A Laboratory Manual for

Analysis and Design of Algorithms (3150703)

B.E. Semester 5 (Computer Engineering)

Directorate of Technical Education, Gandhinagar, Gujarat

Vishwakarma Government Engineering College

Chandkheda, Ahmedabad

Certificate

This is to certify that Mr./Ms. **Mehta Yashesh Jigarkumar** Enrollment No. **230170107155** of B.E. Semester **5** Batch No. **H** Computer Engineering of this Institute (GTU Code: **017**) has satisfactorily completed the Practical / Tutorial work for the subject **Analysis and Design of Algorithms (3150703)** for the academic year 2025.

Place: Ahmedabad

Date: _____

Name and Sign of 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 non-polynomial 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	CO 2	CO 3	CO 4	CO 5	CO 6
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. To calculate sum of 1 to N numbers using loop. To calculate sum of 1 to N numbers using equation. 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). • Selection Sort • Bubble Sort • Insertion Sort • Merge Sort • 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

Guidelines for Faculty members

1. Teacher should provide the guideline with demonstration of practical to the students with all features.
2. Teacher shall explain basic concepts/theory related to the experiment to the students before starting of each practical
3. Involve all the students in performance of each experiment.
4. Teacher is expected to share the skills and competencies to be developed in the students and ensure that the respective skills and competencies are developed in the students after the completion of the experimentation.
5. Teachers should give opportunity to students for hands-on experience after the demonstration.
6. Teacher may provide additional knowledge and skills to the students even though not covered in the manual but are expected from the students by concerned industry.
7. Give practical assignment and assess the performance of students based on task assigned to check whether it is as per the instructions or not.
8. Teacher is expected to refer complete curriculum of the course and follow the guidelines for implementation.

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	Assessment Marks	Sign. of Teacher with date	Remarks
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. • To calculate sum of 1 to N numbers using loop. • To calculate sum of 1 to N numbers using equation. • 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.	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.						
Total							

Experiment No: 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 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, Any C language editor

Theory:

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

1. Take an input value for N.
2. Initialize a variable sum to zero.
3. Start a loop that iterates from i=1 to i=N.
4. In each iteration, add the value of i to the sum variable.
5. After the loop completes, output the value of sum.

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

1. Take an input value for N.
2. Calculate sum as N*(N+1)/2.
3. Output the value of sum.

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

1. Take an input value for N.
2. Define a recursive function sum (n) that takes an integer argument n and returns the sum of 1 to n.
3. In the sum (n) function, check if n equals 1. If it does, return 1 (the base case).
4. Otherwise, return the sum of n and the result of calling sum (n-1).
5. Output the result.

Implement three functions based on above steps and calculate the number of steps executed by each functions on various inputs ranging from 100 to 500. Take a counter variable to calculate the number of steps and increment it for each statement in the function.

Code:

```
import java.util.*;
import java.io.FileWriter;
import java.io.IOException;

public class Practical_1 {
   // counter variables for each function
   static int loopCount = 0;
   static int eqCount = 0;
   static int recCount = 0;

   // Sum of 1 to N num. using loop
   public static void sumLoop(int n) {
      loopCount = 0;
      if(n<=0) {
         System.out.println("Enter valid number!");
         return;
      }
      int sum = 0;
      loopCount++;
      for(int i=1; i<=n; i++) {
         loopCount++;
         sum+=i;
         loopCount++;
      }
      loopCount++;

      // System.out.println("Sum : " + sum);
      return;
   }

   // Sum of 1 to N num. using eaquation
   public static void sumEquation(int n) {
      eqCount = 0;
      if(n<=0) {
         System.out.println("Enter valid number!");
         return;
      }
      int sum = n * (n + 1) / 2; eqCount++;
      // System.out.println("Sum : " + sum);
      return;
   }
```

```
   // Sum of 1 to N num. using recursion
   public static int sumRecursion(int n) {
      recCount++;
      if(n<=0) {
         return 0;
      }
      return n + sumRecursion(n-1);
   }
   public static void main(String[] args) throws IOException{
      int[] inputs = {100, 200, 300, 400, 500};

      FileWriter writer = new FileWriter("steps_output.txt");
      writer.write("Input,Loop,Equation,Recursion\n");

         System.out.printf("%-10s %-15s %-15s %-15s", "Input", "Loop Method",
"Equations", "Recursion");
      System.out.println();

      for(int n : inputs) {
         sumLoop(n);
         sumEquation(n);
         recCount = 0;
         sumRecursion(n);

         writer.write(n + "," + loopCount + "," + eqCount + "," + recCount + "\n");

            System.out.printf("%-10d %-15d %-15d %-15d", n, loopCount, eqCount,
recCount);
         System.out.println();
      }

      writer.close();
   }
}
```

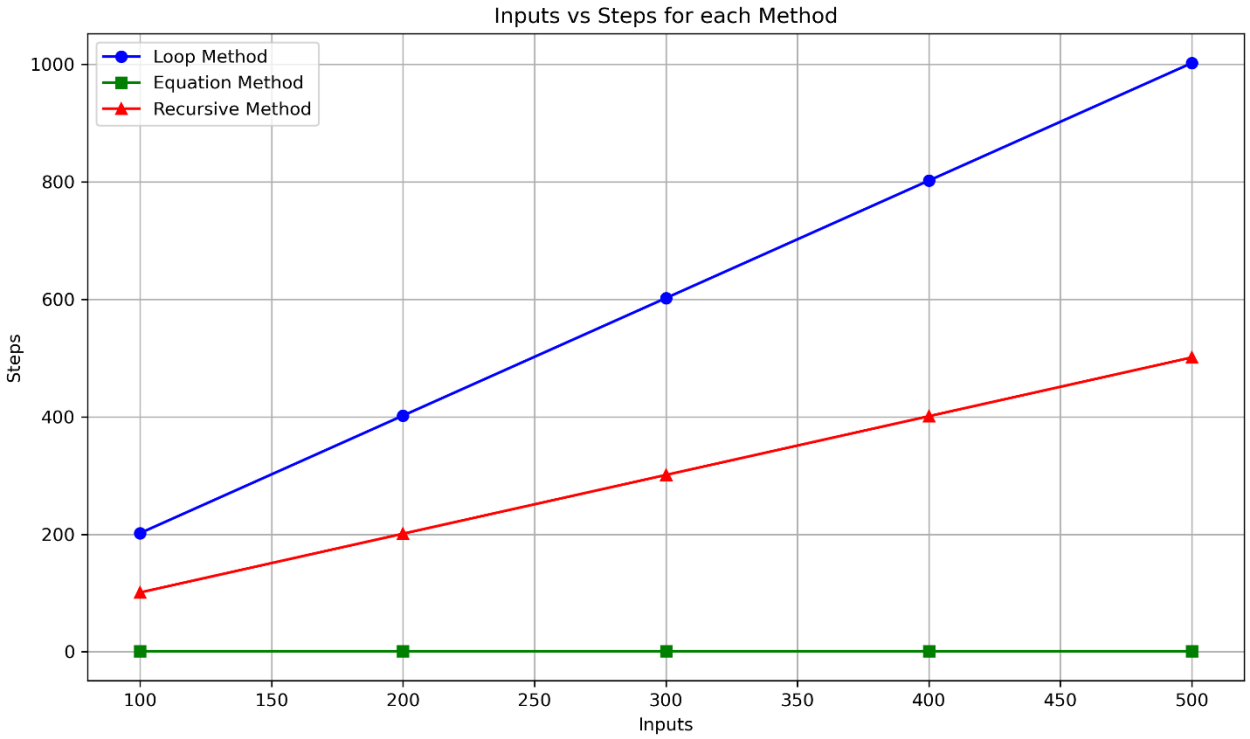
Observations:

- The **Loop** and **Recursion** methods show **linear growth** in steps with increasing input.
- The **Equation** method consistently executes in **1 step**, regardless of input size.
- The loop takes slightly more steps than recursion due to additional operations.

Result: Complete the below table based on your implementation of functions and steps executed by each function.

Inputs	Number of Steps Executed		
	Loop method	**Equations**	**Recursion**
100	202	1	101
200	402	1	201
300	602	1	301
400	802	1	401
500	1002	1	501
Equation→		**n(n+1)/2**	

Chart:



Conclusion:

The Equation method is the most efficient due to its constant time complexity (O(1)). Loop and Recursion methods are linear (O(n)), with the loop being slightly less efficient. Choosing an optimal algorithm based on growth rate improves performance. Hence, in terms of performance and scalability, the Equation method is the best choice.

Quiz:

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

Answer:

- A constant growth rate means the algorithm takes the same number of steps regardless of the input size.
- It has a time complexity of **O(1)**, meaning its execution time does not change as the input increases.

2. 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:

- The algorithm with a growth rate of **n (O(n))** executes faster than the one with **n^2 ($O(n^2)$)**, especially for large input sizes.
- This is because in $O(n^2)$, the number of steps increases quadratically, while in O(n), it increases linearly.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms” by E. Horowitz et al.

References used by the students:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. https://www.w3schools.com/python/matplotlib_intro.asp

Rubric wise marks obtained:

Rubrics	**Understanding of problem (3)**			**Program Implementation (5)**			**Documentation &Timely Submission (2)**			**Total (10)**
	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: 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 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. **Selection Sort Function in C**

```
Void SelectionSort (int a[] , int n)
//Here 'a' is the array having 'n' number of data
{
    intMinIndex,temp;
    for(inti=0;i<n-1;i++)
    {
            MinIndex=i;
            for(int j=i+1;j<n;j++)
            {
                    if (a[MinIndex]>a[j])
                            MinIndex=j;
            }
            if(MinIndex !=i)
            {
                    temp=a[MinIndex];
                    a[MinIndex] = a[j];
                    a[j]=temp;
            }

    }
}
```

2. Bubble Sort Function in C

```
Void BubbleSort (inta[], int n)
//Here ‘a’ is the array having ‘n’ number of data
{
    intSwap_flag,temp;
    for(inti=0;i<n-1;i++)
    {
            Swap_flag=0;
            for(int j=0; j<n-i-1; j++)
            {
                    if (a[j]>a[j+1])
                    {
                            temp=a[j];
                            a[j]=a[j+1];
                            a[j+1]=temp;
                            Swap_flag=1;
                    }
            }
            if(swap_flag==0)
                    return;
    }}
```

3. Insertion Sort Function in C

```
Void InsertionSort (inta[],int n)
//Here ‘a’ is the array having ‘n’ number of data
{
    inti,j,key;
    for(j=1;j<n;j++)
    {
            key=a[j];
            i=j-1;
            while(i>=0 && a[i]>key)
            {
                    a[i+1] = a[i];
                    i= i-1;
            }
            a[i+1] =key;
    }}
```

4. Merge Sort Function in C

```
void merge(inta[],int low, intmid,int high)
{
    int temp[5000];
    inti,j,k;
    k = low;
    j = mid +1;
    i = low;
    while((k<=mid) && (j<=high))
    {
if(a[k]<=a[j])
```

```
            {       temp[i] = a[k] ;
                    k++;
            }
            else
            {       temp[i] = a[j] ;
                    j++;
            }
            i++;
    }
    if(k<=mid)
    for(;k<=mid;k++)
            temp[i++] = a[k];
    else
    for(;j<=high;j++)
            temp[i++]=a[j];
    for(i=low;i<=high;i++)a[i]=temp[i];
}
voidmerge_sort(int *a, long int low, long int high ) //call this function from main
{
    if(low!=high)
    {
            int mid = (low+high)/2;
            merge_sort(a,low,mid);
            merge_sort(a,mid+1,high);
            merge(a,low,mid,high);
    }}
```

5. Quick Sort Function in C

```
int quicksort(int q[],intlb, intub)
{
    int flag=1;
    inti=0,j,key,t1,t2;
    if(lb<ub)
    {
            i=lb;
            j=ub+1;
            key=q[lb];

            while(flag==1)
            {
                    i++;
                    while(q[i]<key)
                    {
                            i++;
                    }
                    j--;
                    while(q[j]>key)
                            j--;
                    if(i<j)
                    {
                            t1=q[i];
                            q[i]=q[j];
```

```
                        q[j]=t1;
                }
                else{
                        flag=0;
                }
        }
            t2=q[lb];
            q[lb]=q[j];
            q[j]=t2;

    quicksort(q,lb,j-1);
    quicksort(q,j+1,ub);
    }
    return;
}
```

Write main function to use above sorting functions and calculate the number of steps executed by each functions on various inputs ranging from 1000 to 5000. Take a counter variable to calculate the number of steps and increment it for each statement in the function.

Below is the sample function that counts the number of steps:

```
voidbubble_sort(int a[], int n)
{
        int swap_flag,i,j,temp;

        steps++; //for int, it is a global variable

        for(i=0;i<n-1;i++)
        {
                steps++; //for loop
                steps++; //for swap
                swap_flag = 0;
                for(j=0;j<n-i-1;j++)
                {
                        steps++; //for inner loop
                        steps++; //for if
                        if(a[j]>a[j+1])
                        {
                                temp = a[j];
                                a[j] = a[j+1];
                                a[j+1] = temp;
                                swap_flag = 1;
                                steps++; //
                                steps++; //
                                steps++; //for four steps
                                steps++; //
                        }
                }
                steps++; //for inner loop

                steps++; //for if
                if(swap_flag==0)
                {
```

```
                        steps++; //for return
                        return;
                }
        }
        steps++; //for outer loop false
}
```

Code:

```
import java.util.*;
import java.io.FileWriter;
import java.io.IOException;

public class Practical_2 {
   static long steps = 0;

   public static void selectionSort(int[] arr) {
      steps = 0;
      int n = arr.length;
      for (int i = 0; i < n - 1; i++) {
         steps++; // outer loop
         int minIndex = i;
         for (int j = i + 1; j < n; j++) {
            steps++; // inner loop
            if (arr[minIndex] > arr[j]) {
               minIndex = j;
               steps++;
            }
         }
         if (minIndex != i) {
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
            steps += 3; // swaps
         }
      }
   }

   public static void bubbleSort(int[] arr) {
      steps = 0;
      int n = arr.length;
      for (int i = 0; i < n - 1; i++) {
         steps++;
         boolean swapped = false;
         for (int j = 0; j < n - i - 1; j++) {
            steps++;
```

```
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                steps += 4;
                swapped = true;
            }
        }
        if (!swapped) {
            steps++;
            break;
        }
    }
}

public static void insertionSort(int[] arr) {
    steps = 0;
    int n = arr.length;
    for (int i = 1; i < n; i++) {
        steps++;
        int current = arr[i];
        int j = i - 1;
        steps++;
        while (j >= 0 && current < arr[j]) {
            arr[j + 1] = arr[j];
            j--;
            steps += 2;
        }
        arr[j + 1] = current;
        steps++;
    }
}

public static void mergeSort(int[] arr) {
    steps = 0;
    divide(arr, 0, arr.length - 1);
}

private static void divide(int[] arr, int low, int high) {
    if (low < high) {
        steps++;
        int mid = low + (high - low) / 2;
        divide(arr, low, mid);
        divide(arr, mid + 1, high);
```

```
            conquer(arr, low, mid, high);
        }
    }

    private static void conquer(int[] arr, int low, int mid, int high) {
        int[] merged = new int[high - low + 1];
        int idx1 = low, idx2 = mid + 1, x = 0;

        while (idx1 <= mid && idx2 <= high) {
            steps++;
            if (arr[idx1] <= arr[idx2]) merged[x++] = arr[idx1++];
            else merged[x++] = arr[idx2++];
        }
        while (idx1 <= mid) merged[x++] = arr[idx1++];
        while (idx2 <= high) merged[x++] = arr[idx2++];
        for(int i=0, j=low; i<merged.length; i++, j++) {
            arr[j] = merged[i];
        }
        steps += (high - low + 1); // copying back
    }

    public static void quickSort(int[] arr) {
        steps = 0;
        quickSortHelper(arr, 0, arr.length - 1);
    }

    private static void quickSortHelper(int[] arr, int low, int high) {
        if (low < high) {
            steps++;
            int pivotIdx = partition(arr, low, high);
            quickSortHelper(arr, low, pivotIdx - 1);
            quickSortHelper(arr, pivotIdx + 1, high);
        }
    }

    private 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;
            steps += 3;
          }
        }
        i++;
        int temp = arr[i];
        arr[i] = pivot;
        arr[high] = temp;
        steps += 3;
        return i; // pivot index
    }

    public static int[] generateRandomArray(int n) {
        int[] arr = new int[n];
        Random rand = new Random();
        for(int i=0;i<n;i++) arr[i] = rand.nextInt(n);
        return arr;
    }

    public static int[] generateAscendingArray(int n) {
        int[] arr = new int[n];
        for(int i=0;i<n;i++) arr[i] = i;
        return arr;
    }

    public static int[] generateDescendingArray(int n) {
        int[] arr = new int[n];
        for(int i=0;i<n;i++) arr[i] = n-i;
        return arr;
    }

    public static void main(String[] args) throws IOException {
        int[] sizes = {1000, 2000, 3000, 4000, 5000};
        String[] types = {"Random", "Ascending", "Descending"};

        for (String type : types) {
            System.out.println("\n--- " + type + " Data ---");
                System.out.printf("%-10s %-10s %-10s %-10s %-10s %-10s\n", "Input",
"Selection", "Bubble", "Insertion", "Merge", "Quick");

            String fileName = type.toLowerCase() + ".txt";

            try (FileWriter writer = new FileWriter(fileName)) {
```

```
            writer.write("Input,Selection,Bubble,Insertion,Merge,Quick\n");  // CSV-
style header

            for (int size : sizes) {
                int[] original = switch (type) {
                    case "Ascending" -> generateAscendingArray(size);
                    case "Descending" -> generateDescendingArray(size);
                    default -> generateRandomArray(size);
                };

                int[] a, b, c, d, e;

                a = Arrays.copyOf(original, size);
                selectionSort(a);
                long sel = steps;

                b = Arrays.copyOf(original, size);
                bubbleSort(b);
                long bub = steps;

                c = Arrays.copyOf(original, size);
                insertionSort(c);
                long ins = steps;

                d = Arrays.copyOf(original, size);
                mergeSort(d);
                long mer = steps;

                e = Arrays.copyOf(original, size);
                quickSort(e);
                long qui = steps;

                  System.out.printf("%-10d %-10d %-10d %-10d %-10d %-10d\n", size,
sel, bub, ins, mer, qui);

                // Write to file
                 writer.write(size + "," + sel + "," + bub + "," + ins + "," + mer + "," + qui +
"\n");

            }
        }
    }
  }
}
```

Observations:

Write observation based on number of steps executed by each algorithm.

- On random data, Merge and Quick Sort perform best.
- On ascending data, Bubble and Insertion Sort show best-case performance (early exits).
- On descending data, Bubble and Insertion Sort show worst-case performance (many swaps).

Result: Complete the below table based on your implementation of functions and steps executed by each function. **Also, prepare similar tables for ascending order sorted data and descending order sorted data.**

Inputs	Number of Steps Executed (Random Data)				
	Selection	**Bubble**	**Insertion**	**Merge**	**Quick**
1000	508747	1498449	502381	19687	18284
2000	2018887	6096382	2053733	43346	37514
3000	4528875	13509314	4514111	68875	66789
4000	8039598	24105380	8065347	94743	89859
5000	12552029	37614813	12579663	121994	111119
Time Complexity→	**O(n^2)**	**O(n^2)**	**O(n^2)**	**O(nlogn)**	**O(nlogn)**

Inputs	Number of Steps Executed (Ascending Order Sorted Data)				
	Selection	**Bubble**	**Insertion**	**Merge**	**Quick**
1000	500499	1001	2997	16019	1502496
2000	2000999	2001	5997	35039	6004996
3000	4501499	3001	8997	55979	13507496
4000	8001999	4001	11997	76079	24009996
5000	12502499	5001	14997	98811	37512496
Time Complexity→	**O(n^2)**	**O(n)**	**O(n)**	**O(nlogn)**	**O(nlogn)**

Inputs	Number of Steps Executed (Descending Order Sorted Data)				
	Selection	**Bubble**	**Insertion**	**Merge**	**Quick**
1000	751999	2498499	1001997	15907	752496
2000	3003999	9996999	4003997	34815	3004996
3000	6755999	22495499	9005997	54731	6757496
4000	12007999	39993999	16007997	75631	12009996
5000	18759999	62492499	25009997	96611	18762496
Time Complexity→	**O(n^2)**	**O(n^2)**	**O(n^2)**	**O(nlogn)**	**O(n^2)**

Chart:

Sample Chart is as below:

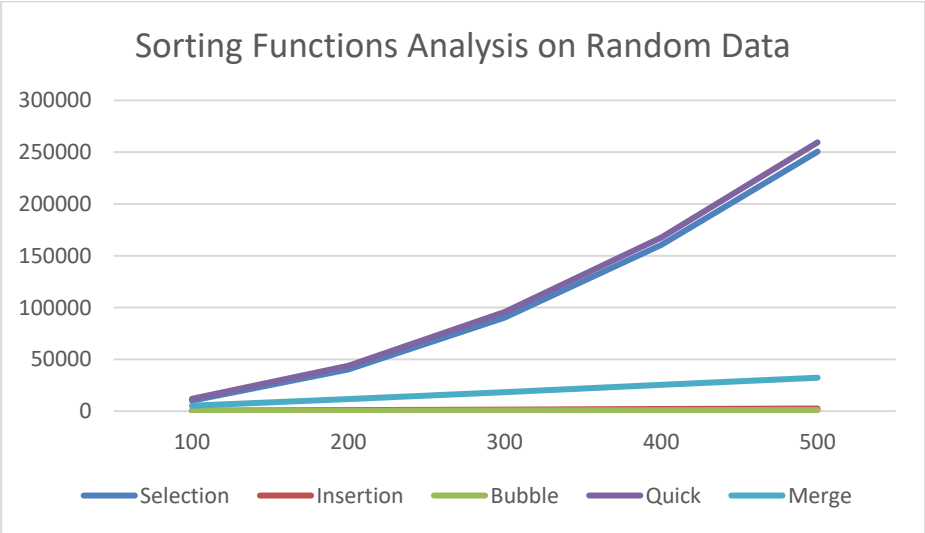


Chart for Random Data:

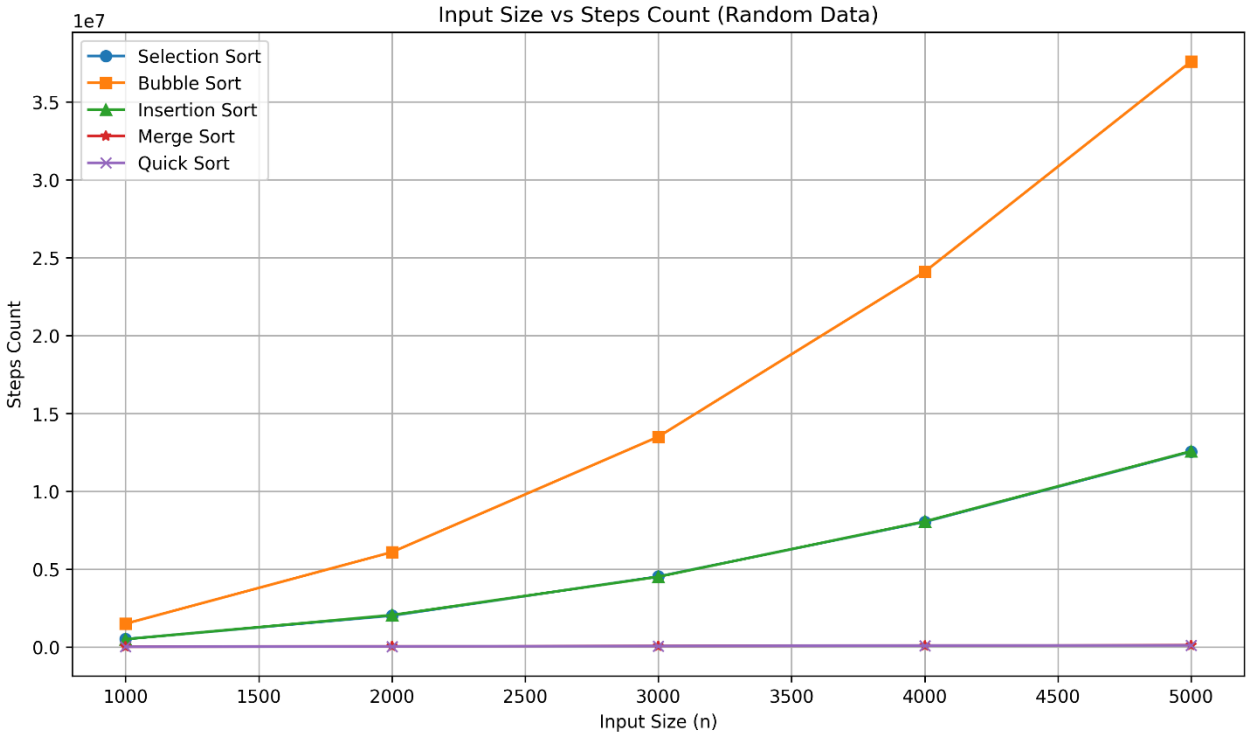


Chart for Ascending Order Sorted Data:

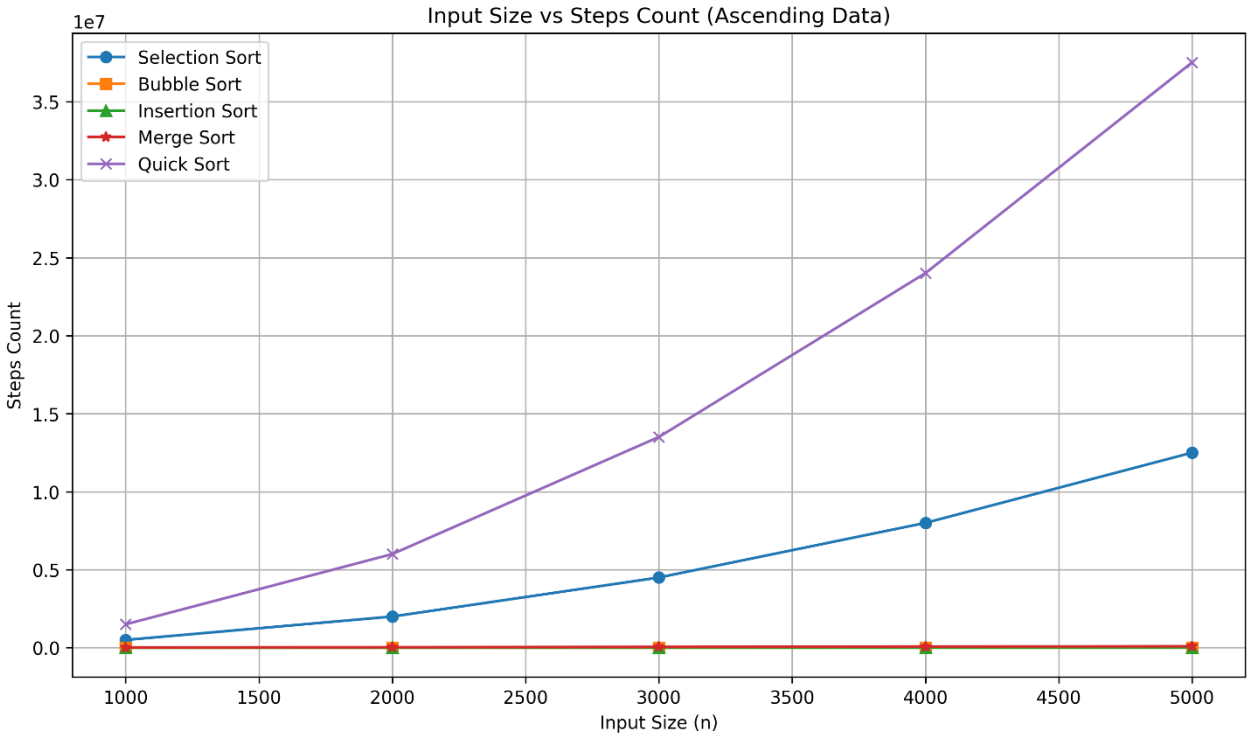
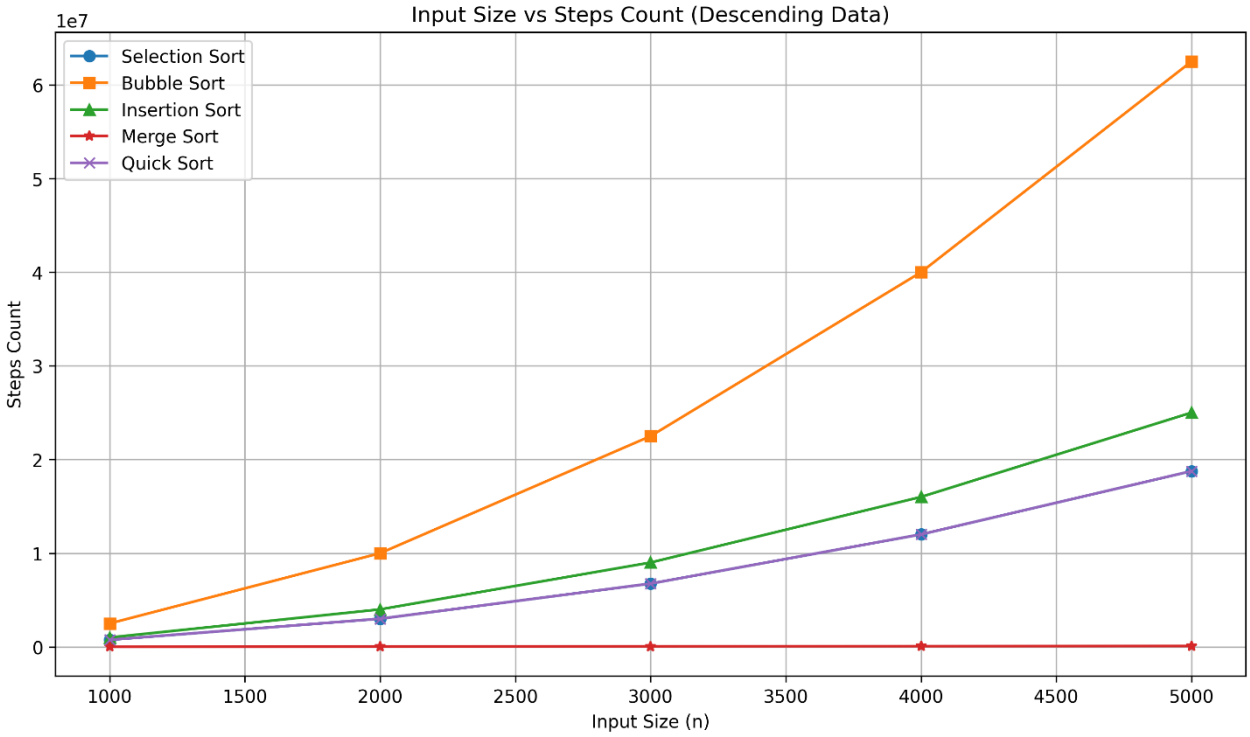


Chart for Descending Order Sorted Data:



Conclusion:

- Merge Sort and Quick Sort are the most efficient for large random data (O(n log n)).
- Bubble and Insertion Sort are inefficient on large or descending data ($O(n^2)$), but good on nearly sorted data.
- Selection Sort consistently performs the same ($O(n^2)$), regardless of input order.

Quiz:

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

Answer: *Insertion Sort* — Because it only compares without shifting elements when data is already sorted, resulting in just O(n) steps.

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

Answer: *Merge Sort* — It consistently performs O(nlogn) steps regardless of the data order, while others like Bubble and Insertion degrade to O(n^2).

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

Answer: *Quick Sort* — It is generally the fastest on average for random data due to efficient partitioning, typically taking (nlogn) steps.

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

Answer: *Already sorted data* — If optimized (like in your code), Bubble Sort will exit early without swaps, taking only O(n)steps.

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

Answer: *Reverse sorted data* — It performs the maximum number of comparisons and swaps, resulting in O(n^2) time.

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

Answer: *Balanced partition data* — When the pivot divides the array into two equal halves each time, leading to optimal O(nlogn) performance.

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

Answer: *Sorted or reverse sorted data* — If the pivot is poorly chosen (like always last element), partitions become unbalanced, leading to O(n^2).

8. **Which sorting algorithms are in-place sorting algorithms?**

Answer: S*election, Bubble, Insertion, and Quick Sort* — These use constant extra space, sorting within the original array without needing additional memory.

9. **Which sorting algorithms are stable sorting algorithms?**

Answer: *Bubble Sort, Insertion Sort, and Merge Sort* — They maintain the relative order of equal elements, which is essential in many applications.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

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

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	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

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 C language editor

Theory:

Steps to implement sequential search is as below:

1. Take an input array A of n elements and a key value K.
2. Define a variable pos, initially set to -1.
3. Iterate through the array A, starting from the first element and continuing until either the key value is found or the end of the array is reached.
4. For each element, compare its value to the key value K.
5. If the values match, set pos to the index of the current element and exit the loop.
6. If the end of the array is reached and the key value has not been found, pos remain equal to -1.
7. Output the value of pos.

The algorithm works by sequentially iterating through the elements of the array and comparing each element to the target value. If a match is found, the algorithm exits the loop.

Implement above functions and calculate the number of steps executed by each functions on various inputs ranging from 1000 to 5000. Take a counter variable to calculate the number of steps and increment it for each statement. Based on algorithm's logic, decide best, worst and average case inputs for the algorithm and prepare a table of steps count.

Code:

```
import java.io.FileWriter;
import java.io.IOException;

public class Practical_3 {
   static int steps = 0;

   public static int sequentialSearch(int arr[], int key) {
```

```
        steps = 0;
        int pos = -1;
        for (int i = 0; i < arr.length; i++) {
            steps++;
            if (arr[i] == key) {
                pos = i;
                break;
            }
        }
        return pos;
    }
    public static void main(String[] args) {
        int[] sizes = {1000, 2000, 3000, 4000, 5000};

        try {
            FileWriter writer = new FileWriter("SequentialSearchSteps.txt");

            // Writing Header
                writer.write("Inputs"+","+"Best  Case"+","+"Average  Case"+","+"Worst
Case\n");

            // Print Console Table Header
                System.out.printf("%-10s %-15s %-15s %-15s\n", "Inputs", "Best Case",
"Average Case", "Worst Case");
                System.out.println("-----------------------------------------------------------
");

            for (int size : sizes) {
                int[] arr = new int[size];
                for (int i = 0; i < size; i++) {
                    arr[i] = i + 1;
                }

                // Best Case
                sequentialSearch(arr, arr[0]);
                int best = steps;

                // Average Case
```

```
            sequentialSearch(arr, arr[size / 2]);
            int avg = steps;

            // Worst Case
            sequentialSearch(arr, arr[size - 1]);
            int worst = steps;

            writer.write(size + "," + best + "," + avg + "," + worst + ","+"\n");

            // Print to console
            System.out.printf("%-10d %-15d %-15d %-15d\n", size, best, avg, worst);
        }

        // Time Complexity row
            System.out.println("------------------------------------------------------------
");
        System.out.printf("%-10s %-15s %-15s %-15s\n", "", "O(1)", "O(n)", "O(n)");

        writer.close();
              // System.out.println("Data written to SequentialSearchSteps.txt
successfully.");
      } catch (IOException e) {
          System.out.println("An error occurred while writing to file.");
          e.printStackTrace();
      }
   }
}
```

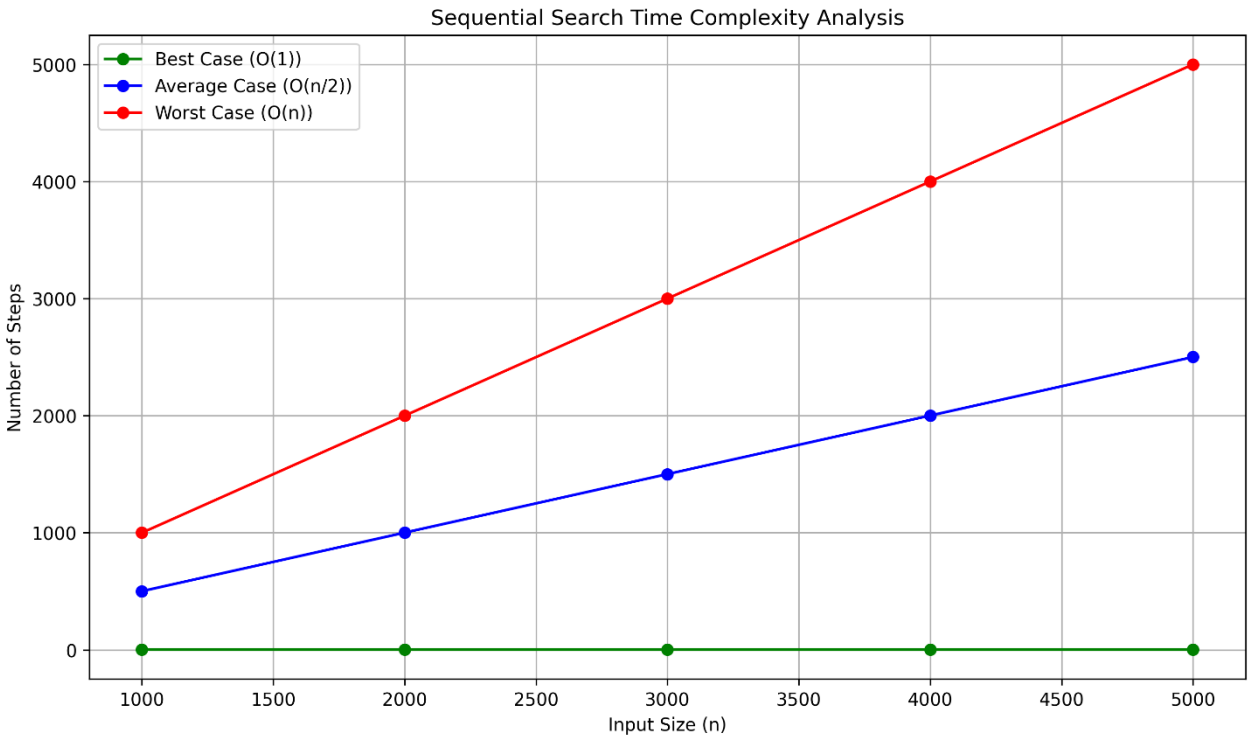
Observations:

- As the input size increases from 1000 to 5000, the number of steps in the **best case** remains constant (O(1)), as the element is found at the first position.
- In the **average case**, steps grow linearly, roughly around *n/2*, since the element is located in the middle.
- In the **worst case**, steps increase proportionally with the input size (*n*), as the element is at the last position.
- This confirms the linear nature of the sequential search algorithm.

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

Inputs	Number of Steps Executed		
	Best Case	**Average Case**	**Worst Case**
1000	1	501	1000
2000	1	1001	2000
3000	1	1501	3000
4000	1	2001	4000
5000	1	2501	5000
Time Complexity→	**O(1)**	**O(n)**	**O(n)**

Chart:



Conclusion:

- The sequential search algorithm exhibits **constant time complexity (O(1))** in the best case and **linear time complexity (O(n))** in both average and worst cases.
- The step count analysis validates theoretical complexities and shows that sequential search is inefficient for large datasets compared to more optimized search algorithms like binary search.

Quiz:

1. **Which is the best case of an algorithm?**

Answer: The best case occurs when the algorithm takes the least time to execute. For sequential search, it's when the key is found at the first position. Time complexity: **O(1)**.

2. Which is the worst case of an algorithm?

Answer: The worst case occurs when the algorithm takes the most time to execute. For sequential search, it's when the key is at the last position or not present. Time complexity: **O(n)**.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms” by E. Horowitz et al.

References used by the students:

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

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	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

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:

Steps to implement binary search are as below:

1. Take an input **sorted** array A of n elements and a target value T.
2. Define variables start and end to represent the start and end indices of the search range, initially set to 0 and n-1 respectively.
3. Repeat the following steps while start <= end:
    a. Calculate the midpoint index mid as (start + end) / 2.
    b. If the value of the midpoint element A[mid] is equal to the target value T, return the value of mid.
    c. If the value of the midpoint element A[mid] is greater than the target value T, set end to mid-1.
    d. If the value of the midpoint element A[mid] is less than the target value T, set start to mid+1.
4. If the target value T is not found in the array, return -1.
5. Output the value returned in Step 3, representing the position of the target value T in the array.

Implement function of binary search algorithm and use linear search function implemented in previous practical. Compare the steps count of both the functions on various inputs ranging from 100 to 500 for each case (Best, Average, and Worst).

Code:

```
import java.io.FileWriter;
import java.io.IOException;

public class Practical_4 {
   static int linearSteps = 0;
   static int binarySteps = 0;

   // Linear Search (Iterative)
   public static int linearSearch(int[] arr, int key) {
      linearSteps = 0;
      for (int i = 0; i < arr.length; i++) {
         linearSteps++;
```

```
        if (arr[i] == key)
            return i;
    }
    return -1;
}

// Binary Search (Recursive)
public static int binarySearch(int arr[], int key, int start, int end) {
    binarySteps++;
    if(start <= end) {
        int mid = (start + end) / 2;

        if(arr[mid] == key) {
            return mid;
        } else if(arr[mid] > key) {
            return binarySearch(arr, key, 0, mid - 1);
        } else {
            return binarySearch(arr, key, mid + 1, end);
        }
    }
    return -1;
}
public static void main(String[] args) {
    int[] sizes = {100, 200, 300, 400, 500};

    try {
        FileWriter bestWriter = new FileWriter("best_case.txt");
        FileWriter avgWriter = new FileWriter("average_case.txt");
        FileWriter worstWriter = new FileWriter("worst_case.txt");

        // Write headers to all files
        bestWriter.write("Input Size,Linear Steps,Binary Steps\n");
        avgWriter.write("Input Size,Linear Steps,Binary Steps\n");
        worstWriter.write("Input Size,Linear Steps,Binary Steps\n");

        // ---------- BEST CASE ----------
        System.out.println("\nNumber of Steps Executed (Best Case)");
        System.out.println("Inputs\tLinear Search\tBinary Search");

        for (int size : sizes) {
            int[] arr = new int[size];
            for (int i = 0; i < size; i++) arr[i] = i + 1;

            int key = arr[0]; // Best case
```

```
            linearSearch(arr, key);
            binarySteps = 0;
            binarySearch(arr, key, 0, size - 1);

            System.out.printf("%-7d%-16d%-15d\n", size, linearSteps, binarySteps);
            bestWriter.write(size + "," + linearSteps + "," + binarySteps + "\n");
        }

        // ---------- AVERAGE CASE ----------
        System.out.println("\nNumber of Steps Executed (Average Case)");
        System.out.println("Inputs\tLinear Search\tBinary Search");

        for (int size : sizes) {
            int[] arr = new int[size];
            for (int i = 0; i < size; i++) arr[i] = i + 1;

            int key = arr[size / 2]; // Middle
            linearSearch(arr, key);
            binarySteps = 0;
            binarySearch(arr, key, 0, size - 1);

            System.out.printf("%-7d%-16d%-15d\n", size, linearSteps, binarySteps);
            avgWriter.write(size + "," + linearSteps + "," + binarySteps + "\n");
        }

        // ---------- WORST CASE ----------
        System.out.println("\nNumber of Steps Executed (Worst Case)");
        System.out.println("Inputs\tLinear Search\tBinary Search");

        for (int size : sizes) {
            int[] arr = new int[size];
            for (int i = 0; i < size; i++) arr[i] = i + 1;

            int key = arr[size - 1]; // Worst case
            linearSearch(arr, key);
            binarySteps = 0;
            binarySearch(arr, key, 0, size - 1);

            System.out.printf("%-7d%-16d%-15d\n", size, linearSteps, binarySteps);
            worstWriter.write(size + "," + linearSteps + "," + binarySteps + "\n");
        }

        // Close files
        bestWriter.close();
```

```
            avgWriter.close();
            worstWriter.close();

                    // System.out.println("\nCSV data written to best_case.txt, average_case.txt, worst_case.txt successfully.");

        } catch (IOException e) {
                    System.out.println("An error occurred while writing files: " + e.getMessage());
        }
    }
}
```

Observations:

- In the best case, both linear and binary search find the element quickly, taking only one or very few steps.
- In the average case, linear search takes about half the input size in steps, increasing linearly. Binary search, however, remains efficient with only a few steps growing logarithmically.
- In the worst case, linear search checks every element, while binary search still performs well with fewer steps. Overall, binary search is much more efficient in average and worst cases on sorted data.

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

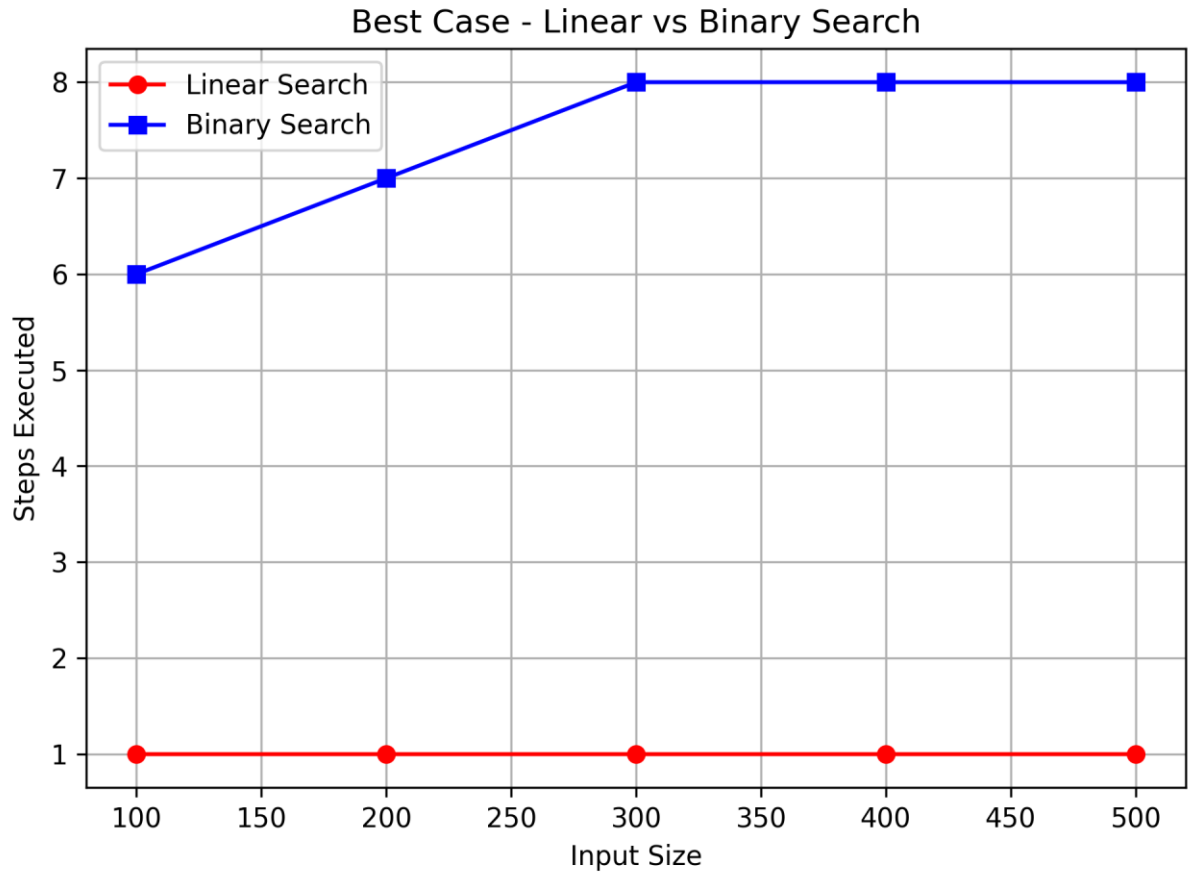
Inputs	**Number of Steps Executed (Best Case)**	
	Linear Search	**Binary Search**
100	1	6
200	1	7
300	1	8
400	1	8
500	1	8
Time Complexity→	**O(1)**	**O(1)**

Inputs	**Number of Steps Executed (Average Case)**	
	Linear Search	**Binary Search**
100	51	14
200	101	13
300	151	21
400	201	20
500	251	28
Time Complexity→	**O(n)**	**O(logn)**

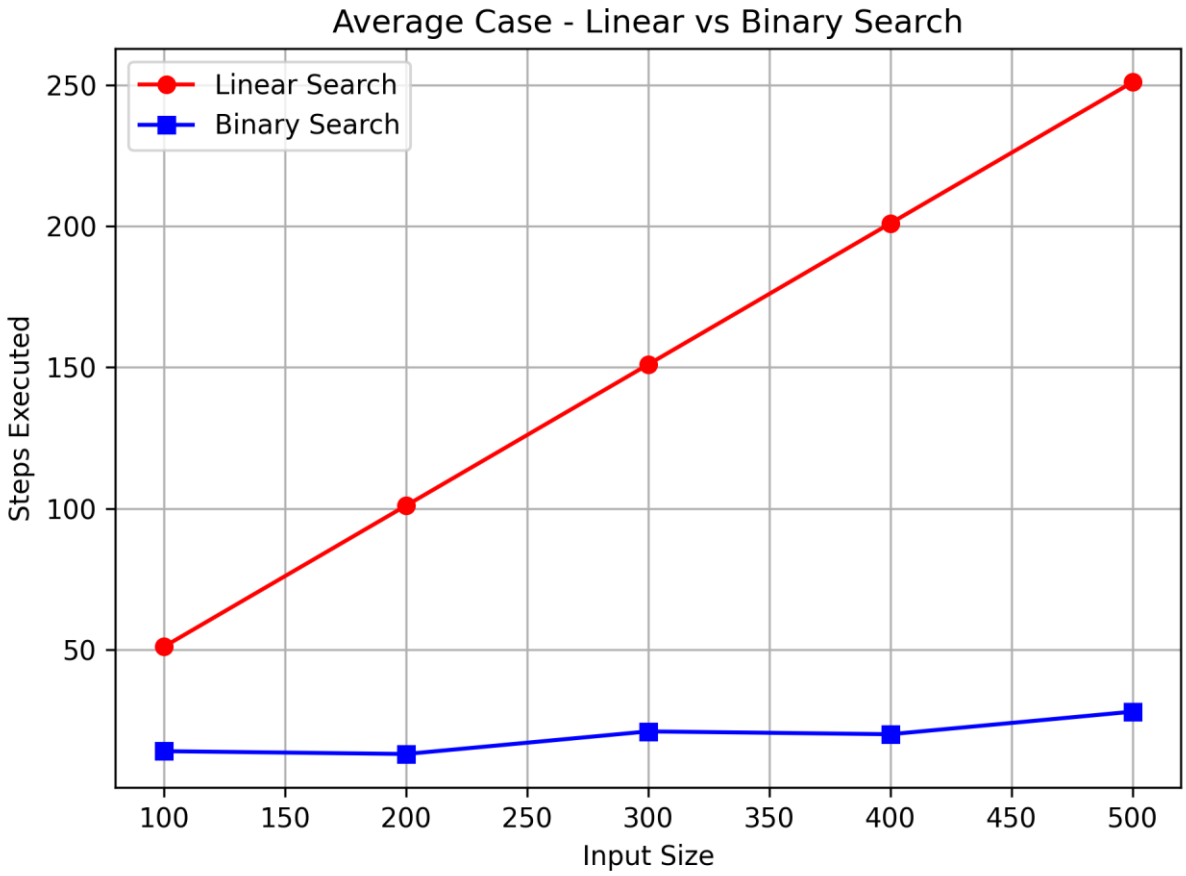
Inputs	Number of Steps Executed (Worst Case)	
	Linear Search	**Binary Search**
100	100	7
200	200	8
300	300	9
400	400	9
500	500	9
Time Complexity→	**O(n)**	**O(logn)**

Chart:

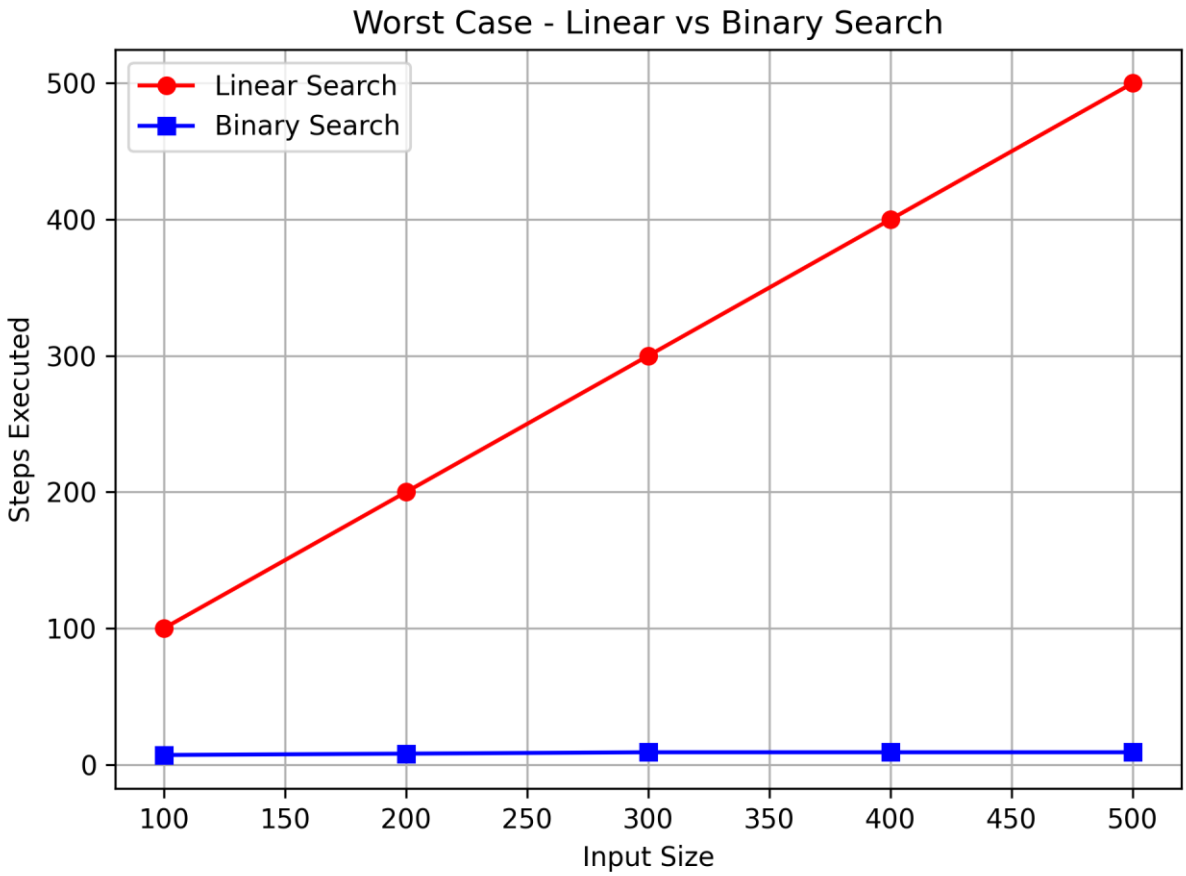
Best Case Analysis: Steps vs Input Size (Linear vs Binary Search)



Average Case Analysis: Steps vs Input Size (Linear vs Binary Search)



Worst Case Analysis: Steps vs Input Size (Linear vs Binary Search)



Conclusion:

- The experiment successfully demonstrates the performance differences between linear and binary search algorithms.
- Linear search shows a linear increase in steps with input size, making it inefficient for large datasets.
- Binary search, due to its divide-and-conquer strategy, performs significantly better in both average and worst cases on sorted data, with logarithmic growth in steps.
- Thus, binary search is more efficient than linear search when working with sorted data.

Quiz:

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

Answer: The best case in binary search occurs when the element being searched is exactly at the middle index of the sorted array. In this case, the algorithm finds the element in just one comparison without any further recursive calls or iterations.

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

Answer: The worst case happens when the element is either not present in the array or lies at one of the extreme ends (first or last element). In such cases, the algorithm must perform the maximum number of divisions before concluding, which is about $\log_2(n)$ comparisons.

3. Which algorithm executes faster in worst case?

Answer: Binary search executes faster in the worst case compared to linear search, but only when the data is sorted. It performs in O(log n) time, whereas linear search takes O(n) time. Therefore, binary search is more efficient for large and sorted datasets.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms” by E. Horowitz et al.

References used by the students:

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

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	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: 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).

Date:

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

Relevant CO: CO1, CO5

Objectives: (a) Compare the performances of two different versions of same problem.
(b) Find the time complexity of algorithms.
(C) Understand the polynomial and non-polynomial problems

Equipment/Instruments: Computer System, Any C language editor

Theory:

The Fibonacci series is the sequence of numbers (also called Fibonacci numbers), where every number is the sum of the preceding two numbers, such that the first two terms are '0' and '1'. In some older versions of the series, the term '0' might be omitted. A Fibonacci series can thus be given as, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . . It can thus be observed that every term can be calculated by adding the two terms before it. We are ignoring initial zero in the series.

To represent any $(n+1)^{th}$ term in this series, we can give the expression as, $F_n = F_{n-1} + F_{n-2}$. We can thus represent a Fibonacci series as shown in the image below,

$$F(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ F(n-1) + F(n-2) & , n > 1 \end{cases}$$

Iterative version to print n^{th} Fibonacci number is as below:

Input: An integer n, where n >= 1.
Output: The n^{th} Fibonacci number.
Steps:
Initialize variables $f_0 = 1$, $f_1 = 1$, and i = 2.
If n is 1 or 2 then
    Print 1
While i < n, do
    a. Set f2 = f0 + f1.
    b. Set f0 = f1.
    c. Set f1 = f2.
    d. Increment i by 1.
Print f1.

Recursive version to print n^{th} Fibonacci number is as below:

Input: An integer n, where n >= 1.
Output: The n^{th} Fibonacci number.
If n is 1 or 2 then
return 1.
else recursively compute next number using the $(n-1)^{th}$ and $(n-2)^{th}$ Fibonacci numbers, and return their sum.
Print the result.

Implement functions of above two versions of Fibonacci series and compare the steps count of both the functions on various inputs ranging from 10 to 50 (if memory permits for recursive version).

Code:

```
import java.io.FileWriter;
import java.io.IOException;

public class Practical_5 {
   static int stepsIter = 0;
   static int stepsRec = 0;

   // Iterative Approach
   public static int fibonacci_iter(int n) {
      stepsIter = 0;
      if (n <= 0) return 0;
      if (n == 1 || n == 2) {
         stepsIter++;
         return 1;
      }

      int f_0 = 0, f_1 = 1, f_2 = 0;
      for (int i = 2; i < n; i++) {
         f_2 = f_0 + f_1;
         f_0 = f_1;
         f_1 = f_2;
         stepsIter++;
      }
      return f_1;
   }

   // Recursive Approach
   public static int fibonacci_rec(int n) {
      stepsRec++;
      if (n == 0) return 0;
      if (n == 1) return 1;
```

```
        return fibonacci_rec(n - 1) + fibonacci_rec(n - 2);
    }

    public static void main(String[] args) {
        String filename = "fibonacci_steps_comparison.txt";

        try (FileWriter writer = new FileWriter(filename)) {
            // Write CSV header
            writer.write("Fibonacci Number,Iterative Steps,Recursive Steps\n");

            System.out.println("Fibonacci Number | Iterative Steps | Recursive Steps");
            System.out.println("-------------------------------------------------");

            for (int n = 10; n <= 40; n += 5) {
                // Iterative
                fibonacci_iter(n);

                // Recursive
                stepsRec = 0;
                fibonacci_rec(n);

                // Console Output
                System.out.printf("%17d | %15d | %15d\n", n, stepsIter, stepsRec);

                // File Output
                writer.write(n + "," + stepsIter + "," + stepsRec + "\n");
            }

            System.out.println("\nData successfully written to: " + filename);

        } catch (IOException e) {
                System.out.println("An error occurred while writing the file: " +
e.getMessage());
        }
    }
}
```

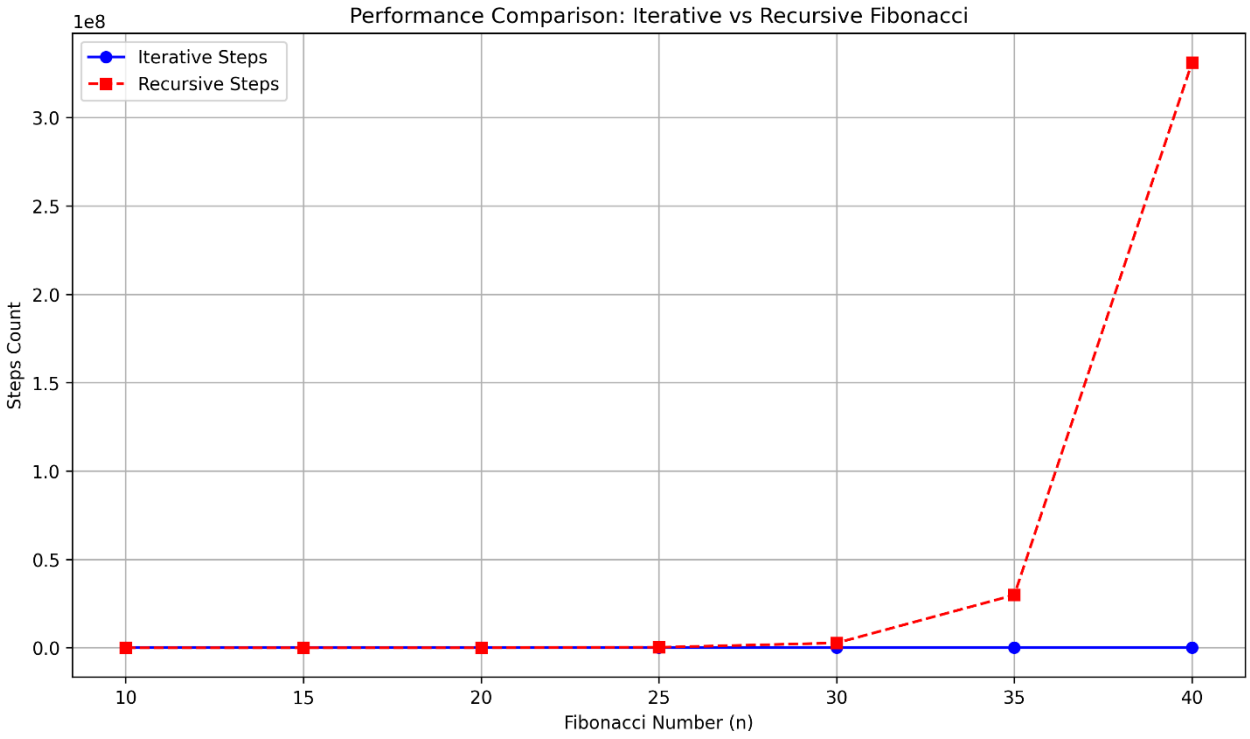
Observations:

- As the input n increases, the **iterative Fibonacci algorithm** grows **linearly**, with the number of steps increasing gradually.
- In contrast, the **recursive algorithm** grows **exponentially**, leading to a massive rise in steps.
- For example, at n = 40, iterative steps are just **38**, while recursive steps exceed **33 crores**, clearly showing how inefficient the recursive method becomes for large inputs due to repeated redundant computations.

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

Inputs	Number of Steps Executed (Random data)	
	Iterative Fibonacci	**Recursive Fibonacci**
10	8	177
15	13	1973
20	18	21891
25	23	242785
30	28	2692537
35	33	29860703
40	38	331160281
45	**(if memory doesn't permit then reduce the range)**	
50		
Time Complexity→	**O(n)**	**O(2^n)**

Chart:



Conclusion:

- The iterative version of the Fibonacci function is significantly more efficient and scalable.
- It runs in **polynomial time O(n)**, while the recursive version runs in **exponential time O(2^n)**.
- Recursive version, although conceptually simpler, suffers from **redundant calculations** due to overlapping subproblems.
- For large values of n, the recursive version consumes **excessive time and memory**, making it unsuitable for practical use without optimization.

Quiz:

1. What is the time complexity of iterative version of Fibonacci function?

Answer: O(n), because it uses a single loop from 2 to n and calculates each Fibonacci number once.

2. What is the time complexity of recursive version of Fibonacci function?

Answer: O(2^n), because it makes two recursive calls for each non-base input, resulting in a binary tree of calls.

3. Can you execute recursive version of Fibonacci function for more inputs?

Answer: Not efficiently without optimization. Beyond n ≈ 40, the number of recursive calls becomes extremely large and may lead to stack overflow or long runtimes. Optimization using memoization or dynamic programming is necessary for higher inputs.

4. What do you mean by polynomial time algorithms and exponential time algorithms?

Answer:

- **Polynomial Time Algorithms** are those whose time complexity can be expressed as a polynomial expression like O(n), O(n^2), etc. These are considered **efficient and scalable**.
- **Exponential Time Algorithms** have time complexities like O(2^n), O(n!), etc., where the time increases **dramatically with input size**. These are considered **inefficient** and often **intractable** for large inputs.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

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

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	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: 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).

Date:

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

Relevant CO: CO1, CO2

Objectives: (a) Improve the performance of quick sort in worst case.
(b) Compare the performance of both the version of quick sort on various inputs

Equipment/Instruments: Computer System, Any C language editor

Theory:

Steps to implement randomized version of quick sort are as below:

```
RANDOMIZED-QUICKSORT(A, low, high)
        if (low< high) {
        pivot= RANDOMIZED_PARTITION(A, low, high);
        RANDOMIZED-QUICKSORT(A, low, pivot);
        RANDOMIZED-QUICKSORT(A, pivot+1,high);
}
RANDOMIZED_PARTITION (A,low,high) {
    pos = Random(low, high)
    pivot = A[pos]
    swap(pivot, a[low])
    left = low
    right = high
    while ( left <right ) {
         /* Move left while item < pivot */
        while( A[left] <= pivot ) left++;
        /* Move right while item > pivot */
        while( A[right] > pivot ) right--;
        if ( left < right )
                swap(A[left],A[right]);}
 /* right is final position for the pivot */
swap(A[right], pivot);
return right; }
```

Implement a function of randomized version of quick sort as per above instructions and use basic version of quick sort (that selects first element as pivot element). Compare the steps count of both the functions on various inputs ranging from 1000 to 5000 for each case (random, ascending, and descending).

Code:

```
import java.util.*;
import java.io.FileWriter;
import java.io.IOException;

public class Practical_6 {
   static long basicSteps = 0;
   static long randomSteps = 0;

   // ------------------- BASIC QUICK SORT -------------------
   public static void basicQuickSort(int[] arr) {
      basicSteps = 0;
      basicQuickSortHelper(arr, 0, arr.length - 1);
   }

   private static void basicQuickSortHelper(int[] arr, int low, int high) {
      if (low < high) {
         basicSteps++;
         int pivotIdx = basicPartition(arr, low, high);
         basicQuickSortHelper(arr, low, pivotIdx - 1);
         basicQuickSortHelper(arr, pivotIdx + 1, high);
      }
   }

   private static int basicPartition(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++;
            swap(arr, i, j);
            basicSteps += 3;
         }
      }
      i++;
      swap(arr, i, high);
      basicSteps += 3;
      return i;
   }

   // ------------------- RANDOMIZED QUICK SORT -------------------
   public static void randomizedQuickSort(int[] arr) {
      randomSteps = 0;
      randomizedQuickSortHelper(arr, 0, arr.length - 1);
```

```
    }

    private static void randomizedQuickSortHelper(int[] arr, int low, int high) {
        if (low < high) {
            randomSteps++;
            int pivotIdx = randomizedPartition(arr, low, high);
            randomizedQuickSortHelper(arr, low, pivotIdx - 1);
            randomizedQuickSortHelper(arr, pivotIdx + 1, high);
        }
    }

    private static int randomizedPartition(int[] arr, int low, int high) {
        Random rand = new Random();
        int randIdx = low + rand.nextInt(high - low + 1);
        swap(arr, randIdx, high);  // Move random pivot to end
        randomSteps++;
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j);
                randomSteps += 3;
            }
        }
        i++;
        swap(arr, i, high);
        randomSteps += 3;
        return i;
    }

    private static void swap(int[] arr, int i, int j) {
        if (i != j) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // ------------------ INPUT GENERATORS ------------------
    public static int[] generateRandomArray(int n) {
        Random rand = new Random();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = rand.nextInt(n);
```

```
        return arr;
    }

    public static int[] generateAscendingArray(int n) {
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = i;
        return arr;
    }

    public static int[] generateDescendingArray(int n) {
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = n - i;
        return arr;
    }

    // ------------------ MAIN DRIVER ------------------
    public static void main(String[] args) throws IOException {
        int[] sizes = {1000, 2000, 3000, 4000, 5000};
        String[] types = {"Random", "Ascending", "Descending"};

        for (String type : types) {
            String fileName = type.toLowerCase() + ".txt";
              System.out.println("\n====================== " + type.toUpperCase() + "
DATA ======================");
               System.out.printf("%-10s %-20s %-20s\n", "Input", "BasicQuickSteps",
"RandomQuickSteps");

            try (FileWriter writer = new FileWriter(fileName)) {
                writer.write("Input,BasicQuickSteps,RandomQuickSteps\n");

                for (int size : sizes) {
                    int[] original = switch (type) {
                        case "Ascending" -> generateAscendingArray(size);
                        case "Descending" -> generateDescendingArray(size);
                        default -> generateRandomArray(size);
                    };

                    int[] arr1 = Arrays.copyOf(original, original.length);
                    int[] arr2 = Arrays.copyOf(original, original.length);

                    basicQuickSort(arr1);
                    randomizedQuickSort(arr2);

                    writer.write(size + "," + basicSteps + "," + randomSteps + "\n");
```

```
                System.out.printf("%-10d %-20d %-20d\n", size, basicSteps,
randomSteps);
            }
        }
    }

        System.out.println("All data saved to random.txt, ascending.txt, and
descending.txt");
  }
}
```

Observations:

- On **random data**, both **Basic** and **Randomized Quick Sort** perform efficiently with ~**O(n log n)** time.
- On **ascending** and **descending** data:
  - **Basic Quick Sort** shows poor performance (**O(n²)**) due to unbalanced partitions.
  - **Randomized Quick Sort** avoids worst-case splits by selecting pivots randomly, maintaining ~**O(n log n)**.

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

Inputs	**Number of Steps Executed (Random data)**	
	Randomized Quick Sort	**Basic Quick Sort**
1000	18663	17099
2000	51408	40951
3000	64930	64653
4000	82434	114999
5000	126469	123583
Time Complexity→	**O(n log n)**	**O(n log n)**

Inputs	**Number of Steps Executed (Ascending data)**	
	Randomized Quick Sort	**Basic Quick Sort**
1000	1502496	19626
2000	6004996	46566
3000	13507496	67954
4000	24009996	104910
5000	37512496	126327
Time Complexity→	**O(n^2)**	**O(n log n)**

Inputs	Number of Steps Executed (Descending data)	
	Randomized Quick Sort	**Basic Quick Sort**
1000	752496	21566
2000	3004996	46350
3000	6757496	70505
4000	12009996	91489
5000	18762496	122747
Time Complexity→	**$O(n^2)$**	**O(n log n)**

Chart:

Chart for Random Data:

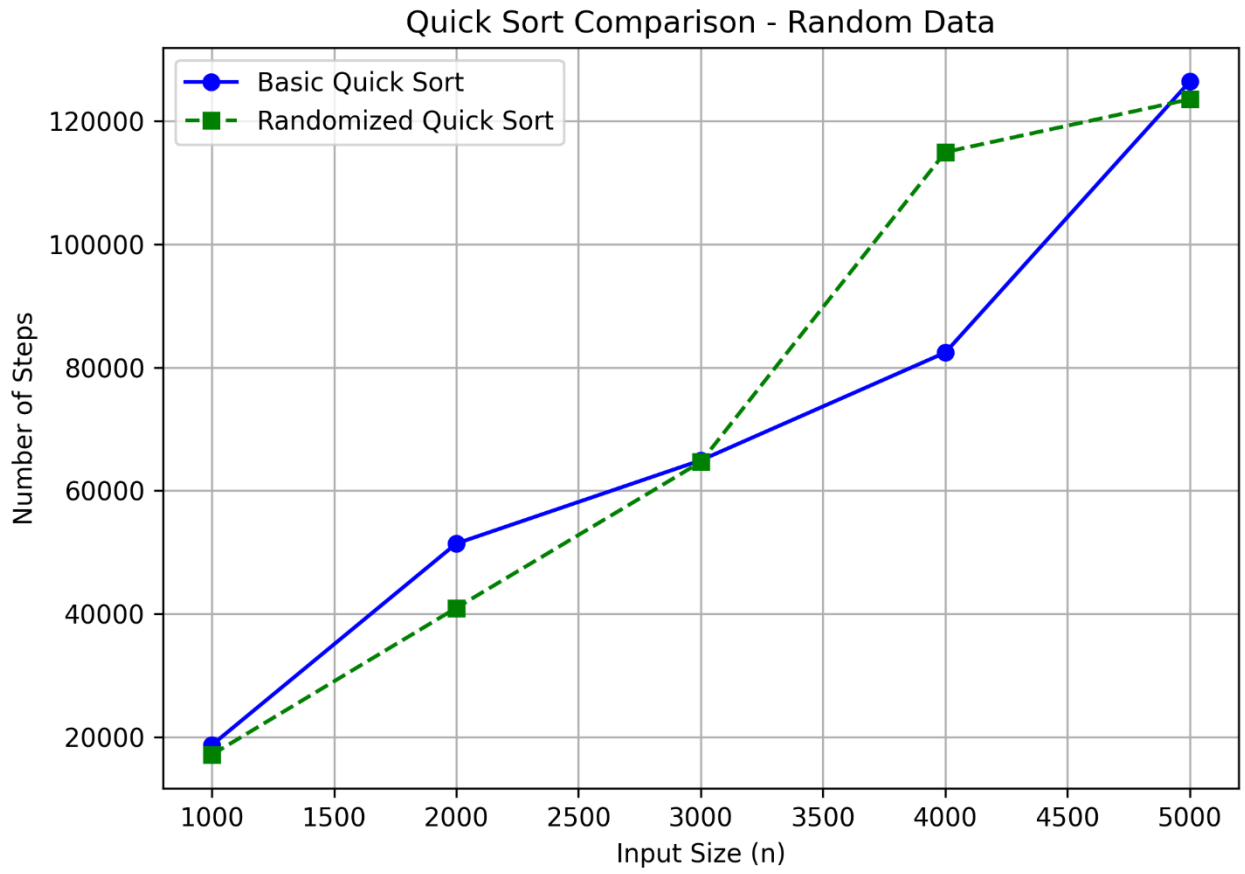


Chart for Ascending Order Data:

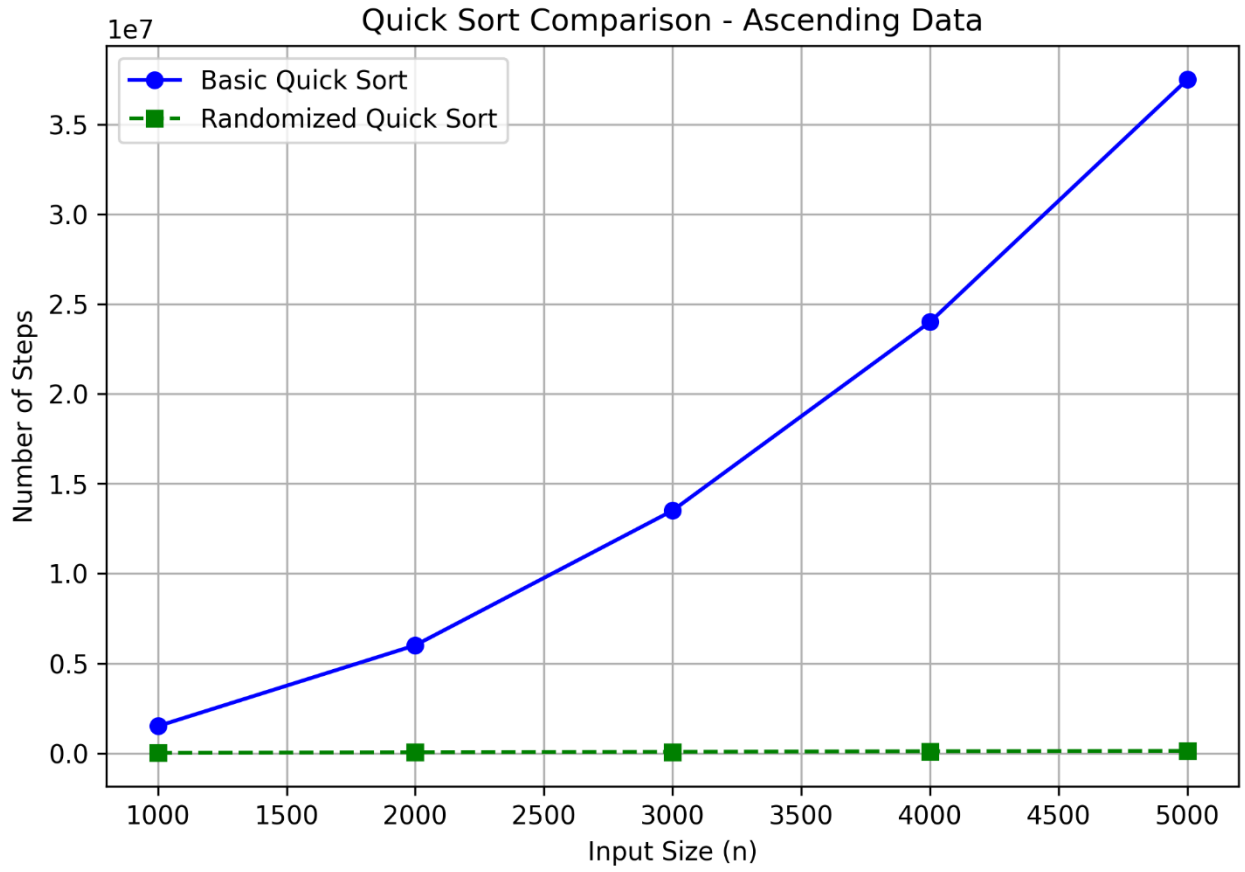
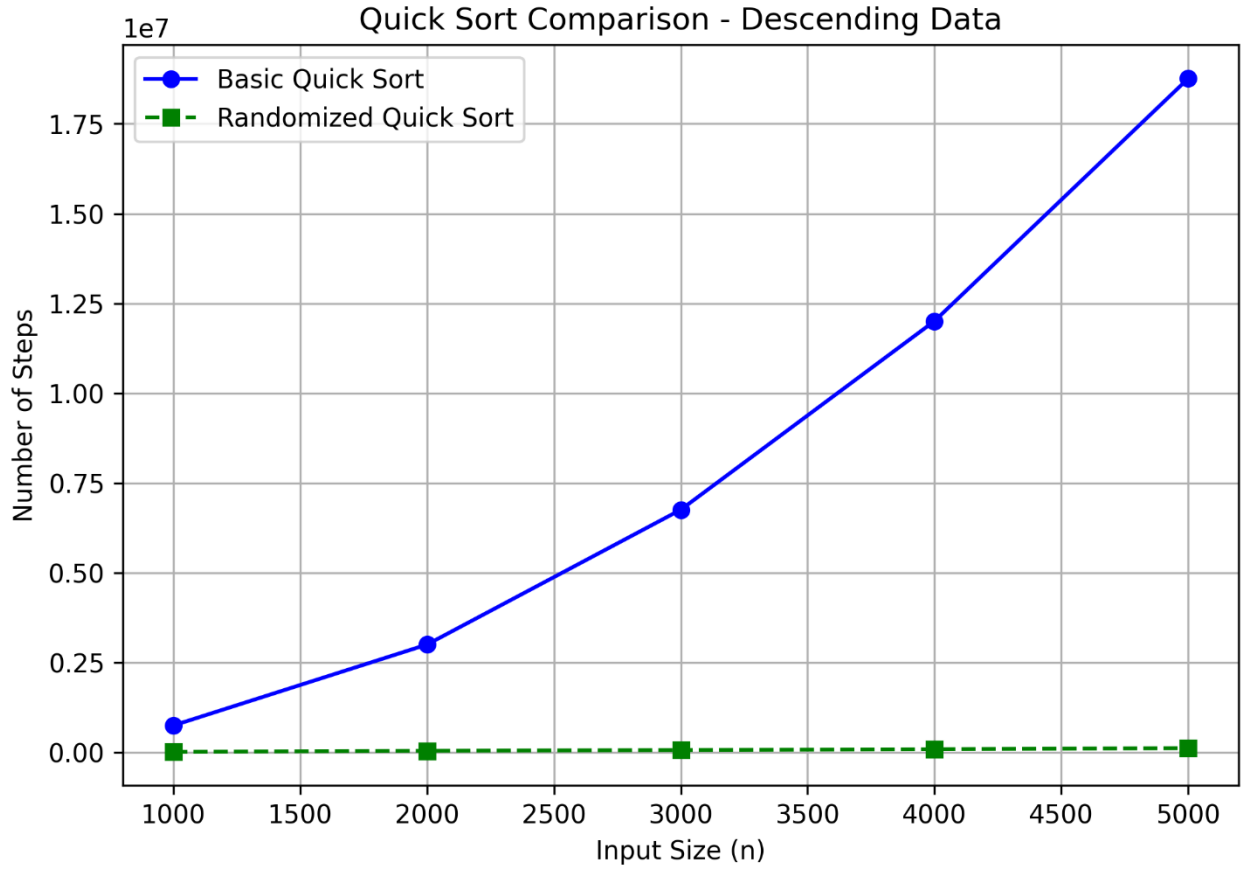


Chart for Descending Order Data:



Conclusion:

- **Randomized Quick Sort** is clearly **more robust** and performs better in cases where **input is already sorted or nearly sorted**, overcoming the worst-case time complexity trap of **Basic Quick Sort**.
- In practical use, **Randomized Quick Sort** is **preferable**, especially when the input distribution is unknown.
- The experiment verifies that **pivot selection strategy has a huge impact** on performance of quick sort.
- Randomization helps in **avoiding the worst-case behavior**, making the algorithm more **consistent and efficient** across different datasets.

Quiz:

1. What is the time complexity of Randomized Quick Sort in worst case?

Answer:

- The worst-case time complexity of Randomized Quick Sort is **$O(n^2)$**.
- This occurs in rare cases when, by chance, the randomly chosen pivots consistently result in unbalanced partitions

2. What is the time complexity of basic version of Quick Sort on sorted data? Give reason of your answer.

Answer:

- The time complexity is **$O(n^2)$**.
- This is because the basic version always picks the last element as the pivot.
- In sorted data (ascending or descending), this leads to the most unbalanced partition where one side has **n-1** elements and the other has 0, causing maximum recursion depth.

3. Can we always ensure O(nlogn) time complexity for Randomized Quick Sort?

Answer:

- No, we **cannot always ensure** O(n log n) time.
- Although the expected (average) time complexity is O(n log n), the pivot selection is random, so there's still a small chance of consistently poor pivot choices that result in $O(n^2)$ time.

4. Which algorithm executes faster on ascending order sorted data?

Answer:

- **Randomized Quick Sort** executes faster.
- Because it selects the pivot randomly, it avoids the worst-case unbalanced partitioning that affects the basic version on sorted data.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

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

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	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: 7

Implement program to solve problem of making a change using dynamic programming.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Making Change problem is to find change for a given amount using a minimum number of coins from a set of denominations. If we are given a set of denominations D = {d0, d1, d2, …,dn} and if we want to change for some amount N, many combinations are possible. Suppose {d1, d2, d5, d8}, {d0, d2, d4}, {d0, d5, d7} all are feasible solutions but the solution which selects the minimum number of coins is considered to be an optimal solution. The aim of making a change is to find a solution with a minimum number of coins / denominations. Clearly, this is an optimization problem.

General assumption is that infinite coins are available for each denomination. We can select any denomination any number of times.

Solution steps are as follow:

Sort all the denominations and start scanning from smallest to largest denomination.In every iterationi, if current denomination *di* is acceptable, then 1 coin is added in solution and total amount is reduced by amount *di*. Hence,
C[i, j] = 1 + (c [i, j – di])

C[i,j] is the minimum number of coins to make change for the amount j. Below figure shows the content of matrix C.

		j											
		0	1	2	3	4	5	6	7	8	9	10	11	12
i	1	0	1	2	3	4	5	1	2	3	4	1	2	2
	2	0	1	2	3	4	5	1	2	3	4	5	6	2
	3	0	1	2	3	4	5	6	7	8	9	10	11	12

Figure: Content of matrix C

using coins if current denomination is larger than current problem size, then we have to skip the denomination and stick with previously calculated solution. Hence,
C[i, j] = C[i – 1, j]

If above cases are not applicable then we have to stick with choice which returns minimum number of coin. Mathematically, we formulate the problem as,

C[i, j] = min {C[i – 1, j] , 1 + C[i, j – di]}

Steps to solve making change problem are as below:

```
Algorithm MAKE_A_CHANGE(d,N)
// d[1…n] = [d1,d2,…,dn] is array of n denominations
// C[1…n, 0…N]  is n x N array to hold the solution of sub problems
// N is the problem size, i.e. amount for which change is required

for i ← 1 to n do
        C[i, 0] ← 0
end
for i ← 1 to n do
        for j ← 1 to N do
                if i = = 1 ans j < d [i]  then
                        C[i, j]  ←  ∞
                else if i == 1 then
                        C[i, j]  ← 1 + C[1, j – d[1])
                else if j < d [i] then
                        C[i, j]  ←  C[I – 1, j]
                else
                        C[i, j]  ← min (C[i – 1, j] ,1 + C[i, j – d[i])
                end
        end
end
return C[n, N]
```

Implement above algorithm and print the matrix C. Your program should return the number of coins required and its denominations.

Observations:

Write observation based on whether this algorithm returns optimal answer or not on various inputs.

Result

Write output of your program

Conclusion:

Quiz:

1. What is the time complexity of above algorithm?

Answer:

2. Does above algorithm always return optimal answer?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.
3. https://codecrucks.com/making-change-problem-using-dynamic-programming/

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	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: 8

Implement program of chain matrix multiplication using dynamic programming.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Given a sequence of matrices A_1, A_2,...,A_n and dimensions p_0, p_1,...,p_n where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.
This algorithm does not perform the multiplications; it just determines the best order in which to perform the multiplications

Two matrices are called compatible only if the number of columns in the first matrix and the number of rows in the second matrix are the same. Matrix multiplication is possible only if they are compatible. Let A and B be two compatible matrices of dimensions p x q and q x r
Suppose dimension of three matrices are :

A_1 = 5 x 4
A_2 = 4 x 6
A_3 = 6 x 2
Matrix multiplication is associative. So

(A_1 A_2) A_3 = {(5 x 4) x (4 x 6) } x (6 x 2)
= (5 x 4 x 6) + (5 x 6 x 2)

= 180

A_1 (A_2 A_3) = (5 x 4) x {(4 x 6) x (6 x 2) }
= (5 x 4 x 2) + (4 x 6 x 2)

= 88

The answer of both multiplication sequences would be the same, but the numbers of multiplications are different. This leads to the question, what order should be selected for a chain of matrices to minimize the number of multiplications?

Let us denote the number of alternative parenthesizations of a sequence of n matrices by p(n). When n = 1, there is only one matrix and therefore only one way to parenthesize the matrix. When

$n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix sub-products, and the split between the two subproducts may occur between the k and $(k + 1)^{st}$ matrices for any k = 1, 2, 3…, n – 1. Thus we obtain the recurrence.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)\, p(n-k) & \text{if } n \geq 2 \end{cases}$$

The solution to the recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n / n^{3/2})$, roughly equal to $\Omega(2^n)$. Thus, the numbers of solutions are exponential in n. A brute force attempt is infeasible to find the solution.

Any parenthesizations of the product $A_i A_{i+1} \ldots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is for some value of k, we first compute the matrices $A_{i....k}$ and $A_{k+1...j}$ and then multiply them together to produce the final product $A_{i...j}$ The cost of computing these parenthesizations is the cost of computing $A_{i....k}$, plus the cost of computing $A_{k+1...j}$ plus the cost of multiplying them together.

We can define m[i, j] recursively as follows. If i == j, the problem is trivial; the chain consists of only one matrix $A_{i...i} = A$. No scalar multiplications are required. Thus m[i, i] = 0 for i = 1, 2 …n. To compute m[i, j] when i < j, we take advantage of the structure of an optimal solution of the first step. Let us assume that the optimal parenthesizations split the product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1}, where $i \leq k < j$. Then m[i, j] is equal to the minimum cost for computing the subproducts $A_{i...k}$ and $A_{k+1...j}$ plus the cost of multiplying these two matrices together.

$$m[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + d_{i-1} \times d_k \times d_j\} & , \text{if } i < j \end{cases}$$

Where $d = \{d_0, d_1, d_2, \ldots, d_n\}$ is the vector of matrix dimensions.
m[i, j] = Least number of multiplications required to multiply matrix sequence $A_i \ldots A_j$.

Steps to solve chain matrix multiplication problem are as below:

```
Algorithm MATRIX_CHAIN_ORDER(p)
// p is sequence of n matrices
n ← length(p) - 1
for i ← 1 to n do
        m[i,i] ← 0
end
for l ← 2 to n do
        for i ← 1 to n – l + 1 do
                j ← i + l - 1
                m[i, j] ← ∞
                for k ← i to j – 1 do
                        q ← m[i, k]+ m[k + 1, j] + di – 1 * dk * dj
                        if q < m[i, j] then
                                m[i,j ] ← q
                                s[i, j] ← k
                        end
```

```
                end
        end
end
return m and s
```

Implement above algorithm and print the matrix *m* ans *c*.

Observations:

Write observation based on whether this algorithm returns optimal number of multiplications or not on various inputs.

Result

Write output of your program

Conclusion:

Quiz:

1. What is the time complexity of above algorithm?

Answer:

2. Does above algorithm always return optimal answer?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “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)			Total (10)
	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: 9

Implement program to solve LCS (Longest Common Subsequence) problem using dynamic programing.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

The Longest Common Subsequence (LCS) problem is a classic computer science problem that involves finding the longest subsequence that is common to two given sequences.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. For example, given the sequence "ABCDE", "ACE" is a subsequence of "ABCDE", but "AEC" is not a subsequence.

Given two sequences X and Y, the LCS problem involves finding the longest common subsequence (LCS) of X and Y. The LCS need not be contiguous in the original sequences, but it must be in the same order. For example, given the sequences "ABCDGH" and "AEDFHR", the LCS is "ADH" with length 3.

Naïve Method:

Let X be a sequence of length m and Y a sequence of length n. Check for every subsequence of X whether it is a subsequence of Y, and return the longest common subsequence found. There are 2^m subsequences of X. Testing sequences whether or not it is a subsequence of Y takes O(n) time. Thus, the naïve algorithm would take $O(n2^m)$ time.

longest common subsequence (LCS) using Dynamic Programming:

Let $X=<x_1,x_2,x_3....,x_m>$ and $Y=<y_1,y_2,y_3....,y_m>$ be the sequences. To compute the length of an element the following algorithm is used.

Step 1 − Construct an empty adjacency table with the size, n × m, where n = size of sequence **X** and m = size of sequence **Y**. The rows in the table represent the elements in sequence X and columns represent the elements in sequence Y.

Step 2 − The zeroth rows and columns must be filled with zeroes. And the remaining values are filled in based on different cases, by maintaining a counter value.

- **Case 1** − If the counter encounters common element in both X and Y sequences, increment the counter by 1.
- **Case 2** − If the counter does not encounter common elements in X and Y sequences at T[i, j], find the maximum value between T[i-1, j] and T[i, j-1] to fill it in T[i, j].

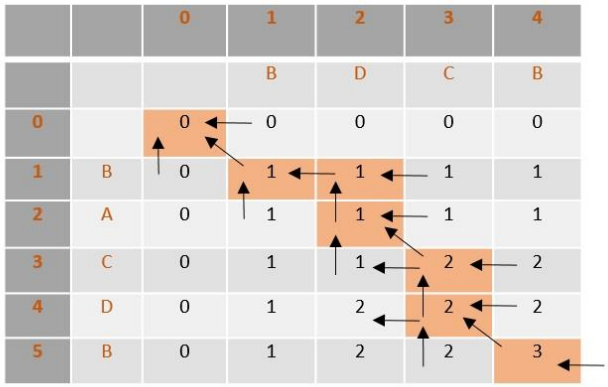
Step 3 − Once the table is filled, backtrack from the last value in the table. Backtracking here is done by tracing the path where the counter incremented first.

Step 4 − The longest common subseqence obtained by noting the elements in the traced path.

Consider the example, we have two strings ***X=BDCB*** and ***Y=BACDB*** to find the longest common subsequence. Following table shows the construction of LCS table.

		0	1	2	3	4
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
5	B	0	1	2	2	3

Once the values are filled, the path is traced back from the last value in the table at T[5, 4].



Algorithm is as below:

Algorithm: LCS-Length-Table-Formulation (X, Y)
m := length(X)
n := length(Y)
for i = 1 to m do
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if xi = yj
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := 'D'
    else
      if C[i -1, j] ≥ C[i, j -1]
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := 'U'
      else
        C[i, j] := C[i, j - 1] + 1

```
        B[i, j] := 'L'
return C and B

Algorithm: Print-LCS (B, X, i, j)
if i=0 and j=0
  return
if B[i, j] = 'D'
  Print-LCS(B, X, i-1, j-1)
  Print(xi)
else if B[i, j] = 'U'
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

Observations:
Write observation based on whether this algorithm returns optimal answer or not on various inputs.

Result
Write output of your program

Conclusion:

Quiz:
1. What is the time complexity of above algorithm?

Answer:

2. Does above algorithm always return optimal answer?

Answer:

3. Does Dynamic programming approach to find LCS perform well compare to naïve approach?

Answer:

Suggested Reference:
1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. "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)			Total (10)
	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: 10

Implement program to solve Knapsack problem using dynamic programming.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Knapsack problem is as stated below:
Given a set of items, each having different weight and value or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.

The knapsack problem is useful in solving resource allocation problem. Let $X = < x_1, x_2, x_3, \ldots, x_n >$ be the set of n items. Sets $W = <w_1, w_2, w_3, \ldots, w_n>$ and $V = < v_1, v_2, v_3, \ldots, v_n>$ are weight and value associated with each item in X. Knapsack capacity is M unit.
The knapsack problem is to find the set of items which maximizes the profit such that collective weight of selected items does not cross the knapsack capacity. Select items from X and fill the knapsack such that it would maximize the profit.
Knapsack problem has two variations. 0/1 knapsack, that does not allow breaking of items. Either add an entire item or reject it. It is also known as a binary knapsack. Fractional knapsack allows breaking of items. Profit will be earned proportionally.
Following are the steps to implement binary knapsack using dynamic programming.

Algorithm DP_BINARY_KNAPSACK (V, W, M)
// Description: Solve binary knapsack problem using dynamic programming
// Input: Set of items X, set of weight W, profit of items V and knapsack capacity M
// Output: Array V, which holds the solution of problem

```
for i ← 1 to n do
        V[i, 0] ← 0
end
for i ← 1 to M do
        V[0, i] ← 0
end
for V[0, i] ← 0 do
        for j ← 0 to M do
                if w[i] ≤ j then
                        V[i, j] ← max{V[i-1, j], v[i] + V[i – 1, j – w[i]]}
                else
```

```
                    V[i, j] ← V[i – 1, j] // w[i]>j
                end
        end
end
```

The above algorithm will just tell us the maximum value we can earn with dynamic programming. It does not speak anything about which items should be selected. We can find the items that give optimum result using the following algorithm.

```
Algorithm TRACE_KNAPSACK(w, v, M)
// w is array of weight of n items
// v is array of value of n items
// M is the knapsack capacity
SW ← { }
SP ← { }
i ← n
j ← M
while ( j> 0 ) do
        if (V[i, j] == V[i – 1, j]) then
                i ← i – 1
        else
                V[i, j] ← V[i, j] – vi
                j ← j – w[i]
                SW ← SW +w[i]
                SP ← SP +v[i]
        end
end
```

Implement the above algorithms for the solution of binary knapsack problem.

Observations:

Write observation based on whether this algorithm returns optimal answer or not on various inputs.

Result

Write output of your program

Conclusion:

Quiz:

1. What is the time complexity of above binary knapsack algorithm?

Answer:

2. Does above algorithm always return optimal answer?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “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)**			**Total (10)**
	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: 11

Implement program for solution of fractional Knapsack problem using greedy design technique.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand greedy algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:
Knapsack problem is as stated below:
Given a set of items, each having different weight and value or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.
Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.
Following are the steps to implement fractional knapsack using greedy design strategy.

1. Compute the value-to-weight ratio for each item in the knapsack.
2. Sort the items in decreasing order of value-to-weight ratio.
3. Initialize the total weight and total value to 0.
4. For each item in the sorted list:
    a. If the entire item can be added to the knapsack without exceeding the weight capacity, add it and update the total weight and total value.
    b. If the item cannot be added entirely, add a fraction of the item that fits into the knapsack and update the total weight and total value accordingly.
    c. If the knapsack is full, stop the algorithm.
5. Return the total value and the set of items in the knapsack.

Implement the program based on above logic for the solution of fractional knapsack problem.

Observations:
Write observation based on whether this algorithm returns optimal answer or not on various inputs.

Result
Write output of your program

Conclusion:

Quiz:

1. What is the time complexity of above knapsack algorithm?

Answer:

2. Does above algorithm always return optimal answer?

Answer:

3. What is the time complexity solving knapsack problem using brute-force method?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “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)			Total (10)
	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: 12

Implement program for solution of Making Change problem using greedy design technique.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand greedy algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Making Change problem is to find change for a given amount using a minimum number of coins from a set of denominations. If we are given a set of denominations D = $\{d_0, d_1, d_2, \ldots, d_n\}$ and if we want to change for some amount N, many combinations are possible. $\{d_1, d_2, d_5, d_8\}$, $\{d_0, d_2, d_4\}$, $\{d_0, d_5, d_7\}$ can be considered as all feasible solutions if sum of their denomination is N. The aim of making a change is to find a solution with a minimum number of coins / denominations. Following are the steps to solve coin change problem using greedy design technique

1. Initialize a list of coin denominations in **descending order**.
2. Initialize a list of coin counts, where each count is initially 0.
3. While the remaining amount is greater than 0:
    a. For each coin denomination in the list:
        i. If the denomination is less than or equal to the remaining amount, add one coin to the count and subtract the denomination from the remaining amount.
        ii. If the denomination is greater than the remaining amount, move on to the next denomination.
4. Return the list of coin counts.

Implement the program based on above steps for the solution of fractional knapsack problem.

Observations:
Write observation based on whether this algorithm returns optimal answer or not on various inputs.

Result
Write output of your program

Conclusion:

Quiz:

1. What is the time complexity of above knapsack algorithm?

Answer:

2. Does above algorithm always return optimal answer?

Answer:

3. What are some variations of the Making Change problem?

Answer:

4. What is the difference between the unbounded coin change problem and the limited coin change problem?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “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)			Total (10)
	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: 13

Implement program for Kruskal's algorithm to find minimum spanning tree.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3, CO6

Objectives: (a) Understand how to use Kruskal's algorithm to find the minimum spanning tree.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

In graph theory, a minimum spanning tree (MST) of an undirected, weighted graph is a tree that connects all the vertices of the graph with the minimum possible total edge weight. In other words, an MST is a subset of the edges of the graph that form a tree and have the smallest sum of weights.

1. Sort all the edges in non-decreasing order of their weight.
2. Initialize an empty set of edges for the minimum spanning tree.
3. For each edge in the sorted order, add the edge to the minimum spanning tree if it does not create a cycle in the tree. To check if adding the edge creates a cycle, you can use the Union-Find algorithm or a similar method to keep track of the connected components of the graph.
4. Continue adding edges until there are V-1 edges in the minimum spanning tree, where V is the number of vertices in the graph.
5. Return the set of edges in the minimum spanning tree.
6. Implement the program based on above steps for the solution of fractional knapsack problem.

Kruskal's Algorithm is as follow:

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1) and (heap not empty)) do
11      {
```

```
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n − 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

Observations:
Write observation based on whether this algorithm always returns minimum spanning tree or not on various inputs.

Result
Write output of your program

Conclusion:

Quiz:
1. What is the time complexity of krushkal's algorithm?

Answer:

2. Does above krushkal's algorithm always return optimal answer?

Answer:

3. What data structure is typically used to keep track of the connected components in Kruskal's algorithm?

Answer:

4. When does Kruskal's algorithm stop adding edges to the minimum spanning tree?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “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)**			**Total (10)**
	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: 14

Implement program for Prim's algorithm to find minimum spanning tree.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3, CO6

Objectives: (a) Understand how to use Prim's algorithm to find the minimum spanning tree.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

In graph theory, a minimum spanning tree (MST) of an undirected, weighted graph is a tree that connects all the vertices of the graph with the minimum possible total edge weight. In other words, an MST is a subset of the edges of the graph that form a tree and have the smallest sum of weights.

Prim's Algorithm is as follow:

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Observations:
Write observation based on whether this algorithm always returns minimum spanning tree or not on various inputs.

Result
Write output of your program

Conclusion:

Quiz:
1. What is the time complexity of Prim's algorithm?

Answer:

2. Does above Prim's algorithm always return optimal answer?

Answer:

3. When does Prim's algorithm stop adding edges to the minimum spanning tree?

Answer:

4. What data structure is typically used to keep track of the vertices in Prim's algorithm?

Answer:

Suggested Reference:
1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. "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)			Total (10)
	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: 15

Implement DFS and BFS graph traversal techniques and write its time complexities.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO6

Objectives: (a) Understand Graph traversal techniques.
(b) Visit all nodes of the graph.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Depth First Search is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It is used to search for a node or a path in a graph, and is implemented recursively or iteratively.
The algorithm starts at a specified node and visits all the nodes in the graph by exploring each branch as far as possible before backtracking to the previous node. When a node is visited, it is marked as visited to prevent loops.

Consider the following steps for the implementation of DFS algorithm:

1. Create an empty stack and push the starting node onto it.
2. Mark the starting node as visited.
3. While the stack is not empty, pop a node from the stack and mark it as visited.
4. For each adjacent node to the current node, if the adjacent node has not been visited, mark it as visited and push it onto the stack.
5. After processing all the adjacent nodes, you can do something with the current node, such as printing it or storing it.
6. Repeat steps 3 to 5 until the stack is empty.

Consider the following steps for the implementation of BFS algorithm:

1. Create a queue Q and a set visited.
2. Add the starting node to the queue Q and mark it as visited.
3. While the queue is not empty:
    a. Dequeue a node from the queue Q and process it.
    b. For each adjacent node of the dequeued node:
        i. If the adjacent node has not been visited, mark it as visited and enqueue it into the queue Q.

Observations:
Write observation based on output of algorithm that which node of graph is traversed first in BFS and DFS.

Result

Write output of your program

Conclusion:

Quiz:

1. What data structure is typically used in the iterative implementation of DFS and BFS?

Answer:

2. What is the time complexity of DFS on a graph with V vertices and E edges?

Answer:

3. What is the time complexity of BFS on a graph with V vertices and E edges?

Answer:

4. In which order are nodes visited in a typical implementation of BFS?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “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)			Total (10)
	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: 16

Implement Rabin-Karp string matching algorithm.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO4

Objectives: (a) Find all occurrences of a pattern in a given text.
(b) Improve the performance of the brute force algorithm.
(c) Find a pattern in a given text with less time complexity in the average case.

Equipment/Instruments: Computer System, Any C language editor

Theory:
It is a string searching algorithm that is named after its authors Richard M. Carp and Michael O. Rabin. This algorithm is used to find all the occurrences of a given pattern 'P'' in a given string 'S' in O(Ns + Np) time in average case, where 'Ns' and 'Np' are the lengths of 'S'' and 'P', respectively.

Let's take an example to make it more clear.
Assume the given string S = "cxyzghxyzvjkxyz" and pattern P = "xyz" and we have to find all the occurrences of 'P' in 'S'.

We can see that "xyz" is occurring in "cxyzghxyzvjkxyz" at three positions. So, we have to print that pattern 'P' is occurring in string 'S' at indices 1, 6, and 12.

Naive Pattern Searching (brute force) algorithm slides the pattern over text one by one and checks for a match. If a match is found, then slide by 1 again to check for subsequent matches. This approach has a time complexity of O(P* (S-P)).

The Rabin-Karp algorithm starts by computing, at each index of the text, the hash value of the string starting at that particular index with the same length as the pattern. If the hash value of that equals to the hash value of the given pattern, then it does a full match at that particular index.

Rabin Karp algorithm first computes the hash value of pattern P and first Np characters from S. If hash values are same, we check the equality of actual strings. If the pattern is found, then it is called hit. Otherwise, it is called a spurious hit. If hash values are not same, no need to compare actual strings.

Steps of Rabin-Karp algorithm are as below:

1. Calculate the hash value of the pattern: The hash value of the pattern is calculated using a hash function, which takes the pattern as input and produces a hash value as output.

2. Calculate the hash values of all the possible substrings of the same length in the text: The hash values of all the possible substrings of the same length as the pattern are calculated using the same hash function.
3. Compare the hash value of the pattern with the hash values of all the possible substrings: If a match is found, the algorithm checks the characters of the pattern and the substring to verify that they are indeed equal.
4. Move on to the next possible substring: If the characters do not match, the algorithm moves on to the next possible substring and repeats the process until all possible substrings have been compared.

Implement the Rabin-Karp algorithm based on above steps and give different input text and pattern to check its correctness. Also, find the time complexity of your implemented algorithm.

Observations:
Write observation based on whether this algorithm able to find a pattern in a given text or not and find it's time complexity in worst case and average case based on its way of working.

Result
Write output of your program

Conclusion:

Quiz:
1. What is the Rabin-Karp algorithm used for?

Answer:

2. What is the time complexity of the Rabin-Karp algorithm in the average case?

Answer:

3. What is the main advantage of the Rabin-Karp algorithm over the brute force algorithm for string matching?

Answer:

4. What is the worst-case time complexity of the Rabin-Karp algorithm?

Answer:

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “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)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										