DRONACHARYA
Group of Institutions

DESIGN AND ANALYSIS OF ALGORITHM LAB

LABORATORY MANUAL

B.Tech. Semester –V

Subject Code: BCS-553

Session: 2024-25, Odd Semester

Name:	
Roll. No.:	
Group/Branch:	

DRONACHARYA GROUP OF INSTITUTIONS
DEPARTMENT OF ECE
#27 KNOWLEDGE PARK 3
GREATER NOIDA

AFFILATED TO Dr. ABDUL KALAM TECHNICAL UNIVERSITY, LUCKNOW

Table of Contents

Vision and Mission of the Institute

Vision:

Instilling core human values and facilitating competence to address global challenges by providing Quality Technical Education.

Mission:

- **M1 -** Enhancing technical expertise through innovative research and education, fostering creativity and excellence in problem-solving.
- **M2 -** Cultivating a culture of ethical innovation and user-focused design, ensuring technological progress enhances the well-being of society.
- **M3 -** Equipping individuals with the technical skills and ethical values to lead and innovate responsibly in an ever-evolving digital landscape.

Vision and Mission of the Department

VISION

To achieve excellence in Electronics and Computer engineering through quality education, research contributing to the emerging technologies and innovation to serve industry and society.

MISSION

- **M1:** To help students achieve their goals by recognizing, identifying, and to bring up their unique strengths through quality education and cutting-edge research training.
- **M2:** To facilitate adequate exposure to the students through training in the state of-the art technologies.
- **M3:** To imbibe ability in the students to solve real life problems as per need of the society through nurturing their skills, creative thinking, and research acumen.

Programme Educational Objectives (PEOs)

PEO 1.

To develop a strong foundation of engineering fundamentals to build successful careers maintaining high ethical standards.

PEO 2.

To prepare graduates for higher studies and research activities, facilitating a commitment to lifelong learning.

PEO 3.

To prepare graduates for higher studies and research activities, facilitating a commitment to lifelong learning.

Programme Outcomes (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

PSO1:

To analyses electronics systems applying principles of mathematics and engineering sciences, to develop innovative ethical solutions to complex engineering problems with team spirit and social commitment.

PSO2:

To develop solution for real world problems based on principles of computer hardware, advanced software and simulation tools with a focus to devise indigenous, eco-friendly and energy efficient projects.

University Syllabus

Design and Analysis of Algorithm Lab (BCS553)		
Course Outcome (CO)		**Bloom's Knowledge Level (KL)**
At the end of course , the student will be able to:		
CO 1	Understand and implement algorithm to solve problems by iterative approach.	K_2, K_4
CO 2	Understand and implement algorithm to solve problems by divide and conquer approach.	K_3, K_5
CO 3	Understand and implement algorithm to solve problems by Greedy algorithm approach.	K_4, K_5
CO 4	Understand and analyze algorithm to solve problems by Dynamic programming, backtracking.	K_4, K_5
CO 5	Understand and analyze the algorithm to solve problems by branch and bound approach.	K_3, K_4

DETAILED SYLLABUS

1. Program for Recursive Binary & Linear Search.
2. Program for Heap Sort.
3. Program for Merge Sort.
4. Program for Selection Sort.
5. Program for Insertion Sort.
6. Program for Quick Sort.
7. Knapsack Problem using Greedy Solution
8. Perform Travelling Salesman Problem
9. Find Minimum Spanning Tree using Kruskal's Algorithm
10. Implement N Queen Problem using Backtracking
11. Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide and- conquer method works along with its time complexity analysis: worst case, average case and best case.
12. Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate how the divide and-conquer method works along with its time complexity analysis: worst case, average case and best case.
13.6. Implement , the 0/1 Knapsack problem using
   (a) Dynamic Programming method
   (b) Greedy method.
14. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.
15. Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.
16. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.
17. Write programs to (a) Implement All-Pairs Shortest Paths problem using Floyd's algorithm.
   (b) Implement Travelling Sales Person problem using Dynamic programming.
18. Design and implement to find a subset of a given set S = {Sl, S2, ... ,Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S ={1, 2, 5, 6, 8} and d= 9, there are two solutions {1,2,6}and {1,8}. Display a suitable message, if the given problem instance doesn't have a solution.
19. Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.

Note: The Instructor may add/delete/modify/tune experiments, wherever he/she feels in a justified manner
It is also suggested that open source tools should be preferred to conduct the lab (C, C++ etc)

Course Outcomes (COs)

CO1	Understand and implement algorithm to solve problems by iterative approach.
CO2	Understand and implement algorithm to solve problems by divide and conquer approach.
CO3	Understand and implement algorithm to solve problems by Greedy algorithm approach
CO4	Understand and analyze algorithm to solve problems by Dynamic programming, backtracking.
CO5	Understand and analyze the algorithm to solve problems by branch and bound approach.

CO-PO Mapping

CO-PO Matrix												
Course Outcomes	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	2	2	3	3	3	-	-	-	-	-	-	2
CO2	3	3	3	2	2	-	-	-	-	-	-	3
CO3	2	3	3	3	3	-	-	-	-	-	-	2
CO4	2	3	2	2	2	-	-	-	-	-	-	2
CO5	2	3	2	2	2	-	-	-	-	-	-	3

CO-PSO Mapping

	PSO1	**PSO2**	**PSO3**
CO1	1	2	1
CO2	1	3	1
CO3	1	3	1
CO4	2	2	1
CO5	2	2	1

Course Overview

This course equips students with the skills to innovate and optimize computational solutions across diverse domains.

- Design and implement algorithms for various computational problems.
- Analyze algorithms to determine their efficiency in terms of time and space.
- Apply appropriate algorithmic paradigms for real-world challenges.
- Understand the limitations of algorithms and explore alternative solutions.

Course Objectives

- **Practical Implementation**: Enable students to implement algorithms using programming languages, reinforcing theoretical concepts learned in lectures.
- **Algorithm Analysis**: Develop skills to analyze the efficiency and performance of algorithms through empirical testing and comparison.
- **Problem-Solving Skills**: Enhance the ability to apply appropriate algorithms to solve complex computational problems effectively.

List of Experiments

Program 1	Write a program for iterartive and recursive Binary Search
Program 2	Write a program for for Selection Sort.
Program 3	Write a program for Insertion Sort.
Program 4	Write a program for Quick sort.
Program 5	Write a program for Merge Sort.
Program 6	Write a program for Heap Sort.
Program 7	a. To implement Fractional knapsack problem using Greedy Technique b. To implement 0/1 Knapsack problem using Dynamic Programming
Program 8	Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm
Program 9	To Implement All Pair Shortest Path Problem using Warshall's and Floyd's Algorithms.
Program 10	To Implement N Queen Problem using Backtracking.

DOs and DON'Ts

DOs

1. Login-on with your username and password.
2. Log off the computer every time when you leave the Lab.
3. Arrange your chair properly when you are leaving the lab.
4. Put your bags in the designated area.
5. Ask permission to print.

DON'Ts

1. Do not share your username and password.
2. Do not remove or disconnect cables or hardware parts.
3. Do not personalize the computer setting.
4. Do not run programs that continue to execute after you log off.
5. Do not download or install any programs, games or music on computer in Lab.
6. Personal Internet use chat room for Instant Messaging (IM) and Sites is strictly prohibited.
7. No Internet gaming activities allowed.
8. Tea, Coffee, Water & Eatables are not allowed in the Computer Lab.

General Safety Precautions

Precautions (In Case of Injury or Electric Shock)

1. **Break Contact Safely**: If a person is in contact with a live electrical source, use an insulator such as a plastic chair or a wooden object to break the contact. Avoid touching the victim with bare hands to prevent electric shock to yourself.
2. **Disconnect Power**: Unplug the affected equipment or turn off the main circuit breaker if accessible. Ensure all systems are powered down to prevent further risk.
3. **Provide Immediate Aid**: If the victim is unconscious, begin CPR immediately. Perform chest compressions and use mouth-to-mouth resuscitation if necessary.
4. **Seek Emergency Help**: Call emergency services and campus security immediately. Time is critical, so act swiftly and efficiently.

Precautions (In Case of Fire)

1. **Power Down Equipment**: Turn off the affected system immediately. If the power switch is not accessible, unplug the device.
2. **Contain the Fire**: If safe to do so, use a fire extinguisher or cover the fire with a heavy cloth to smother it. Ensure the fire does not spread to other devices or components in the lab.
3. **Raise the Alarm**: Activate the nearest fire alarm switch located in the hallway to alert others in the building.
4. **Contact Security and Emergency Services**: Call the security office and emergency services without delay to report the fire and get professional help on site.

Guidelines for Report Preparation for Students in Design and Analysis of Algorithms Lab

All students are required to maintain a comprehensive record of the experiments conducted in the Design and Analysis of Algorithms Lab. The guidelines for preparing this record are as follows:

1. **File Structure**:
    - Each file must begin with a **title page**, followed by an **index page**. Faculty will not sign the file unless there is an entry on the index page.
2. **Student Information**:
    - **Student's Name**, **Roll Number**, and **Date of Conduction** of the experiment must be clearly mentioned on all pages of the record.
3. **Experiment Documentation**:
    - For each algorithm experiment, the record must include the following sections:
        - **Algorithm Name**
        - **Code Implementation**
        - **Analysis of Time and Space Complexity**
        - **Output and Observations**
4. **Additional Notes**:
    - Students must bring their **lab record** to every lab session.
    - Ensure that the lab record is **regularly evaluated** by the faculty. Consistent updates and evaluations are essential for accurate assessment.

Lab Assessment Criteria for Design and Analysis of Algorithms Lab

In a semester, approximately 10 lab classes are conducted for each lab course. These classes are assessed continuously based on five assessment criteria. The performance in each experiment contributes to the computation of Course Outcome (CO) attainment and internal marks. The grading criteria are detailed in the following table:

Grading Criteria	Exemplary (4)	Competent (3)	Needs Improvement (2)	Poor (1)
AC1: Designing Algorithms	The student identifies optimal algorithms and explores innovative approaches to solve problems.	The student identifies appropriate algorithms but lacks a clear goal or strategy for optimization.	The student struggles to define the problem or select suitable algorithms.	The student does not identify or apply relevant algorithms.
AC2: Implementing and Testing	Develops a clear, efficient procedure for coding and testing algorithms. Implementation is correct and adheres to best practices.	Completes the implementation with necessary corrections, but with minor issues in efficiency or adherence to best practices.	Completes the implementation but with significant errors or deviations from expected practices.	Implementation is incomplete or incorrect, with major deviations or errors.
AC3: Analyzing Algorithm Efficiency	Correctly interprets time and space complexities, providing thorough analysis and verification of results.	Provides a reasonable analysis of time and space complexities, but with minor inaccuracies.	Attempts analysis but with several inaccuracies or gaps in understanding.	Lacks understanding or does not attempt to analyze algorithm efficiency.
AC4: Drawing Conclusions	Thoroughly interprets and analyzes the results, proposing viable improvements or optimizations.	Provides conclusions, but they are somewhat incomplete or lack depth.	Attempts to draw conclusions, but they are largely inaccurate or superficial.	Does not provide conclusions or the conclusions are irrelevant.
AC5: Lab Record	Well-organized, clear, and confidently	The record is acceptable but may have minor	The record lacks clarity, organization,	No effort exhibited; the record is poorly

	presented record that correlates theoretical concepts with practical results.	issues with clarity or organization.	and completeness.	presented or incomplete.

Additional Notes:

- **Continuous Assessment**: Each experiment is evaluated during the lab sessions.
- **CO Attainment**: Performance in each experiment contributes to the Course Outcome (CO) attainment.
- **Internal Marks**: The cumulative performance determines the internal marks for the lab course.

Details of Conducted Experiments

The following is a list of the experiments conducted in the Design and Analysis of Algorithms Lab. Each experiment focuses on implementing and analyzing various algorithms fundamental to the field.

Program 1: Iterative and Recursive Binary Search

- **Objective**: Write and implement both iterative and recursive versions of the binary search algorithm. Analyze and compare their performance in terms of time complexity.

Program 2: Selection Sort

- **Objective**: Write a program for the Selection Sort algorithm. Discuss the algorithm's efficiency and its practical applications.

Program 3: Insertion Sort

- **Objective**: Create a program to implement the Insertion Sort algorithm. Evaluate its performance for small data sets and discuss its best and worst-case scenarios.

Program 4: Quick Sort

- **Objective**: Develop a program to implement the Quick Sort algorithm. Analyze its average, best, and worst-case time complexities.

Program 5: Merge Sort

- **Objective**: Implement the Merge Sort algorithm and evaluate its performance. Focus on understanding the divide-and-conquer approach.

Program 6: Heap Sort

- **Objective**: Write a program to implement the Heap Sort algorithm. Analyze the time complexity and understand the use of a binary heap in sorting.

Program 7: Knapsack Problem

- **Objective**:
  - **Part A**: Implement the Fractional Knapsack problem using the Greedy Technique.
  - **Part B**: Implement the 0/1 Knapsack problem using Dynamic Programming. Compare the results of both approaches.

Program 8: Minimum Cost Spanning Tree using Kruskal's Algorithm

- **Objective**: Write a program to find the Minimum Cost Spanning Tree (MST) of a given undirected graph using Kruskal's algorithm. Analyze the algorithm's efficiency.

Program 9: All Pair Shortest Path Problem

- **Objective**: Implement the All-Pair Shortest Path Problem using both Warshall's and Floyd's algorithms. Compare the time complexities of the two methods.

Program 10: N Queen Problem using Backtracking

- **Objective**: Write a program to solve the N Queen problem using backtracking. Analyze the solution's complexity and discuss the efficiency of the backtracking approach.

Lab Experiments

1. Program to implement Recursive and iterative Linear and Binary Search

Linear Search:

- Iterative: Loops through the array to find the element.
- Recursive: Calls itself with the next index until the element is found or the end is reached.

Binary Search (for sorted arrays):

- Iterative: Uses a while loop to narrow down the search range.
- Recursive: Divides the range into two halves, searching recursively in the appropriate half.

Code:

```c
#include <stdio.h>

// Function prototypes
int iterative_linear_search(int arr[], int n, int key);
int recursive_linear_search(int arr[], int n, int key, int index);
int iterative_binary_search(int arr[], int n, int key);
int recursive_binary_search(int arr[], int left, int right, int key);

// Main function
int main() {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter the number to search: ");
    scanf("%d", &key);

    // Iterative Linear Search
    int linear_iter_result = iterative_linear_search(arr, n, key);
    printf("Iterative Linear Search: Element %s found\n",
           (linear_iter_result == -1) ? "not" : "is");

    // Recursive Linear Search
    int linear_recur_result = recursive_linear_search(arr, n, key, 0);
    printf("Recursive Linear Search: Element %s found\n",
           (linear_recur_result == -1) ? "not" : "is");

    // Iterative Binary Search
    int binary_iter_result = iterative_binary_search(arr, n, key);
    printf("Iterative Binary Search: Element %s found\n",
```

```
            (binary_iter_result == -1) ? "not" : "is");

    // Recursive Binary Search
    int binary_recur_result = recursive_binary_search(arr, 0, n - 1, key);
    printf("Recursive Binary Search: Element %s found\n",
            (binary_recur_result == -1) ? "not" : "is");

    return 0;
}

// Iterative Linear Search
int iterative_linear_search(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

// Recursive Linear Search
int recursive_linear_search(int arr[], int n, int key, int index) {
    if (index >= n)
        return -1;
    if (arr[index] == key)
        return index;
    return recursive_linear_search(arr, n, key, index + 1);
}

// Iterative Binary Search
int iterative_binary_search(int arr[], int n, int key) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key)
            return mid;
        if (arr[mid] < key)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}

// Recursive Binary Search
int recursive_binary_search(int arr[], int left, int right, int key) {
    if (left > right)
        return -1;
    int mid = left + (right - left) / 2;
```

```
    if (arr[mid] == key)
        return mid;
    if (arr[mid] < key)
        return recursive_binary_search(arr, mid + 1, right, key);
    return recursive_binary_search(arr, left, mid - 1, key);
}
```

Complexity:

Linear Search:

- **Time Complexity:**
  - **Best Case**: O(1) (The target element is the first element in the array)
  - **Worst Case**: O(n) (The target element is the last element or not present)
  - **Average Case**: O(n) (On average, the element is in the middle of the array)
- **Space Complexity**: O(1) (No additional space is used) for Iterative approach. O(logn) for recursive implementation due to the call stack.

Binary Search:

- **Time Complexity:**
  - **Best Case**: O(1) (The middle element is the target)
  - **Worst Case**: O(logn) (Each iteration halves the search space)
  - **Average Case**: O(logn) (Dividing the search space in half each time)
- **Space Complexity**: O(1) for iterative implementation, O(logn) for recursive implementation due to the call stack.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c:\Users\sharm\Do
Enter the number to search: 5
Iterative Linear Search: Element is found
Recursive Linear Search: Element is found
Iterative Binary Search: Element is found
Recursive Binary Search: Element is found
PS C:\Users\sharm\Downloads\daa_lab>
```

2. Write a Program to implement Selection Sort.

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on sorting order) element from the unsorted portion of the list and moving it to the beginning (or end) of the sorted portion.

Code:

```c
// C program for implementation of selection sort
#include <stdio.h>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {

        // Assume the current position holds
        // the minimum element
        int min_idx = i;

        // Iterate through the unsorted portion
        // to find the actual minimum
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {

                // Update min_idx if a smaller element is found
                min_idx = j;
            }
        }

        // Move minimum element to its
        // correct position
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);
```

```
    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Complexity:

- **Time Complexity:**
  - **Best Case**: $O(n^2)$
    - Even if the array is already sorted, the algorithm still performs n(n−1)/2n(n-1)/2n(n−1)/2 comparisons.
  - **Worst Case**: $O(n^2)$
    - The algorithm always makes n(n−1)/2n(n-1)/2n(n−1)/2 comparisons regardless of the initial order of the elements.
  - **Average Case**: $O(n^2)$
    - The average case also involves n(n−1)/2n(n-1)/2n(n−1)/2 comparisons, making the overall time complexity $O(n^2)$.
- **Space Complexity:**
  - **Space Complexity**: O(1)
    - Selection Sort is an in-place sorting algorithm, which means it does not require additional space proportional to the input size.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c:\Users\sh
Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64
PS C:\Users\sharm\Downloads\daa_lab>
```

3. Write a Program to implement Insertion Sort.

Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array one element at a time. It is much like the way you might sort playing cards in your hands.

Code:

```
// C program for implementation of Insertion Sort
#include <stdio.h>

/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver method
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 1, 10 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Complexity:

- **Time Complexity**
  - **Best case: O(n)**, If the list is already sorted, where n is the number of elements in the list.
  - **Average case: O(n^2)**, If the list is randomly ordered
  - **Worst case: O(n^2)**, If the list is in reverse order
- **Space Complexity**
  - **Auxiliary Space:** O(1), Insertion sort requires **O(1)** additional space, making it a space-efficient sorting algorithm.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c:\U
ile }
1 5 6 10 11 12 13
PS C:\Users\sharm\Downloads\daa_lab>
```

4. Write a Program to implement Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

There are many different versions of QuickSort that pick pivot in different ways.
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in QuickSort is partition. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

Code:

```
// C program to implement Quick Sort Algorithm
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {

    // Initialize pivot to be the first element
    int p = arr[low];
    int i = low;
    int j = high;

    while (i < j) {

        // Find the first element greater than
        // the pivot (from starting)
        while (arr[i] <= p && i <= high - 1) {
            i++;
        }

        // Find the first element smaller than
        // the pivot (from last)
        while (arr[j] > p && j >= low + 1) {
            j--;
        }
        if (i < j) {
            swap(&arr[i], &arr[j]);
        }
```

```
        }
        swap(&arr[low], &arr[j]);
        return j;
    }

    void quickSort(int arr[], int low, int high) {
        if (low < high) {

            // call partition function to find Partition Index
            int pi = partition(arr, low, high);

            // Recursively call quickSort() for left and right
            // half based on Partition Index
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    int main() {

        int arr[] = { 7, 9, 4, 2, 5, 3, 1 };
        int n = sizeof(arr) / sizeof(arr[0]);

        // calling quickSort() to sort the given array
        quickSort(arr, 0, n - 1);

        for (int i = 0; i < n; i++)
            printf("%d ", arr[i]);

        return 0;
    }
```

Complexity:

- **Time Complexity**
  - **Best case: O(nlogn)**
    - Occurs when the pivot divides the array into two nearly equal halves at every step.
    - The recurrence relation for the best case is $T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n)T(n)=2T(n/2)+O(n)$, which resolves to (nlogn).
  - **Average case: O(nlogn)**
    - On average, the pivot divides the array into reasonably balanced parts, leading to logarithmic depth of recursion with linear work per level.
  - **Worst case: $O(n^2)$**
    - Occurs when the pivot is the smallest or largest element, causing unbalanced partitions.
    - The recurrence relation for the worst case is $T(n)=T(n-1)+O(n)T(n) = T(n-1) + O(n)T(n)=T(n-1)+O(n)$, which resolves to $O(n^2)$.

- **Space Complexity:** O(logn) (for the recursive call stack)
  - In the best and average cases, the depth of recursion is O(log⁡n)O(\log n)O(logn) due to balanced partitions.
  - In the worst case, the space complexity can go up to O(n)O(n)O(n) due to highly unbalanced partitions.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c:\U
1 2 3 4 5 7 9
PS C:\Users\sharm\Downloads\daa_lab>
```

5. Write a Program to implement Merge Sort.

Merge Sort is a popular, efficient, and stable sorting algorithm that follows the divide-and-conquer strategy. It divides the input array into two halves, recursively sorts them, and then merges the sorted halves.

Code:

```
// C program for the implementation of merge sort
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int leftArr[n1], rightArr[n2];

    // Copy data to temporary arrays
    for (i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        rightArr[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[left..right]
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        }
        else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of leftArr[], if any
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
```

```
    }

    // Copy the remaining elements of rightArr[], if any
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// The subarray to be sorted is in the index range [left-right]
void mergeSort(int arr[], int left, int right) {
    if (left < right) {

        // Calculate the midpoint
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

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

int main() {
    int arr[] = { 10, 12, 11, 13, 5, 6, 7, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);

      // Sorting arr using mergesort
    mergeSort(arr, 0, n - 1);

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Complexity:

Time Complexity:

1. **Best Case**: O(nlogn)
   - Even if the array is already sorted, Merge Sort will still divide the array and merge it back, performing O(nlogn) operations.
2. **Average Case**: O(nlogn)
   - The array is divided into halves log nnn times, and merging each level requires O(n) operations.
3. **Worst Case**: O(nlogn)

- The time complexity remains O(nlogn) in the worst case because the process of dividing and merging is the same irrespective of the initial order of elements.

Space Complexity:

- **Space Complexity**: O(n)
  - Merge Sort requires additional space proportional to the size of the input array to hold the temporary arrays during the merge process. Each level of recursion uses space for merging subarrays, resulting in a total space complexity of O(n).

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd
4 5 6 7 10 11 12 13
PS C:\Users\sharm\Downloads\daa_lab>
```

6. Write a Program to implement Heap Sort.

Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements. It's known for its efficiency and in-place sorting capability. Here's a breakdown of how it works:

1. Building the Heap:

- The algorithm first transforms the input array into a max-heap. A max-heap is a complete binary tree where the value of each parent node is greater than or equal to the value of its children.
- This is done by starting from the middle of the array and working backwards, "heapifying" each element. Heapifying means ensuring that the subtree rooted at that element satisfies the max-heap property.

2. Sorting:

- Once the max-heap is built, the largest element (which is at the root of the heap) is swapped with the last element in the array.
- The heap size is then reduced by one, and the new root is heapified to maintain the max-heap property.
- This process is repeated until the heap size is 1, at which point the entire array is sorted in ascending order.

Code:

```c
#include <stdio.h>

// Function to swap two integers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to heapify a subtree rooted at index i
void heapify(int arr[], int n, int i) {
    int largest = i;   // Initialize largest as root
    int l = 2 * i + 1; // Left = 2*i + 1
    int r = 2 * i + 2; // Right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest]) {
        largest = l;
    }

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest]) {
        largest = r;
    }

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
```

```
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to perform heap sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: \n");
    printArray(arr, n);

    heapSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Complexity:

Time Complexity:

- O(n log n) in all cases (worst, average, and best). This means the time it takes to sort grows proportionally to n multiplied by the logarithm of n, where n is the number of items being sorted.

Space Complexity:

- O(1). This means Heap Sort uses a constant amount of extra memory, regardless of the size of the input. It sorts the data in place.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c
Unsorted array:
12 11 13 5 6 7
Sorted array:
5 6 7 11 12 13
PS C:\Users\sharm\Downloads\daa_lab>
```

7.

a. To implement Fractional knapsack problem using Greedy Technique

The fractional knapsack problem is a classic optimization problem where you have a knapsack with a maximum weight capacity and a set of items, each with a weight and a value. The goal is to maximize the total value of items you can put in the knapsack, with the key difference from the 0/1 knapsack problem being that you can take fractions of items.

A greedy approach works optimally for the fractional knapsack problem.

Code:

```
#include <stdio.h>
int n = 5;
int p[10] = {3, 3, 2, 5, 1};
int w[10] = {10, 15, 10, 12, 8};
int W = 10;
int main(){
   int cur_w;
   float tot_v;
   int i, maxi;
   int used[10];
   for (i = 0; i < n; ++i)
      used[i] = 0;
   cur_w = W;
   while (cur_w > 0) {
      maxi = -1;
      for (i = 0; i < n; ++i)
         if ((used[i] == 0) &&
               ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
            maxi = i;
      used[maxi] = 1;
      cur_w -= p[maxi];
      tot_v += w[maxi];
      if (cur_w >= 0)
         printf("Added object %d (%d, %d) completely in the bag. Space left:
%d.\n", maxi + 1, w[maxi], p[maxi], cur_w);
      else {
         printf("Added %d%% (%d, %d) of object %d in the bag.\n", (int)((1 +
(float)cur_w/p[maxi]) * 100), w[maxi], p[maxi], maxi + 1);
         tot_v -= w[maxi];
         tot_v += (1 + (float)cur_w/p[maxi]) * w[maxi];
```

```
        }
    }
    printf("Filled the bag with objects worth %.2f.\n", tot_v);
    return 0;
}
```

Complexity:

Time Complexity
The time complexity of the **Fractional Knapsack** problem using the greedy approach can be broken down as follows:

1. **Sorting the Items**: The main computational step involves sorting the items based on their value-to-weight ratio. Sorting n items takes O(nlogn) time.
2. **Iterating through the Items**: After sorting, we iterate through the list of items, which takes O(n) time.

Thus, the overall time complexity is:

O(nlogn)+O(n)= O(nlogn)

Space Complexity
The space complexity is determined by the space needed for:

1. **Storing the Items**: An array of n items is used, each holding a value and weight. This requires O(n) space.
2. **Auxiliary Space for Sorting**: The sorting algorithm (e.g., quicksort, mergesort) requires additional space. The space complexity for sorting is generally O(logn) due to recursive stack space in quicksort or O(n) in mergesort.

Thus, the overall space complexity is:

O(n)+O(logn) or O(n)= O(n)

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c:\Users\sharm\Downlo
Added object 5 (8, 1) completely in the bag. Space left: 9.
Added object 2 (15, 3) completely in the bag. Space left: 6.
Added object 3 (10, 2) completely in the bag. Space left: 4.
Added object 1 (10, 3) completely in the bag. Space left: 1.
Added 19% (12, 5) of object 4 in the bag.
Filled the bag with objects worth 45.40.
PS C:\Users\sharm\Downloads\daa_lab>
```

b. To implement 0/1 Knapsack problem using Dynamic Programming

The 0/1 Knapsack problem is a classic optimization problem where you're given a knapsack with a maximum weight capacity (W) and a set of items, each with a weight (wt) and a value (val). The goal is to select items to put in the knapsack such that the total value is maximized, but you can either take an entire item or not take it at all (hence "0/1"—you can't take fractions of items).

Dynamic programming is a suitable technique to solve the 0/1 Knapsack problem optimally.

Code:

```
#include <stdio.h>
#include <stdlib.h>

// Function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve the 0/1 Knapsack problem using dynamic programming
int knapsack(int W, int wt[], int val[], int n) {
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i -
1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    return K[n][W];
}

int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
```

```
    printf("Maximum value: %d\n", knapsack(W, wt, val, n)); // Output:
Maximum value: 220

    int val2[] = {10,40,30,50};
    int wt2[] = {5,4,6,3};
    int W2 = 10;
    int n2 = sizeof(val2) / sizeof(val2[0]);
    printf("Maximum value: %d\n", knapsack(W2, wt2, val2, n2)); // Output:
Maximum value: 90

    return 0;
}
```

Complexity:

- **Time Complexity**: O(n×W)
- **Space Complexity**:
  - O(n×W) for the standard 2D dynamic programming table.
  - O(W) for the space-optimized 1D array approach.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd
Maximum value: 220
Maximum value: 90
PS C:\Users\sharm\Downloads\daa lab>
```

8. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm

Kruskal's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a weighted, undirected graph. An MST is a subset of the edges that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Code:

```c
// C code to implement Kruskal's algorithm

#include <stdio.h>
#include <stdlib.h>

// Comparator function to use in sorting
int comparator(const void* p1, const void* p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;

    return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

// Function to find the parent of a node
int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;

    return parent[component]
           = findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    // Finding the parents
    u = findParent(parent, u);
    v = findParent(parent, v);
```

```
    if (rank[u] < rank[v]) {
        parent[u] = v;
    }
    else if (rank[u] > rank[v]) {
        parent[v] = u;
    }
    else {
        parent[v] = u;

        // Since the rank increases if
        // the ranks of two sets are same
        rank[u]++;
    }
}

// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
    // First we sort the edge array in ascending order
    // so that we can access minimum distances/cost
    qsort(edge, n, sizeof(edge[0]), comparator);

    int parent[n];
    int rank[n];

    // Function to initialize parent[] and rank[]
    makeSet(parent, rank, n);

    // To store the minimun cost
    int minCost = 0;

    printf(
        "Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++) {
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];

        // If the parents are different that
        // means they are in different sets so
        // union them
        if (v1 != v2) {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0],
                   edge[i][1], wt);
        }
```

```
    }

    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

// Driver code
int main()
{
    int edge[5][3] = { { 0, 1, 10 },
                       { 0, 2, 6 },
                       { 0, 3, 5 },
                       { 1, 3, 15 },
                       { 2, 3, 4 } };

    kruskalAlgo(5, edge);

    return 0;
}
```

Complexity:

Time Complexity: O(E * logE) or O(E * logV)

- Sorting of edges takes O(E * logE) time.
- After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(logV) time.
- So overall complexity is O(E * logE + E * logV) time.
- The value of E can be at most O(V2), so O(logV) and O(logE) are the same. Therefore, the overall time complexity is O(E * logE) or O(E*logV)

Auxiliary Space: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c:\Users\
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

9. To Implement Floyd's warshall algorithm.

Floyd-Warshall is an algorithm for finding the shortest paths between all pairs of vertices in a weighted graph. Unlike Dijkstra's algorithm, which finds shortest paths from a single source, Floyd-Warshall handles all sources simultaneously. It also works with graphs that have negative edge weights (but not negative cycles, which would lead to infinite loops).

Code:

```
// C Program for Floyd Warshall Algorithm
#include <stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
  value. This value will be used
  for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{
    int i, j, k;

    /* Add all vertices one by one to
      the set of intermediate vertices.
      ---> Before start of an iteration, we
      have shortest distances between all
      pairs of vertices such that the shortest
      distances consider only the
      vertices in set {0, 1, 2, .. k-1} as
      intermediate vertices.
      ----> After the end of an iteration,
      vertex no. k is added to the set of
      intermediate vertices and the set
      becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++) {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++) {
                // If vertex k is on the shortest path from
                // i to j, then update the value of
                // dist[i][j]
```

```
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf(
        "The following matrix shows the shortest distances"
        " between every pair of vertices \n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver's code
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { INF, INF, 0, 1 },
                        { INF, INF, INF, 0 } };

    // Function call
    floydWarshall(graph);
    return 0;
}
```

Complexity:

- **Time Complexity:** O(V3), where V is the number of vertices in the graph and we run three nested loops each of size V
- **Auxiliary Space:** O(V2), to create a 2-D matrix in order to store the shortest distance for each pair of nodes.

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd "c:\Users\sharm\Downloads\daa_lab\"
The following matrix shows the shortest distances between every pair of vert
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
PS C:\Users\sharm\Downloads\daa_lab>
```

10.To Implement N Queen Problem using Backtracking.

The N-Queens problem is a classic constraint satisfaction problem in which you need to place N queens on an N×N chessboard so that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal. Backtracking is a very effective way to solve this.

Code:

```
// C program to solve N Queen Problem using backtracking

#define N 4
#include <stdbool.h>
#include <stdio.h>

// A utility function to print solution
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if(board[i][j])
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}

// A utility function to check if a queen can
// be placed on board[row][col]. Note that this
// function is called when "col" queens are
// already placed in columns from 0 to col -1.
// So we need to check only left side for
// attacking queens
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
```

```
    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
            // doesn't lead to a solution, then
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}

// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
// cannot be placed, otherwise, return true and
```

```
// prints placement of queens in the form of 1s.
// Please note that there may be more than one
// solutions, this function prints one  of the
// feasible solutions.
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

// Driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```

Complexity:

Time Complexity: O(N!)
Auxiliary Space: O(N2)

Output:

```
PS C:\Users\sharm\Downloads\daa_lab> cd
. . Q .
Q . . .
. . . Q
. Q . .
PS C:\Users\sharm\Downloads\daa_lab>
```