attention to utilize every second of time

allotted for practical amongst students, instructors and faculty members to achieve relevant outcomes by performing the experiments rather than having merely study type experiments. It is must for effective implementation of competency focused outcome-based curriculum that every practical is keenly designed to serve as a tool to develop and enhance relevant competency required by the various industry among every student. These psychomotor skills are very difficult to develop through traditional chalk and board content delivery method in the classroom. Accordingly, this lab manual is designed to focus on the industry defined relevant outcomes, rather than old practice of conducting practical to prove concept and theory.

By using this lab manual students can go through the relevant theory and procedure in advance before the actual performance which creates an interest and students can have basic idea prior to performance. This in turn enhances pre-determined outcomes amongst students. Each experiment in this manual begins with competency, industry relevant skills, course outcomes as well as practical outcomes (objectives). The students will also achieve safety and necessary precautions to be taken while performing practical.

This manual also provides guidelines to faculty members to facilitate student centric lab activities through each experiment by arranging and managing necessary resources in order that the students follow the procedures with required safety and necessary precautions to achieve the outcomes. It also gives an idea that how students will be assessed by providing rubrics.

Algorithms are an integral part of computer science and play a vital role in solving complex problems efficiently. The goal of this subject is to equip students with the knowledge and skills required to design and analyze algorithms for various applications. Designing of algorithms is important before implementation of any program or solving any problem. Analysis and Design of Algorithms is essential for efficient problem-solving, optimizing resource utilization, developing new technologies, and gaining a competitive advantage. This lab manual is designed to help you learn algorithms by doing. Each experiment is structured to provide you with step-by-step instructions on how to analyze and design a particular algorithm for specific problem. You will learn how to analyze various algorithms and decide efficient algorithm in terms of time complexity. By the end of this lab, you will have a solid understanding of algorithm design and analysis.

Utmost care has been taken while preparing this lab manual however always there is chances of improvement. Therefore, we welcome constructive suggestions for improvement and removal of errors if any.

Practical – Course Outcome matrix

Course Outcomes (Cos):

1. Analyze the asymptotic performance of algorithms.

2. Derive and solve recurrences describing the performance of divide-and-conquer algorithms.

3. Find optimal solution by applying various methods.

4. Apply pattern matching algorithms to find particular pattern.

5. Differentiate polynomial and nonpolynomial problems.

6. Explain the major graph algorithms and their analyses. Employ graphs to model engineering problems, when appropriate.

Sr. No.	Objective(s) of Experiment	CO 1	CO2	CO3	CO4	CO5	CO6
1.	Implement a function for each of following problems and count the number of steps executed/time taken by each function on various inputs (100 to 500) and write time complexity of each function. Also draw a comparative chart of number of input versus steps executed/time taken. In each of the following function N will be passed by user. 1. To calculate sum of 1 to N numbers using loop. 2. To calculate sum of 1 to N numbers using equation. 3. To calculate sum of 1 to N numbers using recursion.	√					
	Write user defined functions for the following sorting methods and compare their performance by steps executed/time taken for execution on various inputs (1000 to 5000) of random nature, ascending order and descending order sorted data. Also draw a						

	comparative chart of number of input versus steps executed/time taken for each cases (random, ascending, and descending).					
2.	• ○ 1. Selection Sort 2. Bubble Sort 3. Insertion Sort 4. Merge Sort 5. Quick Sort	√	√			
3.	Implement a function of sequential search and count the steps executed by function on various inputs (1000 to 5000) for best case, average case and worst case. Also, write time complexity in each case and draw a comparative chart of number of input versus steps executed by sequential search for each case.	√				
4.	Compare the performance of linear search and binary search for Best case, Average case and Worst case inputs.	√	√			
5.	Implement functions to print n^{th} Fibonacci number using iteration and recursive method. Compare the performance of two methods by counting number of steps executed on various inputs. Also, draw a comparative chart. (Fibonacci series 1, 1, 2, 3, 5, 8….. Here 8 is the 6^{th} Fibonacci number).	√				√
6.	Implement a program for randomized version of quick sort and compare its performance with the normal version of quick sort using steps count on various inputs (1000 to 5000) of random nature, ascending order and descending order sorted data. Also draw a comparative chart of number of input versus steps executed/time taken for each cases (random, ascending, and descending).	√	√			
7.	Implement program to solve problem of making a change using dynamic programming.			√		
8.	Implement program of chain matrix multiplication using dynamic programming.			√		
9.	Implement program to solve LCS problem using dynamic programing.			√		
10.	Implement program to solve Knapsack problem using dynamic programming.			√		
11.	Implement program for solution of fractional Knapsack problem using greedy design technique.			√		
12.	Implement program for solution of Making Change problem using greedy design technique			√		
13.	Implement program for Kruskal's algorithm to find minimum spanning tree.			√		√
14.	Implement program for Prim's algorithm to find minimum spanning tree.			√		√
15.	Implement DFS and BFS graph traversal techniques and write its time complexities.					√
16.	Implement Rabin-Karp string matching algorithm.				√	

Industry Relevant Skills

The following industry relevant competencies are expected to be developed in the student by undertaking the practical work of this laboratory.

1. Expertise in algorithm analysis
2. Judge best algorithm among various algorithms
3. Ability to solve complex problems
4. Ability to design efficient algorithm for some problems

Instructions for Students

1. Students are expected to carefully listen to all the theory classes delivered by the faculty members and understand the COs, content of the course, teaching and examination scheme, skill set to be developed etc.
2. Students shall organize the work in the group and make record of all observations.

3. Students shall develop maintenance skill as expected by industries.
4. Student shall attempt to develop related hand-on skills and build confidence.
5. Student shall develop the habits of evolving more ideas, innovations, skills etc. apart from those included in scope of manual.
6. Student shall refer technical magazines and data books.
7. Student should develop a habit of submitting the experimentation work as per the schedule and s/he should be well prepared for the same.

Common Safety Instructions

1. Switch on the PC carefully (not to use wet hands)
2. Shutdown the PC properly at the end of your Lab
3. Carefully handle the peripherals (Mouse, Keyboard, Network cable etc).
4. Use Laptop in lab after getting permission from Teacher

Index

(Progressive Assessment Sheet)

Sr. No.	Objective(s) of Experiment	Page No.	Date of performance	Date of submission	Marks	Sign. of Teacher with date
1.	Implement a function for each of following problems and count the number of steps executed/time taken by each function on various inputs (100 to 500) and write time complexity of each function. Also draw a comparative chart of number of input versus steps executed/time taken. In each of the following function N will be passed by user. 1. To calculate sum of 1 to N numbers using loop. 2. To calculate sum of 1 to N numbers using equation. 3. To calculate sum of 1 to N numbers using recursion.					
2.	Write user defined functions for the following sorting methods and compare their performance by steps executed/time taken for execution on various inputs (1000 to 5000) of random nature, ascending order and descending order sorted data. Also draw a comparative chart of number of input versus steps executed/time taken for each cases (random, ascending, and descending). 1. Selection Sort 2. Bubble Sort 3. Insertion Sort 4. Merge Sort 5. Quick Sort					
3.						
4.						
5.	Implement functions to print n^{th} Fibonacci number using iteration and recursive method. Compare the performance of two methods by counting number of steps executed on various inputs. Also, draw a comparative chart. (Fibonacci series 1, 1, 2, 3, 5, 8….. Here 8 is the 6^{th} Fibonacci number). Implement a program for randomized version of quick sort and compare its performance with the normal version of quick sort using steps count					

6. on various inputs (1000 to 5000) of random nature, ascending order and descending order sorted data. Also, draw a comparative chart of number of input versus steps executed/time taken for each cases (random, ascending, and descending).
7. Implement program to solve problem of making a change using dynamic programming.
8. Implement program of chain matrix multiplication using dynamic programming.
9. Implement program to solve LCS problem using dynamic programing.
10. Implement program to solve Knapsack problem using dynamic programming.
11. Implement program for solution of fractional Knapsack problem using greedy design technique.
12. Implement program for solution of Making Change problem using greedy design technique.
13. Implement program for Kruskal's algorithm to find minimum spanning tree.
14. Implement program for Prim's algorithm to find minimum spanning tree.
15. Implement DFS and BFS graph traversal techniques and write its time complexities.
16. Implement Rabin-Karp string matching algorithm.

Total

EXPERIMENT NO: 1

Aim:

Implement a function for each of following problems and count the number of steps executed/time taken by each function on various inputs (100 to 500) and write equation for the growth rate of each function. Also draw a comparative chart of number of input versus steps executed/time taken. In each of the following function N will be passed by user.

1. To calculate sum of 1 to N numbers using loop.
2. To calculate sum of 1 to N numbers using equation.
3. To calculate sum of 1 to N numbers using recursion.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1

Objectives: (a) Compare performance of various algorithms

(b) Judge best algorithm in terms of growth rate or steps executed

Equipment/Instruments: Computer System, Java language editor

Theory/Algorithm:

1. **Below are the steps to calculate sum of 1 to N numbers using loop**

Step 1 - Initialize a variable **sum** to 0.

Step 2 - Create a loop that iterates from 1 to N (inclusive).

Step 3 - Inside the loop, add the current value of the loop variable to the **sum**.

Step 4 - After the loop, the variable **sum** will contain the sum of integers from 1 to N.

1. **Below are the steps to calculate sum of 1 to N numbers using equation**

Step 1 - Use the formula for the sum of an arithmetic series: **sum = (N * (N + 1)) / 2**.

Step 2 - Plug in the value of N into this formula.

Step 3 - Calculate the result using the formula, and it will give you the sum of integers from 1 to N.

1. **Below are the steps to calculate sum of 1 to N numbers using recursion.**

Step 1 - Create a recursive function that takes an integer N as its parameter.

Step 2 - In the base case, when N is 1, return 1, as the sum of integers from 1 to 1 is 1.

Step 3 - In the recursive case, call the function with **N - 1** and add **N** to the result of the recursive call.

Step 4 - The function will keep calling itself with decreasing values of N until it reaches the base case.

When the base case is reached, the recursive calls will start returning values, and these values will be added together to compute the sum of integers from 1 to N.

Program:

```
import java.util.Scanner;

public class Sumcalculator {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a positive integer N: ");

int N = scanner.nextInt();

System.out.println("Choose a method to calculate the sum:");

System.out.println("1. Using loop");

System.out.println("2. Using equation");

System.out.println("3. Using recursion");

System.out.print("Enter your choice (1/2/3): ");

int choice = scanner.nextInt();

long startTime, endTime, executionTime;

switch (choice) {

case 1:

startTime = System.nanoTime();

int sumLoop = calculateSumUsingLoop(N);

endTime = System.nanoTime();

executionTime = endTime - startTime;
```

```
System.out.println("Sum using loop: " + sumLoop);
System.out.println("Execution time: " + executionTime + " nanoseconds");
break;
case 2:
startTime = System.nanoTime();
int sumEquation = calculateSumUsingEquation(N);
endTime = System.nanoTime();
executionTime = endTime - startTime;
System.out.println("Sum using equation: " + sumEquation);
System.out.println("Execution time: " + executionTime + " nanoseconds");
break;
case 3:
startTime = System.nanoTime();
int sumRecursion = calculateSumUsingRecursion(N);
endTime = System.nanoTime();
executionTime = endTime - startTime;
System.out.println("Sum using recursion: " + sumRecursion);
System.out.println("Execution time: " + executionTime + " nanoseconds");
break;
default:
System.out.println("Invalid choice.");
}
scanner.close();
}
// Function to calculate sum using a loop
static int calculateSumUsingLoop(int N) {
int sum = 0;
for (int i = 1; i <= N; i++) {
sum += i;
}
return sum;
```

```
}

// Function to calculate sum using an equation

static int calculateSumUsingEquation(int N) {

return (N * (N + 1)) / 2;

}

// Function to calculate sum using recursion

static int calculateSumUsingRecursion(int N) {

if (N == 1) {

return 1;

}

return N + calculateSumUsingRecursion(N - 1);

}

}
```

Observations:

-
  -
    -

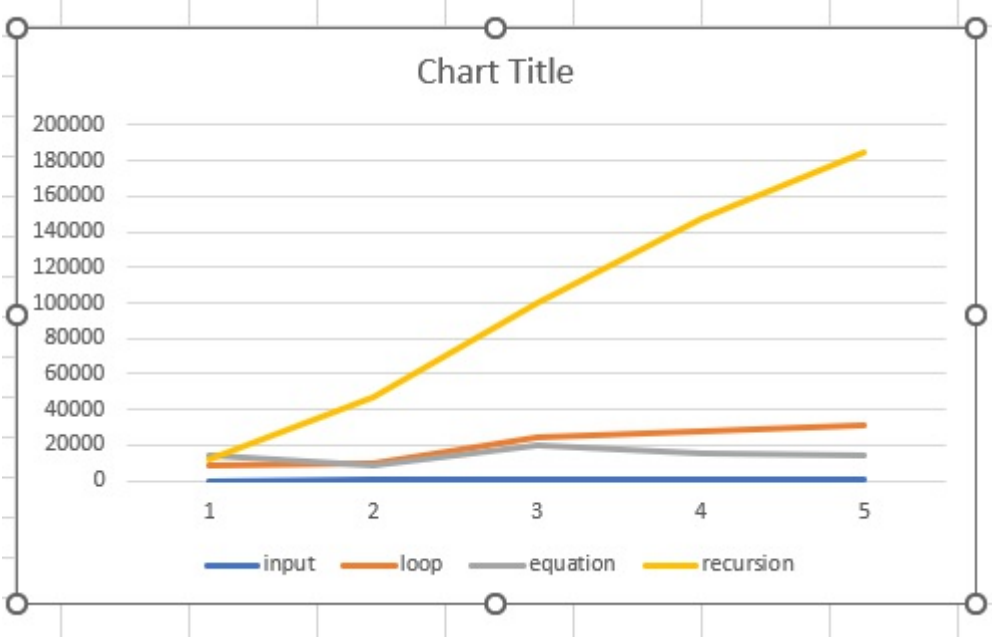
1. **Using loop : In This method we use a simple loop for iterate from 1 to N and find the sum of the Numbers. The Time complexity for Loop Method is O(N).**
2. **Using Equation: In This Method we use a equation for finding the sum so the Time complexity for this method is O(1).**
3. **Using Recursion: In This method We use a recursive call instead of the loop so that the function itself call again and again until the termination condition occurs.**

Result:

Inputs	**Number of Steps Executed**		
	Loop method	**Equations**	**Recursion**
105	8500	14300	12300
205	9800	8100	47400
305	24800	19900	99500
405	28200	15300	147500
495	31000	14600	184300

Equation→

Chart:



Conclusion: From the above graph we can conclude that ,if the number of input is more than it is better to use equation function because it takes less time as compare to recursion and loop. Loop take more time than equation.

Quiz:

1. What is the meaning of constant growth rate of an algorithm?

Answer: A constant resource need is one where the resource need does not grow. Growth rate of the algorithm is constant at any resource.

1. If one algorithm has a growth rate of n^2 and second algorithm has a growth rate of n then which algorithm execute faster? Why?

Answer: Any function that contains an n2 term will grow faster than a function whose leading term is n. N^2 > N then we say that N^2 Is exponentially bigger than N.

Suggested Reference:

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

“Fundamentals of Algorithms” by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Understanding of problem	Program Implementation	Documentation &Timely Submission	T (1

Rubrics	(3)			(5)			(2)		
	Good	Avg.	Poor	Good	Avg.	Poor	Good	Avg.	Poor
	(3)	(2-1)	(1-0)	(5-4)	(3-2)	(1-0)	(2)	(1)	(0)

Marks

EXPERIMENT NO: 2

Aim:

Write user defined functions for the following sorting methods and compare their performance by steps executed/time taken for execution on various inputs (1000 to 5000) of random nature, ascending order and descending order sorted data. Also, draw a comparative chart of number of inputs versus steps executed/time taken for each Cases (random, ascending, and descending).

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1, CO2

Objectives: (a) Compare performance of various algorithms.

(b) Judge best sorting algorithm on sorted and random inputs in terms of growth rate/time complexity.

(c) Derive time complexity from steps count on various inputs.

Equipment/Instruments: Computer System, Any C language editor

Theory:

1. **Algorithm for Selection Sort**

Step 1 :- for i =1 to length(A) - 1

Step 2 :- minIndex = i

Step 3 :- for j = i + 1 to length(A)

Step 4 :- if A[i] < A[minIndex]

Step 5 :- minIndex = j

Step 6 :- swap A[i] with A[minIndex]

1. **Algorithm for Bubble Sort**

Step 1 :- BUBBLESORT(A)

Step 2 :- for i = 1 to A.length - 1

Step 3 :- for j = A.length downto i+1

Step 4 :- if A[j] < A[j-1]

Step 5 :- Exchange A[j] with A[j-1]

1. **Algorithm for Insertion Sort Function in C**

Step 1 :- for j=2 to A.length

Step 2 :- key = A[j]

Step 3 :- i = j-1

Step 4 :- while i>0 and A[i] > key

Step 5 :- A[i+1] = A[i]

Step 6 :- i = i-1

Step 7 :- A[i+1] = key

1. **Algorithm for Merge Sort Function in C**

Step 1 :- MERGE_SORT(arr, beg, end)

Step 2 :- if beg < end

Step 3 :- set mid = (beg + end)/2

Step 4 :- MERGE_SORT(arr, beg, mid)

Step 5 :- MERGE_SORT(arr, mid + 1, end)

Step 6 :- MERGE (arr, beg, mid, end)

Step 7 :- end of if

Step 8 : -END MERGE_SORT

5. Algorithm for Quick Sort Function in C

Step 1 :- QUICKSORT (array A, start, end)

Step 2 : - if (start < end)

Step 3 :- p = partition(A, start, end)

Step 4 :- QUICKSORT (A, start, p - 1)

Step 5 :- QUICKSORT (A, p + 1, end)

Program:

```
import java.util.Arrays;
import java.util.Random;
```

```
import java.util.Scanner;

public class SortingAlgorithms {
public static void selectionSort(int[] arr) {
int n = arr.length;
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;
}
}
int temp = arr[i];
arr[i] = arr[minIndex];
arr[minIndex] = temp;
}
}

public static void insertionSort(int[] arr) {
int n = arr.length;
for (int i = 1; i < n; 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;
}
}

public static void bubbleSort(int[] arr) {
int n = arr.length;
boolean swapped;
for (int i = 0; i < n - 1; i++) {
swapped = false;
for (int j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
swapped = true;
}
}
if (!swapped) {
break;
}
}
}

public static void mergeSort(int[] arr) {
if (arr.length <= 1) {
return;
}
int mid = arr.length / 2;
int[] left = Arrays.copyOfRange(arr, 0, mid);
```

```
int[] right = Arrays.copyOfRange(arr, mid, arr.length);

mergeSort(left);
mergeSort(right);

int i = 0, j = 0, k = 0;
while (i < left.length && j < right.length) {
if (left[i] < right[j]) {
arr[k++] = left[i++];
} else {
arr[k++] = right[j++];
}
}
while (i < left.length) {
arr[k++] = left[i++];
}
while (j < right.length) {
arr[k++] = right[j++];
}
}

public static void quickSort(int[] arr, int low, int high) {
if (low < high) {
int pivotIndex = partition(arr, low, high);
quickSort(arr, low, pivotIndex - 1);
quickSort(arr, pivotIndex + 1, high);
}
}

public static int partition(int[] arr, int low, int high) {
int pivot = arr[high];
int i = low - 1;
for (int j = low; j < high; j++) {
if (arr[j] < pivot) {
i++;
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
}
int temp = arr[i + 1];
arr[i + 1] = arr[high];
arr[high] = temp;
return i + 1;
}

public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("enter number of elements in array : ");
int n = scanner.nextInt();
int[] arr = new int[n];
Random rand = new Random();
for (int i = 0; i < n; i++) {
arr[i] = rand.nextInt(100);
}
```

```
System.out.println("Select sorting algorithm (1: Selection, 2: Insertion, 3: Bubble, 4: Merge, 5: Quick): ");

System.out.print("Enter a choice: ");
int num = scanner.nextInt();


long startTime = System.nanoTime();

switch (num) {
case 1 -> selectionSort(arr);
case 2 -> insertionSort(arr);
case 3 -> bubbleSort(arr);
case 4 -> mergeSort(arr);
case 5 -> quickSort(arr, 0, arr.length - 1);
default -> {
System.out.println("Invalid choice.");
return;
}
}

long endTime = System.nanoTime();
long elapsedTime = endTime - startTime;

System.out.println("Time elapsed: " + (elapsedTime) + " nanoseconds");
}
}
```

Descending Order Program :-

```
import java.util.Random;
import java.util.Scanner;

public class SortingAlgorithmDescending {
public static void selectionSort(int[] arr) {
int n = arr.length;
for (int i = 0; i < n - 1; i++) {
int maxIndex = i;
for (int j = i + 1; j < n; j++) {
if (arr[j] > arr[maxIndex]) {
maxIndex = j;
}
}
// Swap arr[i] and arr[maxIndex]
int temp = arr[i];
arr[i] = arr[maxIndex];
arr[maxIndex] = temp;
}
}
public static void insertionSort(int[] arr) {
int n = arr.length;
for (int i = 1; i < n; i++) {
int key = arr[i];
int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key
// to one position ahead of their current position
```

```
while (j >= 0 && arr[j] < key) {
arr[j + 1] = arr[j];
j = j - 1;
}
arr[j + 1] = key;
}
}
public static void bubbleSort(int[] arr) {
int n = arr.length;
boolean swapped;
for (int i = 0; i < n - 1; i++) {
swapped = false;
for (int j = 0; j < n - 1 - i; j++) {
if (arr[j] < arr[j + 1]) {
// Swap arr[j] and arr[j + 1]
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
swapped = true;
}
}
// If no two elements were swapped in the inner loop, the array is already sorted.
if (!swapped) {
break;
}
}
}
public static void mergeSort(int[] arr, int left, int right) {
if (left < right) {
int mid = (left + right) / 2;

// Recursively sort the left and right halves
mergeSort(arr, left, mid);
mergeSort(arr, mid + 1, right);

// Merge the sorted halves
mergeDescending(arr, left, mid, right);
}
}

public static void mergeDescending(int[] arr, int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;

int[] leftArray = new int[n1];
int[] rightArray = new int[n2];

// Copy data to temp arrays leftArray[] and rightArray[]
for (int i = 0; i < n1; i++) {
leftArray[i] = arr[left + i];
}
for (int j = 0; j < n2; j++) {
rightArray[j] = arr[mid + 1 + j];
}

int i = 0, j = 0, k = left;
```

```
// Merge the two subarrays in descending order
while (i < n1 && j < n2) {
if (leftArray[i] >= rightArray[j]) {
arr[k] = leftArray[i];
i++;
} else {
arr[k] = rightArray[j];
j++;
}
k++;
}


while (i < n1) {
arr[k] = leftArray[i];
i++;
k++;
}


while (j < n2) {
arr[k] = rightArray[j];
j++;
k++;
}
}
public static void quickSort(int[] arr, int low, int high) {
if (low < high) {
int pivotIndex = partition(arr, low, high);
quickSort(arr, low, pivotIndex - 1);
quickSort(arr, pivotIndex + 1, high);
}
}

public static int partition(int[] arr, int low, int high) {
int pivot = arr[high];
int i = low - 1;

for (int j = low; j < high; j++) {
if (arr[j] >= pivot) {
i++;

int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
}



int temp = arr[i + 1];
arr[i + 1] = arr[high];
arr[high] = temp;

return i + 1;
}
```

```
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("enter number of elements in array : ");
int n = scanner.nextInt();
int[] arr = new int[n];
Random rand = new Random();
for (int i = 0; i < n; i++) {
arr[i] = rand.nextInt(100);
}

System.out.println("Select sorting algorithm (1: Selection, 2: Insertion, 3: Bubble, 4: Merge, 5: Quick): ");

System.out.print("Enter a choice: ");
int num = scanner.nextInt();


long startTime = System.nanoTime();

switch (num) {
case 1 -> selectionSort(arr);
case 2 -> insertionSort(arr);
case 3 -> bubbleSort(arr);
case 4 -> mergeSort(arr , 0 ,arr.length-1);
case 5 -> quickSort(arr, 0, arr.length - 1);
default -> {
System.out.println("Invalid choice.");
return;
}
}

long endTime = System.nanoTime();
long elapsedTime = endTime - startTime;

System.out.println("Time elapsed: " + (elapsedTime) + " nanoseconds");
}

}
```

Observations:

Selection sort :- The algorithm repeatedly finds the minimum element in the unsorted part and swaps it with the leftmost element of the unsorted part, thus expanding the sorted part by one element. Selection sort has a time complexity of O (n^2) in all cases, where n is the number of elements in the list

Bubble sort :- It is a simple sorting algorithm that works by repeatedly swapping adjacent elements in an array if they are in the wrong order. However, bubble sort still has a time complexity of O (n^2) in the average and worst-case scenarios.

Insertion sort is a simple and efficient sorting algorithm that works by inserting each element of an unsorted array into its correct position in a sorted subarray.insertion sort has a time complexity of O (n^2) .

Merge sort :- Merge Sort Algorithm breaks down a complex problem into simpler subproblems and solves them recursively. Merge sort is one of the most efficient and stable sorting algorithms, with a time complexity of O (n log n) in all cases, where n is

the number of elements in the array.

Quick sort :- Quick sort uses the divide and conquer approach, which means it breaks down a complex problem into simpler subproblems and solves them recursively. Quick sort is one of the most efficient and stable sorting algorithms, with a time complexity of O (n log n) in all of them.

Result:

Random Data :-

Inputs	**Number of Steps Executed (Random Data)**				
	Selection	**Bubble**	**Insertion**	**Merge**	**Quick**
1000	4914600	9390000	4545100	1876200	824300
2000	11608400	14864200	8216000	2256200	1246500
3000	13203400	23229700	11940400	2605600	1851900
4000	24219800	37923000	21692100	5090200	1976800
5000	25401800	52883800	13423000	3148100	2036300
Time Complexity→					

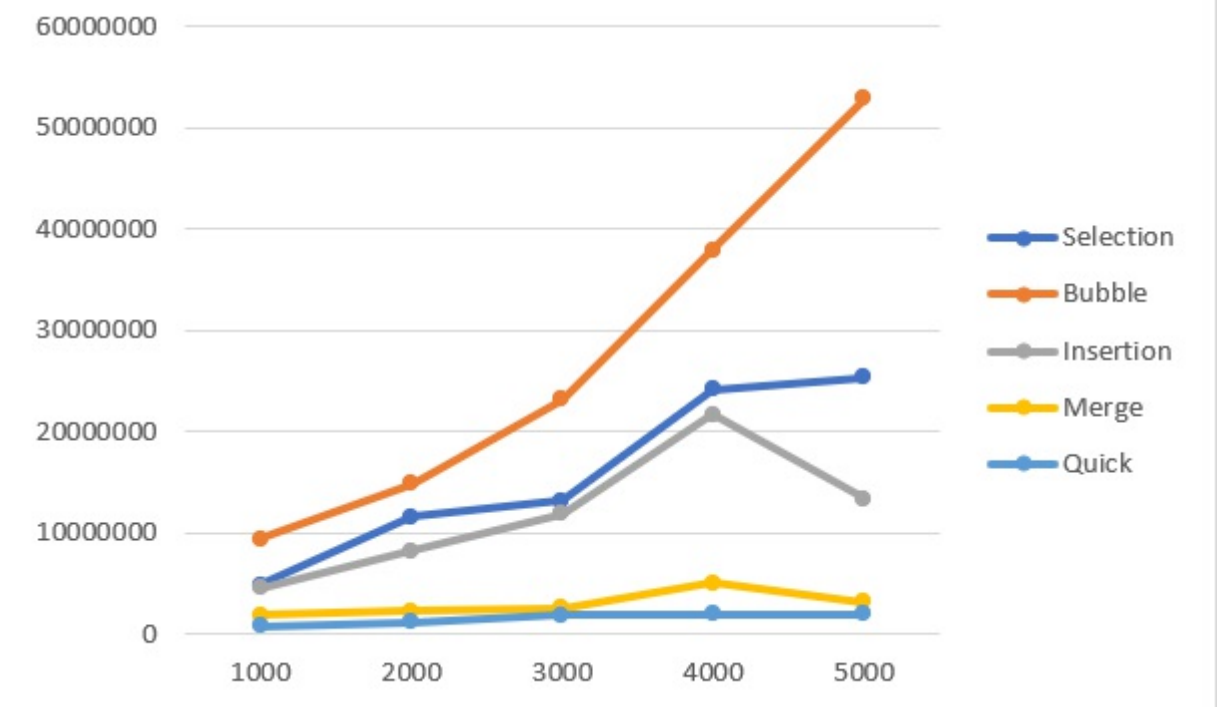
Ascending Order Sorted Data :-

Inputs	**Number of Steps Executed (Random Data)**				
	Selection	**Insertion**	**Bubble**	**Merge**	**Quick**
1000	5592200	3607400	8278000	1169399	611300
2000	7244700	11172600	11045199	1869100	1007300
3000	10191900	9600600	17759700	2627100	1333501
4000	14625900	12542800	25295600	2909600	1615400
5000	19721000	12885100	37334900	3113900	2354199
Time Complexity→					

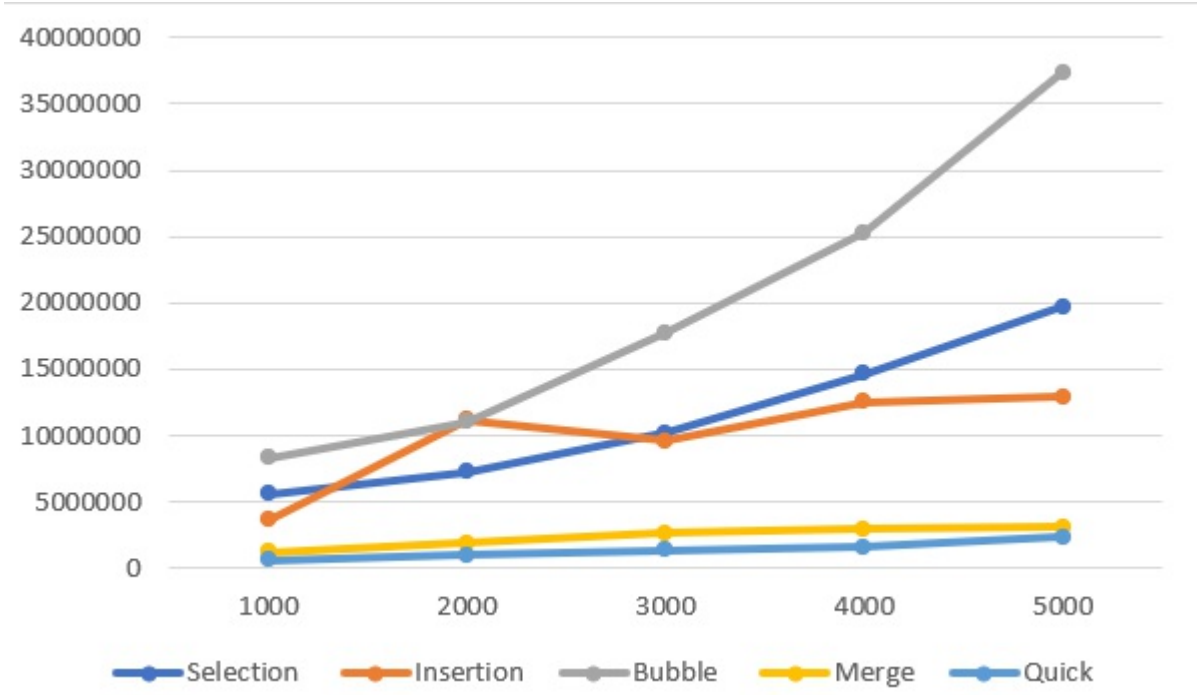
Descending Order Sorted Data :-

Inputs	**Number of Steps Executed (Random Data)**				
	Selection	**Insertion**	**Bubble**	**Merge**	**Quick**
1000	5610100	5176400	13481500	998200	526200
2000	7192000	13501600	10826200	1367500	1298600
3000	10278800	11526300	15637900	2288500	1538600
4000	14403400	13087300	25220900	2660000	1774600
5000	19605600	14555300	37648100	3413100	2002700
Time Complexity→					

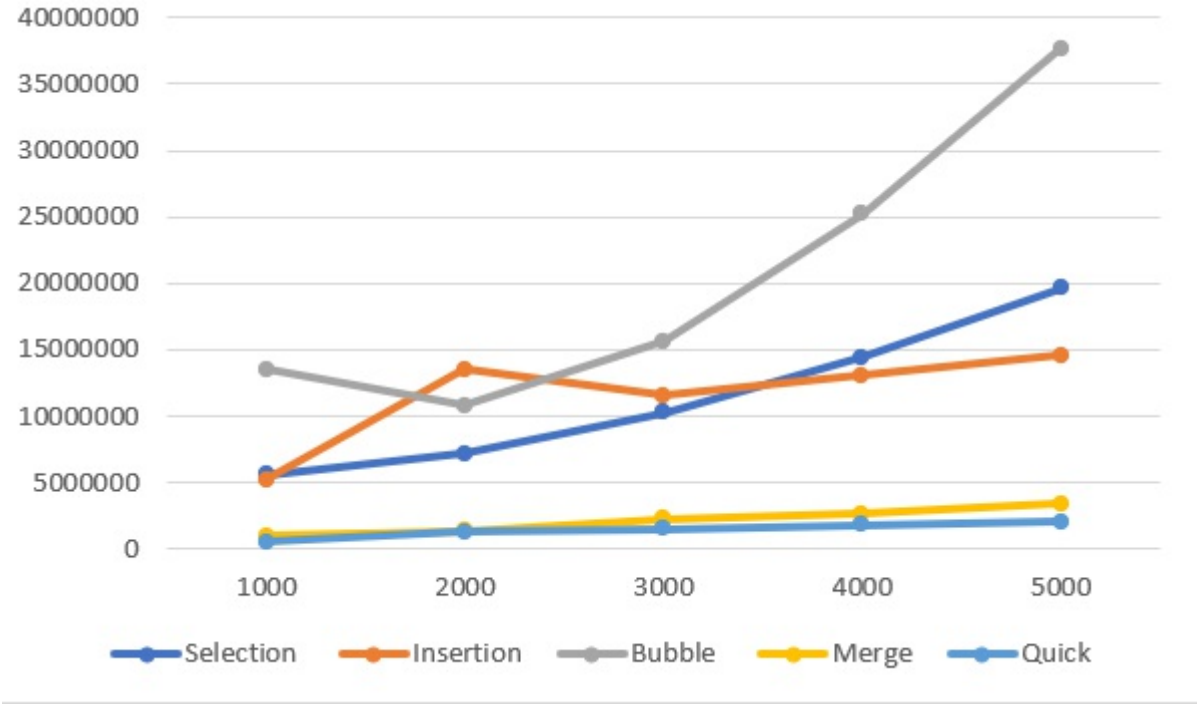
Chart:



Asceding Order Chart :-



Descending data :-



Conclusion:

As we see in above Charts and analyze it. We see that the line for bubble sort is increased with the number of input increased the growth is exponential. And the merge sort and quick sort had a nearly equal line for the small. Selection sort and insertion sort has a less inclined than bubble sort.

Quiz:

1. Which sorting function execute faster (has small steps count) in case of ascending order sorted data?

Answer:

The Quick sort function executes faster than all of them in ascending order sorting.

1. Which sorting function execute faster (has small steps count) in case of descending order sorted data?

Answer:

The merge sort and quick sort algorithms tend to execute faster and have smaller step counts compared to the other sorting algorithms.

1. Which sorting function execute faster (has small steps count) in case of random data?

Answer:

The performance of sorting algorithms on random data can vary depending on several factors, including the specific algorithm, the size of the data, and the distribution of values within the data. In general, quicksort can be considered efficient for sorting random data.

1. On what kind of data, the best case of Bubble sort occurs?

Answer:

when the input data is already sorted in ascending order if we have to arrange it in ascending order vise versa with the Descending order

1. On what kind of data, the worst case of Bubble sort occurs?

Answer:

The worst-case scenario for the bubble sort algorithm occurs when the input data is sorted in descending order if we have to arrange it in ascending order vise versa with Ascending order.

1. On what kind of data, the best case of Quick sort occurs?

Answer:

The best-case scenario for the quicksort algorithm occurs when the input data is evenly partitioned around the pivot element during each partitioning step. In this case, quicksort exhibits its optimal performance.

1. On what kind of data, the worst case of Quick sort occurs?

Answer:

Specifically, this worst-case scenario happens when the pivot chosen is either the smallest or largest element in the array during each partitioning step.

1. Which sorting algorithms are in-place sorting algorithms?

Answer:

In-place sorting algorithms are algorithms that sort a list or array of elements without requiring additional memory space proportional to the size of the input.

1. Which sorting algorithms are stable sorting algorithms?

Answer:

A stable sorting algorithm is one in which the relative order of equal elements in the sorted output remains the same as it was in the input.

Suggested Reference:

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

“Fundamentals of Algorithms” by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)	Program Implementation (5)	Documentation &Timely Submission (2)	T (1

Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)

Marks

EXPERIMENT NO: 3

Aim:

Implement a function of sequential search and count the steps executed by function on various inputs (1000 to 5000) for best case, average case and worst case. Also, write time complexity in each case and draw a comparative chart of number of input versus steps executed by sequential search for each case.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1

Objectives: (a) Identify Best, Worst and Average cases of given problem.

(b) Derive time complexity from steps count on various inputs.

Equipment/Instruments: Computer System, Any java language editor

Theory/Algorithm:

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Program :

```
import java.util.Random;

public class Sequentialsearch {

public static void main(String[] args) {
```

```
int[] array = generateArray(5000);

// Sequential Search - Best Case

long startTime = System.nanoTime();

sequentialSearch(array, array[0]);

long endTime = System.nanoTime();

long bestCaseTime = endTime - startTime;

System.out.println("Best Case Time Complexity: " + bestCaseTime + " ns");

// Sequential Search - Average Case

startTime = System.nanoTime();

sequentialSearch(array, array[array.length/2]);

endTime = System.nanoTime();

long averageCaseTime = endTime - startTime;

System.out.println("Average Case Time Complexity: " + averageCaseTime + " ns");

// Sequential Search - Worst Case

startTime = System.nanoTime();

sequentialSearch(array, -1);

endTime = System.nanoTime();

long worstCaseTime = endTime - startTime;

System.out.println("Worst Case Time Complexity: " + worstCaseTime + " ns");

}

private static int[] generateArray(int size) {

int[] array = new int[size];

Random random = new Random();

for (int i = 0; i < size; i++) {

array[i] = random.nextInt();

}

return array;

}

private static int sequentialSearch(int[] array, int target) {

for (int i = 0; i < array.length; i++) {

if (array[i] == target) {
```

```
return i;

}

}

return -1;

}

}
```

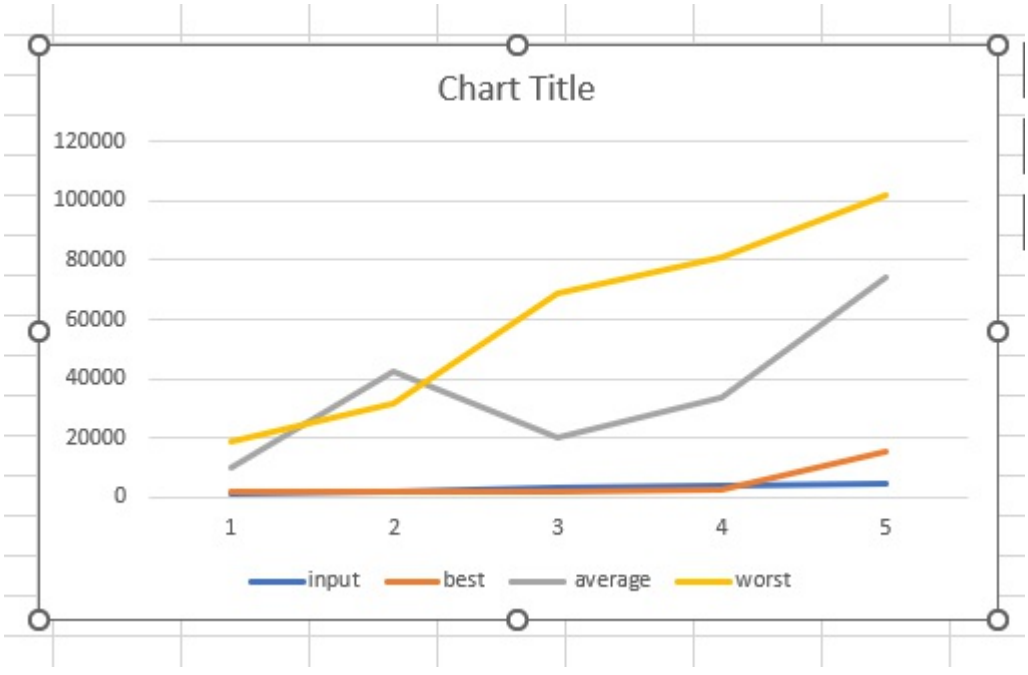
Observations:

It examines the first element in the list and then examines each "sequential" element in the list until a match is found. The Match Found will be returned and print it. The Array will traversed until it find the Match.

Result:

Inputs	**Number of Steps Executed**		
	Best Case	**Average Case**	**Worst Case**
1050	1800	9700	19000
2050	1700	42400	31800
3050	1500	20300	68700
3850	2100	33300	80900
4500	15600	74000	101900
Time Complexity→	O(1)	O(n/2)	O(n)

Chart:



Conclusion:

From above graph we can conclude that,

The best case in sequential search take less time in execution (nanoseconds) because search element is at 0^{th} position.

The average case in sequential search take average time in execution (nanoseconds) because search element is at any position other than 0^{th} position and element not found.

The worst case in sequential search take More time in execution (nanoseconds) because element will not found or we element which we have to find it found at last position of the array .

Quiz:

1. Which is the best case of an algorithm?

Answer:

A best case for the searching in the sequential search is the we found the element at the first position so the step will be reduced cause the loop will be terminated.

1. Which is the worst case of an algorithm?

Answer:

The worst case of an algorithm refers to the scenario in which the algorithm exhibits the maximum possible time or space complexity among all possible input instances or we can say that for the searching the element found at the last position or not found so the time complexity increased.

Suggested Reference:

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

“Fundamentals of Algorithms” by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

	Understanding of problem			Program Implementation			Documentation &Timely Submission			T (1
Rubrics	(3)			(5)			(2)			
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	

Marks

EXPERIMENT NO: 4

Aim:

Compare the performances of linear search and binary search for Best case, Average case and Worst case inputs.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1, CO2

Objectives: (a) Identify Best, Worst and Average cases of given problem.

(b) Derive time complexity from steps count for different inputs.

Equipment/Instruments: Computer System, Any C language editor

Theory/Algorithm:

Binary Search (Array A, Value x)

Step 1: start = 0 and end = n-1

Step 2: mid = (start + end)/2

Step 3: if mid = x then return A[mid]

Step 4: if mid > x then end = mid-1 Go to Step 2

Step 6: if mid < x then start = mid+1 and Go to Step 2

Step 7: Repeat Steps 2-5 until x is found or x > end

Step 8: Print Element x Found at index i and go to step 10

Step 9: if Element not found then return -1

Step 10: Exit

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Program:

```
import java.util.Arrays;
import java.util.Random;
public class BinarySearchExample1 {
public static void main(String[] args) {
int size = 500;
int[] arr = generateRandomArray(size);
Arrays.sort(arr);
int target = arr[new Random().nextInt(size)];
//System.out.println("Array: " + Arrays.toString(arr));
//System.out.println("Target: " + target);
int result = binarySearch(arr, target);
if (result == -1) {
System.out.println("Target not found.");
} else {
System.out.println("Target found at index: " + result);
}
// Best case scenario: target is the first element
int[] bestCaseArr = generateAscendingArray(size);
int bestCaseTarget = bestCaseArr[250];
long startTime1 = System.nanoTime();
int bestCaseResult = binarySearch(bestCaseArr, bestCaseTarget);
long endTime1 = System.nanoTime();
long duration1 = endTime1 - startTime1;
System.out.println("Best case scenario:");
System.out.println("Time taken: " + duration1 + " nanoseconds");
//System.out.println("Array: " + Arrays.toString(bestCaseArr));
System.out.println("Target: " + bestCaseTarget);
System.out.println("Target found at index: " + bestCaseResult);
// Worst case scenario: target is not in the array
int[] worstCaseArr = generateRandomArray(size);
```

```
Arrays.sort(worstCaseArr);
int worstCaseTarget = 1000;
long startTime2 = System.nanoTime();
int worstCaseResult = binarySearch(worstCaseArr, worstCaseTarget);
long endTime2 = System.nanoTime();
long duration2 = endTime2 - startTime2;
System.out.println("Worst case scenario:");
System.out.println("Time taken: " + duration2 + " nanoseconds");
//System.out.println("Array: " + Arrays.toString(worstCaseArr));
System.out.println("Target: " + worstCaseTarget);
System.out.println("Target not found.");
// Average case scenario: random target and array
int[] averageCaseArr = generateRandomArray(size);
Arrays.sort(averageCaseArr);
int averageCaseTarget = averageCaseArr[new Random().nextInt(size)];
long startTime3 = System.nanoTime();
int averageCaseResult = binarySearch(averageCaseArr, averageCaseTarget);
long endTime3 = System.nanoTime();
long duration3 = endTime3 - startTime3;
System.out.println("Average case scenario:");
System.out.println("Time taken: " + duration3 + " nanoseconds");
//System.out.println("Array: " + Arrays.toString(averageCaseArr));
System.out.println("Target: " + averageCaseTarget);
if (averageCaseResult == -1) {
System.out.println("Target not found.");
} else {
System.out.println("Target found at index: " + averageCaseResult);
}
}
public static int[] generateRandomArray(int size) {
int[] arr = new int[size];
```

```
Random random = new Random();
for (int i = 0; i < size; i++) {
arr[i] = random.nextInt(size * 10);
}
return arr;
}
public static int[] generateAscendingArray(int size) {
int[] arr = new int[size];
for (int i = 0; i < size; i++) {
arr[i] = i;
}
return arr;
}
public static int binarySearch(int[] arr, int target) {
int left = 0;
int right = arr.length - 1;
while (left <= right) {
int mid = left + (right - left) / 2;
if (arr[mid] == target) {
return mid;
} else if (arr[mid] < target) {
left = mid + 1;
} else {
right = mid - 1;
}
}
return -1;
}
}
```

Observations:

Write observation based on number of steps executed by both algorithms.

Linear search examines the first element in the list and then examines each "sequential" element in the list until a match is found. This match could be a desired work that you are searching for, or the minimum number in the list.

Binary Search first important thing is that the array must be sorted in ascending order after that it will half the array and break it into two parts in this there are three cases

first the searched element is at the middle of the array then it is called the best-case scenario

Second if the searched element is at any position, then it is said to be and average case scenario

Third if the searched element is not in the array is said to in worst case scenario.

It works by repeatedly dividing the search space in half and comparing the target value with the middle element of the array

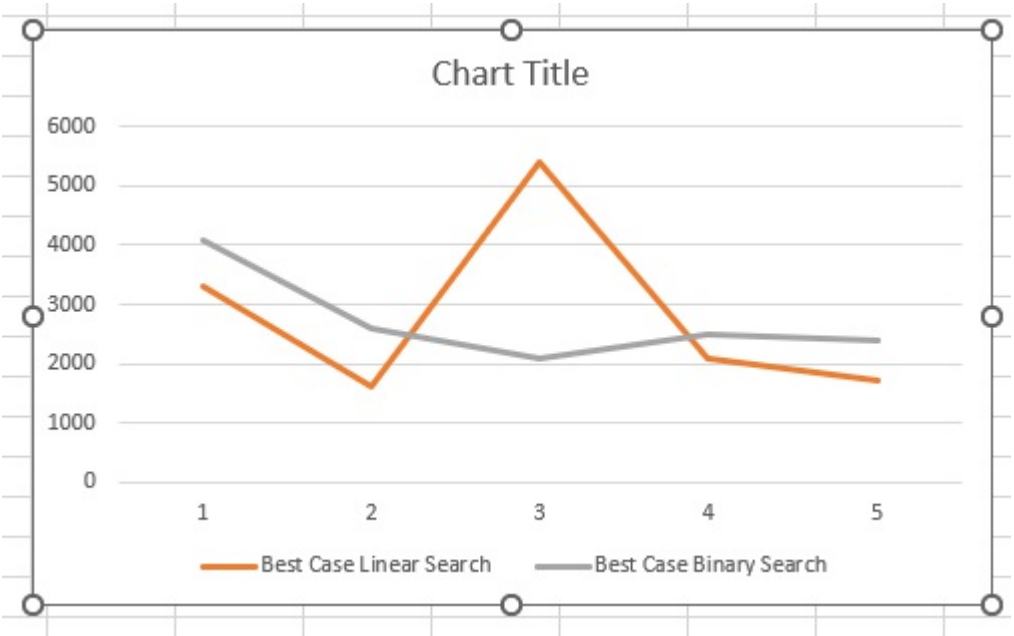
Result: Complete the below table based on your implementation of sequential search algorithm and steps executed by the function.

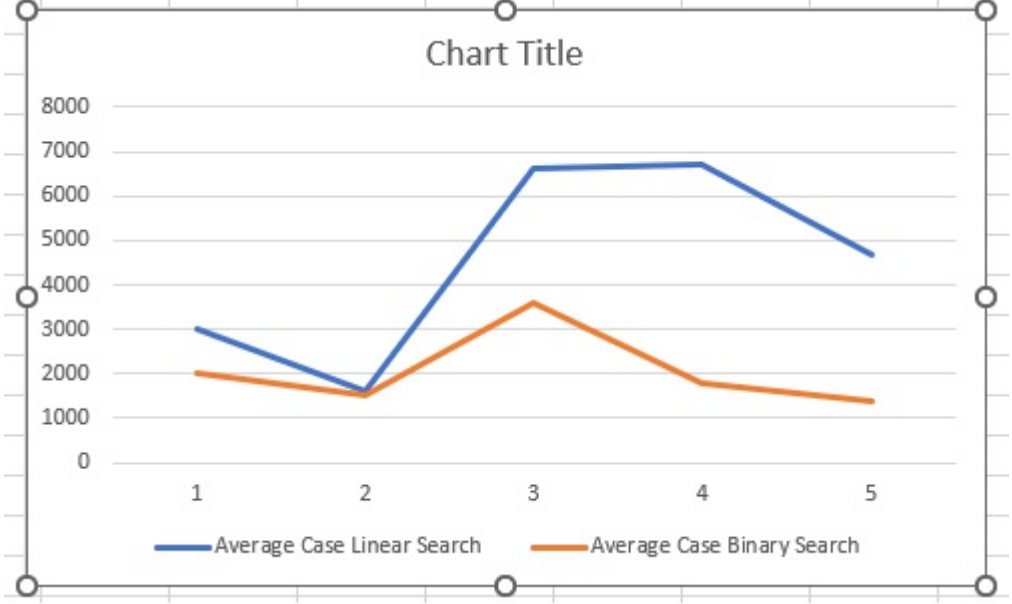
Inputs	**Best Case**		**Average Case**		**Worst Case**	
	Linear Search	**Binary Search**	**Linear Search**	**Binary Search**	**Linear Search**	**Binary Search**
105	3300	4100	3000	2000	4400	1700
200	1600	2600	1600	1500	3700	1800
285	5400	2100	6600	3600	7000	1500
350	2100	2500	6700	1800	6700	1600
410	1700	2400	4700	1400	8400	1400
Time Complexity	O(1)	O(1)	O(n)	O(logn)	O(n)	O(logn)

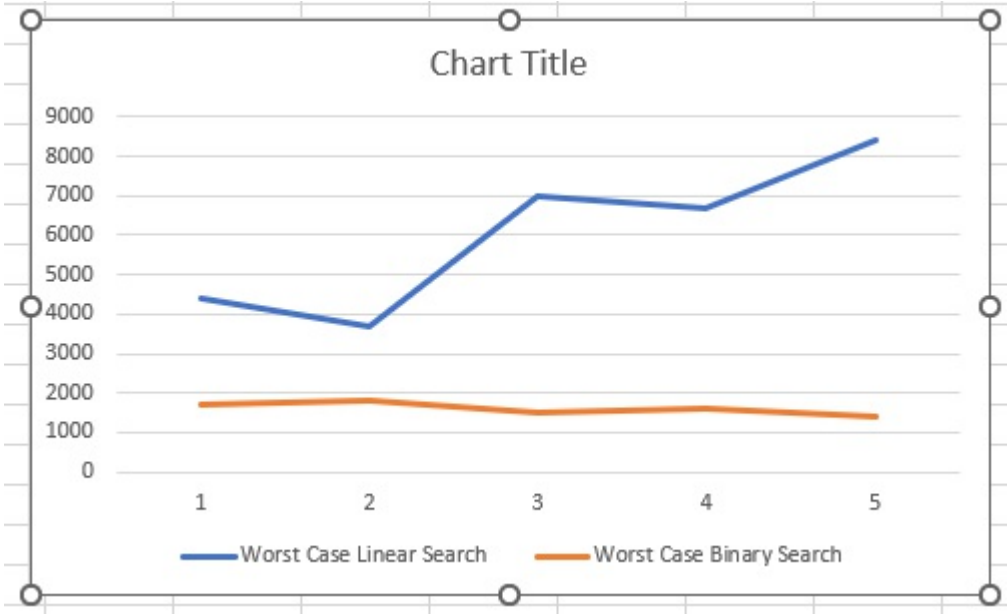
Prepare similar tables for Average case and Worst case of both algorithms.

Chart:

<Draw Comparative Charts of inputs versus number of steps executed by both algorithms in various cases>







Conclusion:

Linear search scans each element one by one until it finds the target element or reaches the end of the array.

Binary Search Divide the array into two halfs and compare with the target element with the middle element of each half. So the Worst case time complexity is O(log n) while linear search requires O(n).

Linear search is simpler to implement for the two dimension array but we can't implement the Binary Search Algorithm

Quiz:

1. Which element should be searched for the best case of binary search algorithm?

Answer: An element that is located exactly at the middle of the sorted array should be searched.

1. Which element should be searched for the worst case of binary search algorithm?

Answer: An element that is either not present in the sorted array or is located at one of the extreme ends of the array (either the first element or the last element) should be searched**.**

1. Which algorithm executes faster in worst case?

Answer: In general, binary search executes faster in the worst-case scenario compared to linear search.

Suggested Reference:

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

“Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			T (1
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										