# LABORATORY MANUAL

## B.Tech. Semester- V

## DESIGN & ANALYSIS OF ALGORITHMS USING C++ LAB
## Subject code: LC-CSE-325G

| | | |
|---|---|---|
| **Prepared by:** | **Checked by:** | **Approved by:** |
| Prof. Vimmi Malhotra | Dr. Ashima Mehta | Name : Prof. (Dr.) Isha Malhotra |
| **Sign.: …………………….** | **Sign.: ………………….** | **Sign.: ………………….** |

**DEPARTMENT OF CSE/CSIT/IT/IOT**
**DRONACHARYA COLLEGE OF ENGINEERING**
**KHENTAWAS, FARRUKH NAGAR, GURUGRAM (HARYANA)**

## Table of Contents

## Vision and Mission of the Institute

### Vision:

"Empowering human values and advanced technical education to navigate and address global challenges with excellence."

### Mission:

- M1: Seamlessly integrate human values with advanced technical education.
- M2: Supporting the cultivation of a new generation of innovators who are not only skilled but also ethically responsible.
- M3: Inspire global citizens who are equipped to create positive and sustainable impact, driving progress towards a more inclusive and harmonious world.

## Vision and Mission of the Department

### Vision:

"Steering the future of computer science through innovative advancements, fostering ethical values and principles through technical education."

### Mission:

**M1:** Directing future innovations in computer science through revolutionary progress.

**M2:** Instilling a foundation of ethical values and principles in every technologist.

**M3:** Offering a comprehensive technical education to equip individuals for a meaningful and influential future.

## Programme Educational Objectives (PEOs)

PEO1: Apply the technical competence in Computer Science and Engineering for solving problems in the real world.

PEO2: Carry out research and develop solutions on problems of social applications.

PEO3: Work in a corporate environment, demonstrating team skills, work morals, flexibility and 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 modeling 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: Exhibit design and programming skills to develop and mechanize business solutions using revolutionary technologies.

PSO2: Learn strong theoretical foundation leading to brilliance and enthusiasm towards research, to provide well-designed solutions to complicated problems.

PSO3: Work effectively with diverse Engineering fields as a team to design, build and develop system applications.

PSO1: Exhibit design and programming skills to develop and mechanize business solutions using revolutionary technologies.

PSO2: Learn strong theoretical foundation leading to brilliance and enthusiasm towards research, to provide well-designed solutions to complicated problems.

PSO3: Work effectively with diverse Engineering fields as a team to design, build and develop system applications.

## University Syllabus

1. Write a Program for iterative and recursive Binary Search.
2. Write a Program for iterative and recursive Linear Search.
3. Write a Program to sort a given set of elements using the Quick Sort.
4. Write a Program to sort a given set of elements using the Merge Sort.
5. Write a Program to sort a given set of elements using the Selection Sort.
6. Write a Program for implementation of Fractional Knapsack problem using Greedy Method and 0/1 Knapsack problem using Dynamic Programming.
7. Write a Program to find the shortest path from a given vertex to other vertices in a weighted connected graph using Dijkstra’s algorithm.
8. Write a Program to find the minimum cost spanning tree (MST) of a given undirected graph using Kruskal’s algorithm/Prim’s Algorithms.
9. Write a Program to implement N-Queens problem using back tracking.
10. Write a Program to check whether a given graph is connected or not using DFS method.
11. Write a program to implement the Travelling Salesman Problem (TSP).

## Course Outcomes (COs)

Upon successful completion of the course, the students will be able to:

C324.1: Help in improving the programming skills of the students.

C325.2: Design of algorithms for any problem.

C325.3: inculcate structured thinking process in the students and

C325.4: Improve the analytical power.

## CO-PO Mapping

| | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C325.1 | 1 | 1 | 3 | 3 | 3 | | 3 | 3 | 3 | 2 | 2 | 2 |
| C325.2 | 1 | 1 | 3 | 3 | 3 | 3 | 2 | 2 | 1 | | 1 | 2 |
| C325.3 | | 2 | 3 | 2 | 1 | 2 | 3 | 3 | | 1 | | 2 |
| C325.4 | | 2 | 3 | 3 | 1 | 1 | 2 | 2 | | 1 | 1 | 2 |
| **C325** | **1** | **1.5** | **3** | **2.75** | **2** | **1.5** | **2.5** | **2.5** | **1** | **1** | **1** | **2** |

## CO-PSO Mapping

| | **PSO1** | **PSO2** | **PSO3** |
|---|---|---|---|
| C325.1 | 1 | 1 | 1 |
| C325.2 | 2 | 1 | 2 |
| C325.3 | 3 | 3 | 3 |
| C325.4 | 1 | 1 | 1 |
| **C325** | **1.75** | **1.5** | **1.75** |

## Course Overview

The design and analysis of algorithms is a field within computer science that focuses on developing efficient and effective algorithms for solving computational problems. It involves the study of various techniques, methodologies, and mathematical tools to design algorithms, analyze their performance, and evaluate their efficiency.

The process of designing an algorithm involves understanding the problem at hand, defining the problem's requirements and constraints, and devising a step-by-step procedure to solve it. This includes selecting appropriate data structures, determining the sequence of operations, and considering algorithmic paradigms such as divide and conquer, greedy algorithms, dynamic programming, and more.

Once an algorithm is designed, the next step is to analyze its performance. This involves evaluating factors like time complexity (how the algorithm's running time increases with input size), space complexity (how much memory the algorithm requires), and other relevant metrics. By analyzing an algorithm's performance, we can determine its efficiency and scalability, enabling us to make informed decisions about which algorithm to use for a given problem.

The goal of the design and analysis of algorithms is to develop algorithms that are correct, efficient, and scalable. Correctness ensures that the algorithm produces the correct output for all possible inputs. Efficiency focuses on minimizing the use of computational resources such as time and memory. Scalability considers how well the algorithm performs as the input size becomes larger.

By studying the design and analysis of algorithms, computer scientists can develop solutions that are both practical and optimized, leading to improved performance and resource utilization in a wide range of applications, including data processing, optimization problems, artificial intelligence, and much more.

## List of Experiments mapped with COs

## 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. To break the victim with live electric source, use an insulator such as fire wood or plastic to break the contact. Do not touch the victim with bare hands to avoid the risk of electrifying yourself.
2. Unplug the risk of faulty equipment. If main circuit breaker is accessible, turn the circuit off.
3. If the victim is unconscious, start resuscitation immediately, use your hands to press the chest in and out to continue breathing function. Use mouth-to-mouth resuscitation if necessary.
4. Immediately call medical emergency and security. Remember! Time is critical; be best.

### Precautions (In case of Fire)

1. Turn the equipment off. If power switch is not immediately accessible, take plug off.
2. If fire continues, try to curb the fire, if possible, by using the fire extinguisher or by covering it with a heavy cloth if possible isolate the burning equipment from the other surrounding equipment.
3. Sound the fire alarm by activating the nearest alarm switch located in the hallway.
4. Call security and emergency department immediately:

| | | |
|---|---|---|
| **Emergency** | **:** | **Reception** |
| **(Reception) Security** | **:** | **Front Gate** |

## Guidelines to students for report preparation

All students are required to maintain a record of the experiments conducted by them. Guidelines for its preparation are as follows: -

1) All files must contain a title page followed by an index page. ***The files will not be signed by the faculty without an entry in the index page.***

2) Student's Name, Roll number and date of conduction of experiment must be written on all pages.

3) For each experiment, the record must contain the following

(i) Aim/Objective of the experiment

(ii) Pre-experiment work (as given by the faculty)

(iii) Lab assignment questions and their solutions

(iv) Test Cases (if applicable to the course)

(v) Results/ output

**Note:**

1. Students must bring their lab record along with them whenever they come for the lab.

2. Students must ensure that their lab record is regularly evaluated.

## Lab Assessment Criteria

An estimated 10 lab classes are conducted in a semester for each lab course. These lab classes are assessed continuously. Each lab experiment is evaluated based on 5 assessment criteria as shown in following table. Assessed performance in each experiment is used to compute CO attainment as well as internal marks in the lab course.

| **Grading Criteria** | **Exemplary (4)** | **Competent (3)** | **Needs Improvement (2)** | **Poor (1)** |
|---|---|---|---|---|
| **AC1: Pre-Lab written work (this may be assessed through viva)** | Complete procedure with underlined concept is properly written | Underlined concept is written but procedure is incomplete | Not able to write concept and procedure | Underlined concept is not clearly Understood |
| **AC2: Program Writing/ Modeling** | Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied, Program/solution written is readable | Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied | Assigned problem is properly analyzed & correct solution designed | Assigned problem is properly analyzed |
| **AC3: Identification & Removal of errors/ bugs** | Able to identify errors/ bugs and remove them | Able to identify errors/ bugs and remove them with little bit of guidance | Is dependent totally on someone for identification of errors/ bugs and their removal | Unable to understand the reason for errors/ bugs even after they are explicitly pointed out |
| **AC4: Execution & Demonstration** | All variants of input /output are tested, Solution is well demonstrated and implemented concept is clearly explained | All variants of input /output are not tested, However, solution is well demonstrated and implemented concept is clearly explained | Only few variants of input /output are tested, Solution is well demonstrated but implemented concept is not clearly explained | Solution is not well demonstrated and implemented concept is not clearly explained |
| **AC5: Lab Record Assessment** | All assigned problems are well recorded with objective, design constructs and solution along with Performance analysis using all variants of input and output | More than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output | Less than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output | Less than 40 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output |

# LAB EXPERIMENTS

# LAB EXPERIMENT 1

**OBJECTIVE:**

Write a Program for iterative and recursive Binary Search.

**BRIEF DESCRIPTION:**

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O (log N).

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

**<u>Binary Search Algorithm:</u>**

In this algorithm,

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
  - If the key is smaller than the middle element, then the left side is used for next search.
  - If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

The Binary Search Algorithm can be implemented in the following two ways:

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

**PRE-EXPERIMENT QUESTIONS:**

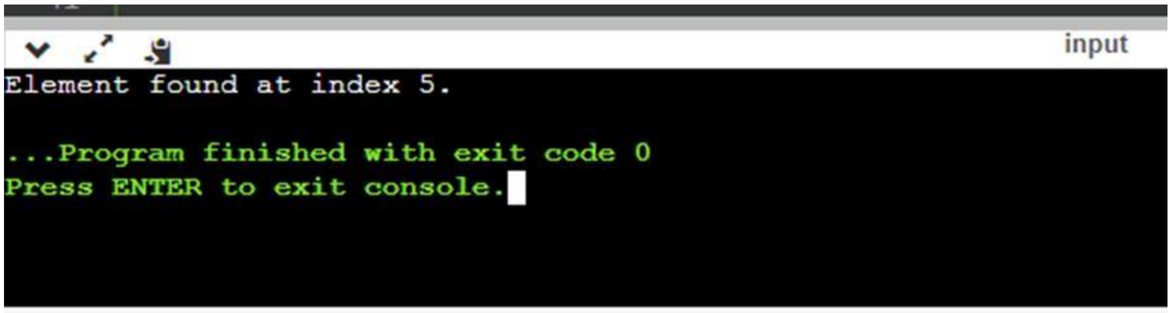
- What is binary search?
- How does the binary search algorithm work, and what are its main steps?
- Discuss the time complexity of the binary search algorithm. Is it efficient compared to other search algorithms?

**Program Code: (Iterative Binary Search)**

```
#include <iostream>
using namespace std;

int binarySearchIterative(int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }

    return -1;
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 23;

    int result = binarySearchIterative(arr, 0, n - 1, target);
    if (result == -1)
        cout << "Element not found in the array.";
    else
        cout << "Element found at index " << result << ".";

    return 0;
}
```

**Output:**

Students will be able to recognize actual position of element in iterative way as shown below:



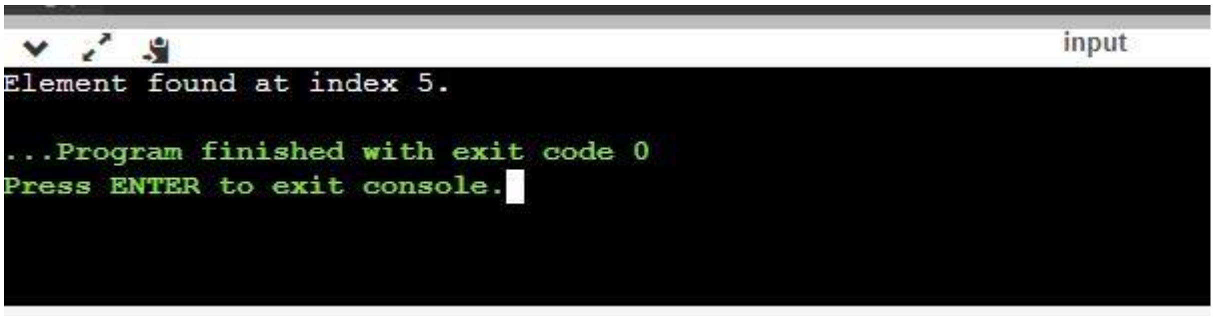
```
Element found at index 5.

...Program finished with exit code 0
Press ENTER to exit console.
```

## Program Code: (Recursive Binary Search)

```
#include <iostream>
using namespace std;

int binarySearchRecursive(int arr[], int left, int right, int target) {
    if (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            return binarySearchRecursive(arr, mid + 1, right, target);

        return binarySearchRecursive(arr, left, mid - 1, target);
    }

    return -1;
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 23;

    int result = binarySearchRecursive(arr, 0, n - 1, target);
    if (result == -1)
        cout << "Element not found in the array.";
    else
        cout << "Element found at index " << result << ".";

    return 0;
}
```

## Output:

Students will be able to recognize actual position of element in recursive way as shown below:



```
Element found at index 5.

...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- What is the basic principle behind binary search, and how does it reduce the search space?
- Explain the time complexity of the binary search algorithm and compare it with other search algorithms.
- What are the necessary conditions for binary search to be applicable, and what happens if these conditions are not met?

# LAB EXPERIMENT 2

**OBJECTIVE:**

Write a Program for iterative and recursive Linear Search.

**BRIEF DESCRIPTION:**

Linear search, also known as sequential search, is a simple and straightforward searching algorithm used to find an element within a collection of items. It is applicable to both ordered and unordered lists.

The linear search algorithm starts at the beginning of the list and compares each element sequentially until a match is found or the end of the list is reached. If the target element is found, the search terminates, and the index or position of the element is returned. If the end of the list is reached without finding a match, the search concludes, and a special value (e.g., -1) is typically returned to indicate that the element was not found.

Linear search is intuitive and easy to implement. It works well for small lists or when the target element is located near the beginning of the list. However, its performance is linear with respect to the size of the list, resulting in a worst-case time complexity of O(n), where n is the number of elements in the list. This means that as the size of the list grows, the time required for the search increases linearly.

Linear search is often used when:

- The list is unsorted or only partially sorted.
- The list is small.
- The cost of sorting the list in advance is not justified.
- The search is infrequent or not time-critical.

**PRE-EXPERIMENT QUESTIONS:**

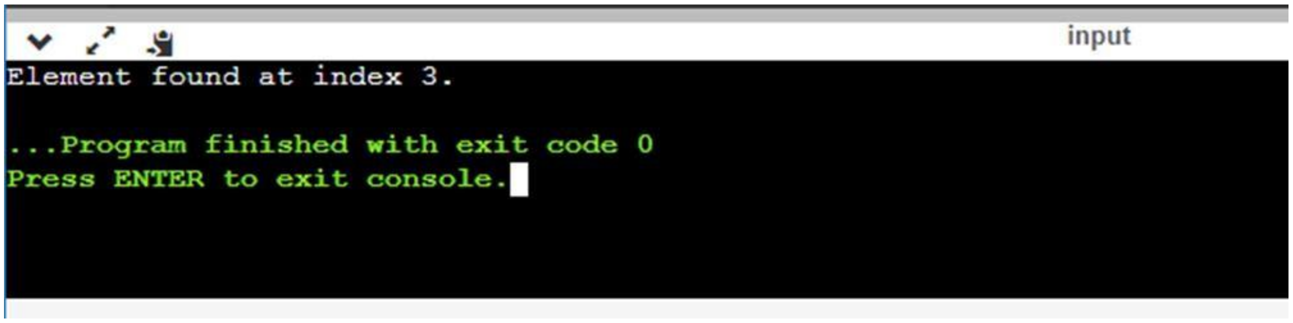
- What is linear search, and how does it work?
- What is the time complexity of linear search, and when is it suitable to use?
- What are the advantages and disadvantages of linear search?

## Program Code:(Iterative Linear Search)

```
#include <iostream>
using namespace std;

int linearSearchIterative(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target)
            return i;
    }

    return -1;
}

int main() {
    int arr[] = {5, 12, 8, 2, 16, 23, 38, 56, 72, 91};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 23;

    int result = linearSearchIterative(arr, size, target);
    if (result == -1)
        cout << "Element not found in the array.";
    else
        cout << "Element found at index " << result << ".";

    return 0;
}
```

## Output:

Students will be able to recognize actual position of element in iterative way in unsorted way as shown below:



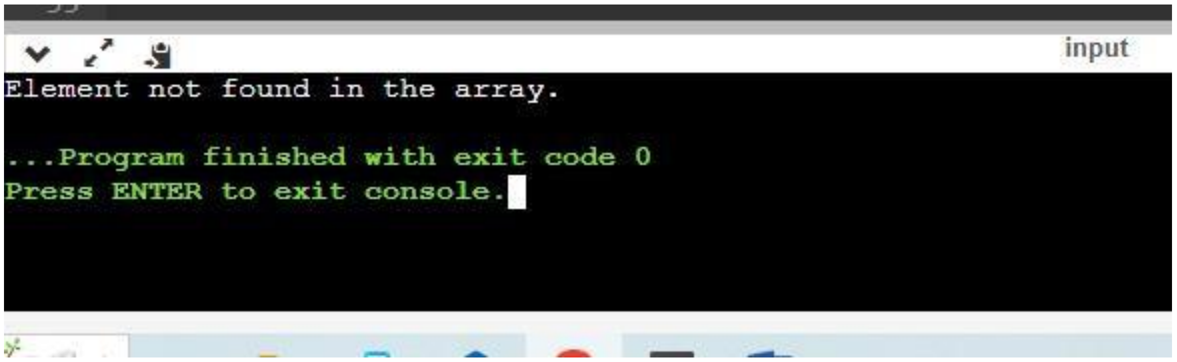
```
Element found at index 3.

...Program finished with exit code 0
Press ENTER to exit console.
```

## Program Code:(Recursive Linear Search)

```
#include <iostream>
using namespace std;

int linearSearchRecursive(int arr[], int left, int right, int target) {
    if (left > right)
        return -1;

    if (arr[left] == target)
        return left;

    return linearSearchRecursive(arr, left + 1, right, target);
}

int main() {
    int arr[] = {5, 12, 8, 2, 16, 23, 38, 56, 72, 91};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 23;

    int result = linearSearchRecursive(arr, 0, size - 1, target);
    if (result == -1)
        cout << "Element not found in the array.";
    else
        cout << "Element found at index " << result << ".";

    return 0;
}
```

## Output:

Students will be able to recognize actual position of element in recursive way in unsorted way as shown below:



```
Element not found in the array.

...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- What is the time complexity of linear search, and under what circumstances is it efficient to use?
- What is the best-case and worst-case time complexity of linear search?
- Are there any modifications or optimizations that can be applied to improve the efficiency of linear search?

## LAB EXPERIMENT 3

**OBJECTIVE:**

Write a Program to sort a given set of elements using the Quick Sort.

**BRIEF DESCRIPTION:**

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

There are many different versions of Quick Sort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in Quick Sort 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.

In this program, the partition function is responsible for rearranging the elements around a pivot element, such that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right. It returns the index of the pivot element.

The quick_sort function recursively partitions the array using the pivot element's index. It then applies the quick_sort function to the sub-arrays formed by the partitioning.

Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**PRE-EXPERIMENT QUESTIONS:**

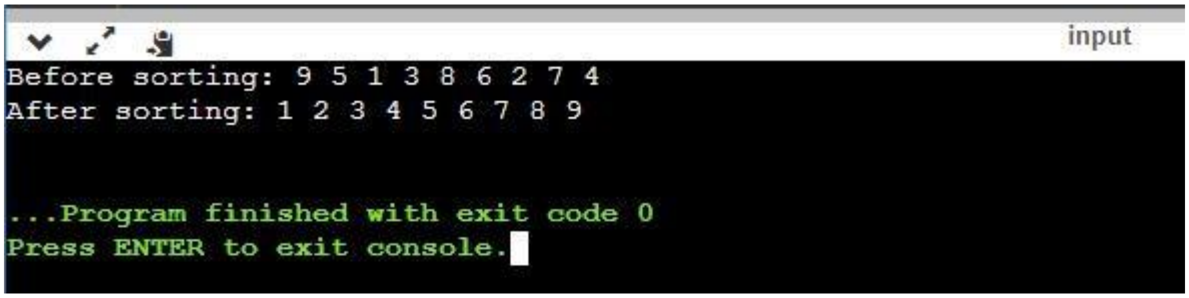
- Explain the concept of divide and conquer.
- Define in place sorting algorithm.
- List different ways of selecting pivot element.

**Program Code:**

```
#include <iostream>
#include <vector>

using namespace std;

// Partition the array and return the index of the pivot element
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

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

// Quick Sort implementation
void quick_sort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivot_index = partition(arr, low, high);
        quick_sort(arr, low, pivot_index - 1);
        quick_sort(arr, pivot_index + 1, high);
    }
}

int main() {
    vector<int> elements = {9, 5, 1, 3, 8, 6, 2, 7, 4};

    cout << "Before sorting: ";
    for (int num : elements) {
        cout << num << " ";
    }
    cout << endl;

    quick_sort(elements, 0, elements.size() - 1);

    cout << "After sorting: ";
    for (int num : elements) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

**Output:**

Students will be able to sort the given array with the help of Pivot element as shown below:



```
Before sorting: 9 5 1 3 8 6 2 7 4
After sorting: 1 2 3 4 5 6 7 8 9


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- What is the average case time complexity of quick sort?
- Describe a technique or modification to the Quick Sort algorithm that ensures better performance even in the worst-case scenario.
- How does the choice of pivot element affect the performance of the Quick Sort algorithm?

## LAB EXPERIMENT 4

**OBJECTIVE:**

Write a Program to sort a given set of elements using the Merge Sort.

**BRIEF DESCRIPTION:**

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

The merge () function is used for merging two halves. The merge (a, low, mid, high) is key process that assumes that a [low, mid] and a [mid+1, high] are sorted and merges the two sorted sub-arrays into one.

In this program, the merge function merges two sorted subarrays (left and right) into a single sorted array. It uses three indices (i, j, and k) to traverse the subarrays and the merged array.

The merge_sort function recursively divides the array into two halves and applies the merge_sort function to each half. It then merges the sorted halves using the merge function.

In the main function, we define the elements vector with the input set of elements. The vector is printed before and after sorting using the Merge Sort algorithm.

Implement merge sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

While comparing two sub lists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sub lists are empty and the new combined sub list comprises all the elements of both the sub lists.

**PRE-EXPERIMENT QUESTIONS:**

- What is the running time of merge sort?
- What technique is used to sort elements in merge sort?
- Is merge sort in place sorting algorithm?

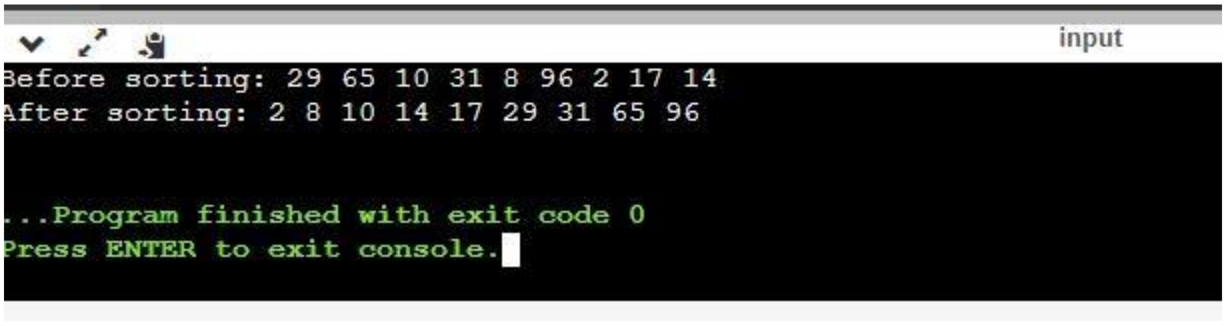
**Program Code:**

```
#include <iostream>
#include <vector>

using namespace std;

// Merge two sorted subarrays into a single sorted array
void merge(vector<int>& arr, int low, int mid, int high) {
    int left_size = mid - low + 1;
    int right_size = high - mid;

    vector<int> left(left_size);
    vector<int> right(right_size);

    // Copy elements to temporary arrays
    for (int i = 0; i < left_size; i++)
        left[i] = arr[low + i];
    for (int j = 0; j < right_size; j++)
        right[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr
    int i = 0; // Index of the left subarray
    int j = 0; // Index of the right subarray
    int k = low; // Index of the merged array

    while (i < left_size && j < right_size) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of left[] and right[], if any
    while (i < left_size) {
        arr[k] = left[i];
        i++;
        k++;
    }

    while (j < right_size) {
        arr[k] = right[j];
        j++;
        k++;
    }
}
```

```
// Merge Sort implementation
void merge_sort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;

        // Divide the array into two halves
        merge_sort(arr, low, mid);
        merge_sort(arr, mid + 1, high);

        // Merge the sorted halves
        merge(arr, low, mid, high);
    }
}

int main() {
    vector<int> elements = {9, 5, 1, 3, 8, 6, 2, 7, 4};

    cout << "Before sorting: ";
    for (int num : elements) {
        cout << num << " ";
    }
    cout << endl;

    merge_sort(elements, 0, elements.size() - 1);

    cout << "After sorting: ";
    for (int num : elements) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

**<u>Output:</u>**

Students will be able to identify sorting with divide and conquer technique as shown below:



```
Before sorting: 29 65 10 31 8 96 2 17 14
After sorting: 2 8 10 14 17 29 31 65 96


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- What makes merge sort a stable sorting algorithm?
- How does Merge Sort manage its memory usage during the sorting process?
- Discuss the advantages and limitations of Merge Sort compared to other sorting algorithms.

## LAB EXPERIMENT 5

**OBJECTIVE:**

Write a Program to sort a given set of elements using the Selection Sort.

**BRIEF DESCRIPTION:**

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

Advantages of Selection Sort Algorithm:

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm:

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

This program defines a function selectionSort to perform the Selection Sort algorithm on the given array. It iterates through the array and selects the minimum element in each iteration and swaps it with the first unsorted element. The program also includes a helper function printArray to print the elements of an array.

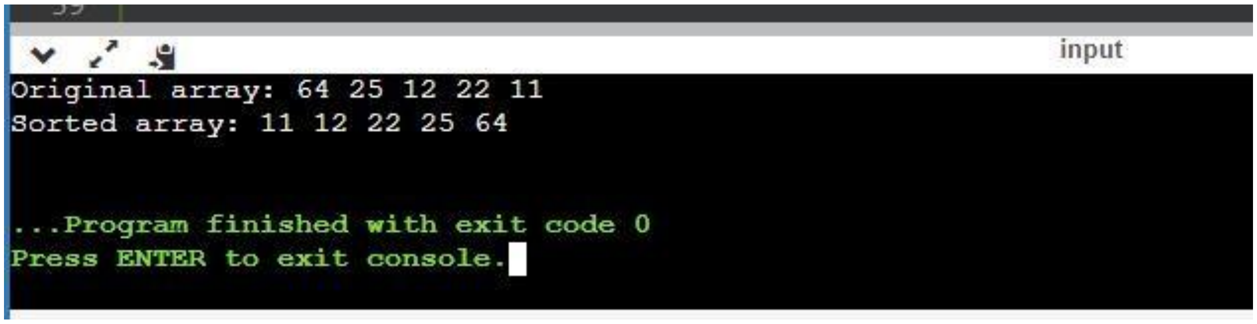
**PRE-EXPERIMENT QUESTIONS:**

- Is Selection Sort Algorithm stable?
- Is Selection Sort Algorithm in-place?
- Does Selection Sort have any advantages or disadvantages compared to other sorting algorithms?

**Program Code:**

```
#include <iostream>

void selectionSort(int arr[], int size) {
   for (int i = 0; i < size - 1; ++i) {
      int minIndex = i;
      for (int j = i + 1; j < size; ++j) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;
         }
      }
      // Swapping the minimum element with the first unsorted element
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}

void printArray(int arr[], int size) {
   for (int i = 0; i < size; ++i) {
      std::cout << arr[i] << " ";
   }
   std::cout << std::endl;
}

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

   std::cout << "Original array: ";
   printArray(arr, size);

   selectionSort(arr, size);

   std::cout << "Sorted array: ";
   printArray(arr, size);

   return 0;
}
```

**Output:**
Students will be able to analysis elements position with larger space taken by them as shown below:



```
Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

1. What is the time complexity of Selection Sort? Can we improve it?
2. What is Selection Sort, and how does it work?
3. What is space complexity of selection sort?

# LAB EXPERIMENT 6

**OBJECTIVE:**

Write a Program for implementation of Fractional Knapsack problem using Greedy Method and 0/1 Knapsack problem using Dynamic Programming.

**BRIEF DESCRIPTION:**

Fractional Knapsack Problem:

Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

The basic idea of the greedy approach is to calculate the ratio profit/weight for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).

This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

**Example:**

Input: arr [] = {{60, 10}, {100, 20}, {120, 30}}, W = 50
Output: 240
Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.
Hence total price will be 60+100+(2/3) (120) = 240.

0/1 Knapsack Problem:

We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

**Example:**

Input: N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}
Output: 3
Explanation: There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.

## PRE-EXPERIMENT QUESTIONS:

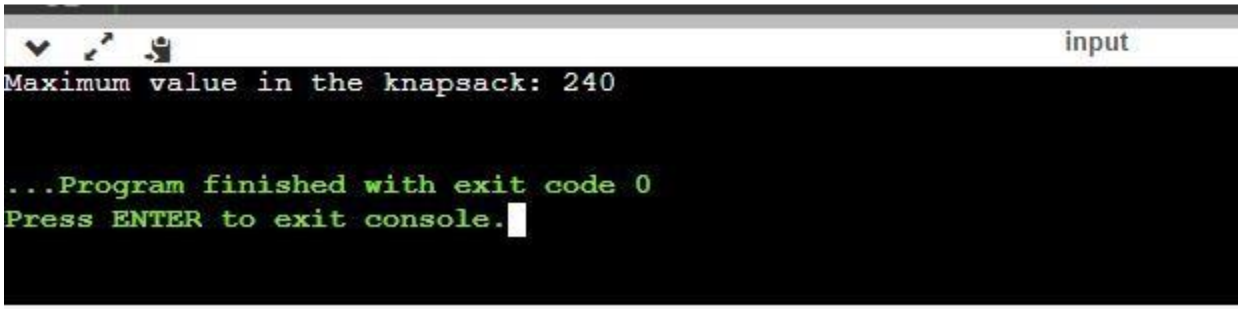
- What is the time complexity of the Greedy algorithm for the Fractional Knapsack problem?
- What is the main advantage of using the Greedy method for the Fractional Knapsack problem?
- How does the time complexity of the Dynamic Programming approach for the 0/1 Knapsack problem depend on the number of items and the knapsack capacity?

## <u>**Program Code:(Fractional Knapsack Problem using Greedy Method)**</u>

```
#include <iostream>
#include <algorithm>
#include <vector>

struct Item {
    int weight;
    int value;
    double valuePerWeight;

    Item(int weight, int value)
        : weight(weight), value(value) {
        valuePerWeight = static_cast<double>(value) / weight;
    }
};

bool compare(Item a, Item b) {
    return a.valuePerWeight > b.valuePerWeight;
}

double fractionalKnapsack(int capacity, std::vector<Item>& items) {
    std::sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;

    for (const auto& item : items) {
        if (capacity >= item.weight) {
            totalValue += item.value;
            capacity -= item.weight;
        } else {
            totalValue += (capacity * item.valuePerWeight);
            break;
        }
    }

    return totalValue;
}

int main() {
    int capacity = 50;
    std::vector<Item> items = {
        Item(10, 60),
        Item(20, 100),
        Item(30, 120),
    };

    double maxValue = fractionalKnapsack(capacity, items);

    std::cout << "Maximum value in the knapsack: " << maxValue << std::endl;

    return 0;
}
```

**Output:**

Students will be able to calculate maximum and optimal profit using Greedy approach as shown below:

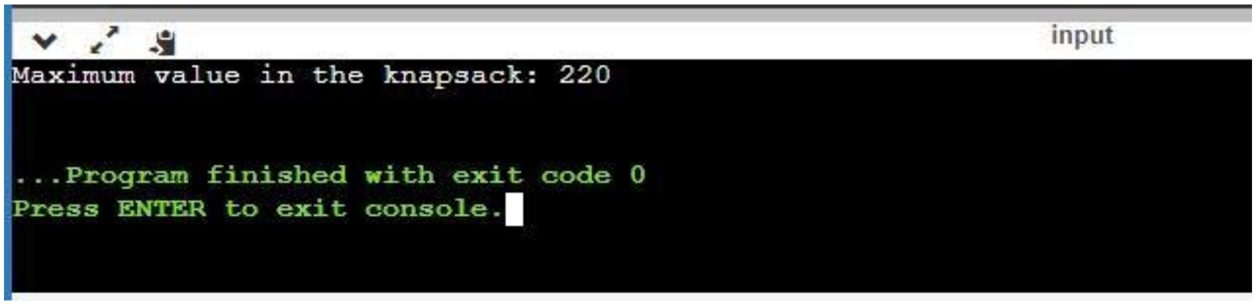


```
Maximum value in the knapsack: 240


...Program finished with exit code 0
Press ENTER to exit console.
```

**Program Code:(0/1 Knapsack Problem using Dynamic Approach)**

```
#include <iostream>
#include <vector>
#include <algorithm>

int knapsack(int capacity, const std::vector<int>& weights, const std::vector<int>& values, int n) {
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= capacity; ++j) {
            if (weights[i - 1] <= j) {
                dp[i][j] = std::max(values[i - 1] + dp[i - 1][j - weights[i - 1]], dp[i - 1][j]);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    return dp[n][capacity];
}

int main() {
    int capacity = 50;
    std::vector<int> weights = {10, 20, 30};
    std::vector<int> values = {60, 100, 120};
    int n = weights.size();

    int maxValue = knapsack(capacity, weights, values, n);

    std::cout << "Maximum value in the knapsack: " << maxValue << std::endl;

    return 0;
}
```

**Output:**
Students will be able to calculate maximum and optimal profit using Greedy approach as shown below:



```
Maximum value in the knapsack: 220


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- How does the Greedy method ensure that it finds the optimal solution for the Fractional Knapsack problem?
- Under what circumstances would the Greedy method produce the same solution as the Dynamic Programming approach for the Fractional Knapsack problem?
- Can the 0/1 Knapsack problem be solved using a Greedy approach? Why or why not?

## LAB EXPERIMENT 7

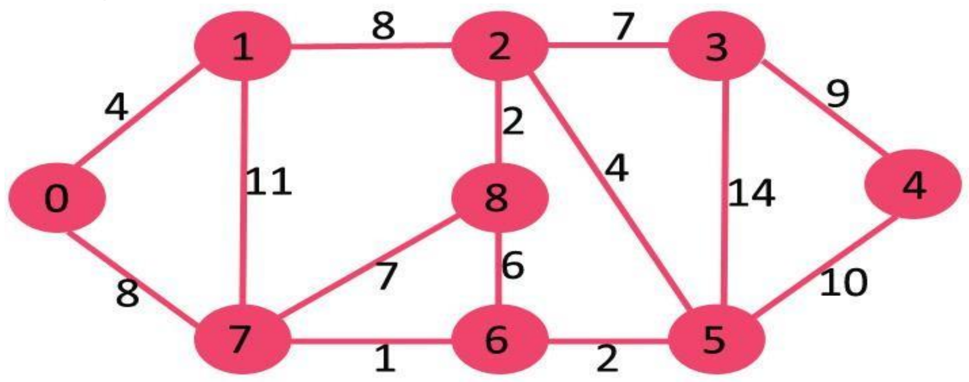
**OBJECTIVE:**

Write a Program to find the shortest path from a given vertex to other vertices in a weighted connected graph using Dijkstra's algorithm.

**BRIEF DESCRIPTION:**
Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

**Example:**

Input: src = 0, the graph is shown below.



**Output: 0 4 12 19 21 11 9 8 14**

Explanation: The distance from 0 to 1 = 4.
The minimum distance from 0 to 2 = 12. 0->1->2
The minimum distance from 0 to 3 = 19. 0->1->2->3
The minimum distance from 0 to 4 = 21. 0->7->6->5->4
The minimum distance from 0 to 5 = 11. 0->7->6->5
The minimum distance from 0 to 6 = 9. 0->7->6
The minimum distance from 0 to 7 = 8. 0->7
The minimum distance from 0 to 8 = 14. 0->1->2->8

**PRE-EXPERIMENT QUESTIONS:**

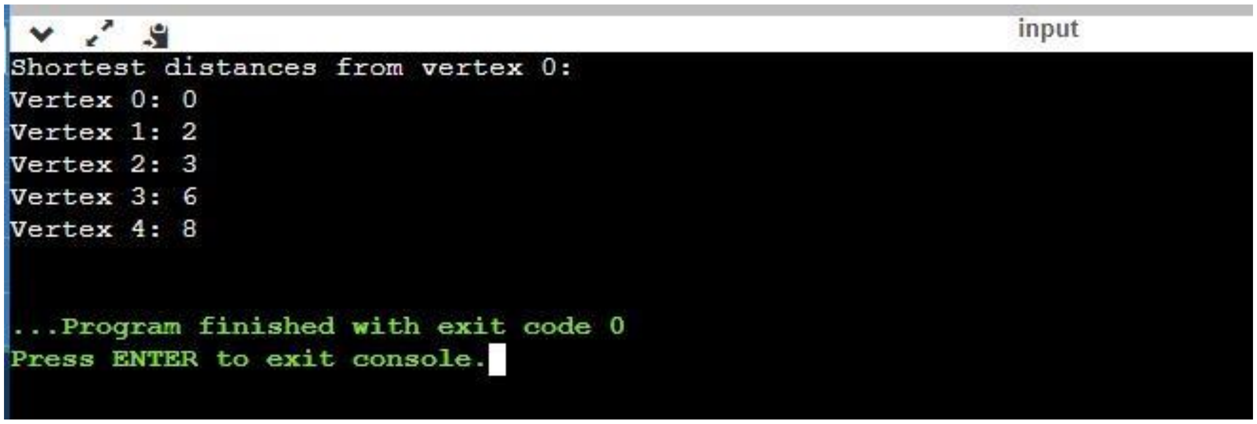
- What is the time complexity of Dijkstra's algorithm?
- Define cost matrix.
- Define directed graph.

**Program Code:**

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

#define INF std::numeric_limits<int>::max()

typedef std::pair<int, int> pii;  // pair of (vertex, weight)

void dijkstra(const std::vector<std::vector<pii>>& graph, int start, std::vector<int>& dist) {
    int numVertices = graph.size();
    dist.resize(numVertices, INF);
    dist[start] = 0;

    std::priority_queue<pii, std::vector<pii>, std::greater<pii>> pq;
    pq.push({start, 0});

    while (!pq.empty()) {
        int u = pq.top().first;
        int uDist = pq.top().second;
        pq.pop();

        // Skip if the current distance is already greater than the stored distance
        if (uDist > dist[u]) {
            continue;
        }

        for (const auto& neighbor : graph[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({v, dist[v]});
            }
        }
    }
}

int main() {
    int numVertices = 5;
    std::vector<std::vector<pii>>  graph(numVertices);

    // Adding edges to the graph
    graph[0].push_back({1,  2});
    graph[0].push_back({2,  4});
    graph[1].push_back({2,  1});
    graph[1].push_back({3,  7});
    graph[2].push_back({3,  3});
    graph[2].push_back({4, 5});
    graph[3].push_back({4, 2});

    int startVertex = 0;
    std::vector<int> dist;

    dijkstra(graph, startVertex, dist);

    // Printing the shortest distances from the start vertex
    std::cout << "Shortest distances from vertex " << startVertex << ":\n";
    for (int i = 0; i < numVertices; ++i) {
        std::cout << "Vertex " << i << ": " << dist[i] << std::endl;
    }

    return 0;
}
```

**Output:**

Students will be able to find shortest map and understand the application GOOGLE MAPS as shown below:



```
Shortest distances from vertex 0:
Vertex 0: 0
Vertex 1: 2
Vertex 2: 3
Vertex 3: 6
Vertex 4: 8


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- How does Dijkstra's algorithm find the shortest path in a weighted graph?
- How does Dijkstra's algorithm handle graphs with unconnected or disconnected vertices?
- Does Dijkstra’s works for negative edge cycle?

## LAB EXPERIMENT 8

**OBJECTIVE:**

Write a Program to find the minimum cost spanning tree (MST) of a given undirected graph using Kruskal's algorithm/Prim's Algorithms.

**BRIEF DESCRIPTION:**

Both Prim's and Kruskal's algorithm finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few major differences between them.

| **Prim's Algorithm** | **Kruskal's Algorithm** |
|---|---|
| It starts to build the Minimum Spanning Tree from any vertex in the graph. | It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph. |
| It traverses one node more than one time to get the minimum distance. | It traverses one node only once. |
| Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps. | Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices. |
| Prim's algorithm gives connected component as well as it works only on connected graph. | Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components |
| Prim's algorithm runs faster in dense graphs. | Kruskal's algorithm runs faster in sparse graphs. |
| It generates the minimum spanning tree starting from the root vertex. | It generates the minimum spanning tree starting from the least weighted edge. |

| **Prim's Algorithm** | **Kruskal's Algorithm** |
|---|---|
| Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc. | Applications of Kruskal algorithm are LAN connection, TV Network etc. |
| Prim's algorithm prefer list data structures. | Kruskal's algorithm prefer heap data structures. |

**PRE-EXPERIMENT QUESTIONS:**

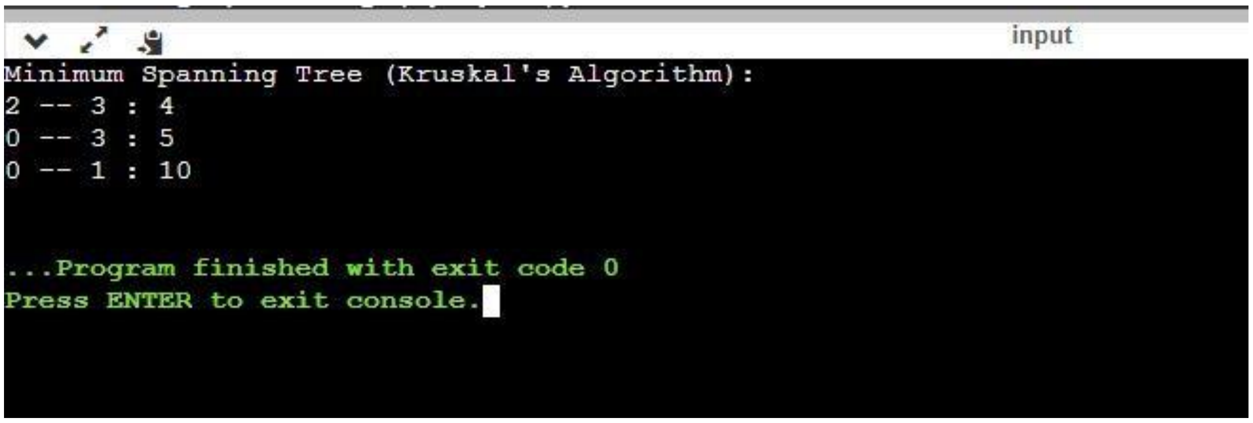
- Why Prim's and Kruskal's MST algorithm fails for Directed Graph?
- Can Kruskal's algorithm and Prim's algorithm handle graphs with cycles? Why or why not?
- What is the time complexity of Kruskal's algorithm and Prim's algorithm?

**Program Code:(Kruskal's Algorithm)**

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Edge {
    int src;
    int dest;
    int weight;

    Edge(int src, int dest, int weight)
        : src(src), dest(dest), weight(weight) {}
};

struct Graph {
    int numVertices;
    std::vector<Edge> edges;

    Graph(int numVertices)
        : numVertices(numVertices) {}

    void addEdge(int src, int dest, int weight) {
        edges.push_back(Edge(src, dest, weight));
    }
    static bool compare(Edge a, Edge b) {
        return a.weight < b.weight;
    }
};

struct DisjointSet {
    std::vector<int> parent;
    std::vector<int> rank;

    DisjointSet(int size) {
        parent.resize(size);
        rank.resize(size);

        for (int i = 0; i < size; ++i) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    int find(int x) {
        if (x != parent[x]) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

std::vector<Edge> kruskalMST(Graph& graph) {
    std::vector<Edge> mst;
    std::sort(graph.edges.begin(), graph.edges.end(), Graph::compare);

    DisjointSet disjointSet(graph.numVertices);

    for (const auto& edge : graph.edges) {
        int srcRoot = disjointSet.find(edge.src);
        int destRoot = disjointSet.find(edge.dest);

        if (srcRoot != destRoot) {
            mst.push_back(edge);
            disjointSet.unite(srcRoot, destRoot);
        }
    }

    return mst;
}

int main() {
    int numVertices = 4;
    Graph graph(numVertices);

    graph.addEdge(0, 1, 10);
    graph.addEdge(0, 2, 6);
    graph.addEdge(0, 3, 5);
    graph.addEdge(1, 3, 15);
    graph.addEdge(2, 3, 4);

    std::vector<Edge> mst = kruskalMST(graph);

    std::cout << "Minimum Spanning Tree (Kruskal's Algorithm):\n";
    for (const auto& edge : mst) {
        std::cout << edge.src << " -- " << edge.dest << " : " << edge.weight << std::endl;
    }

    return 0;
}
```

**Output:**

Students will be able to implement Kruskal’s algorithm as shown below:



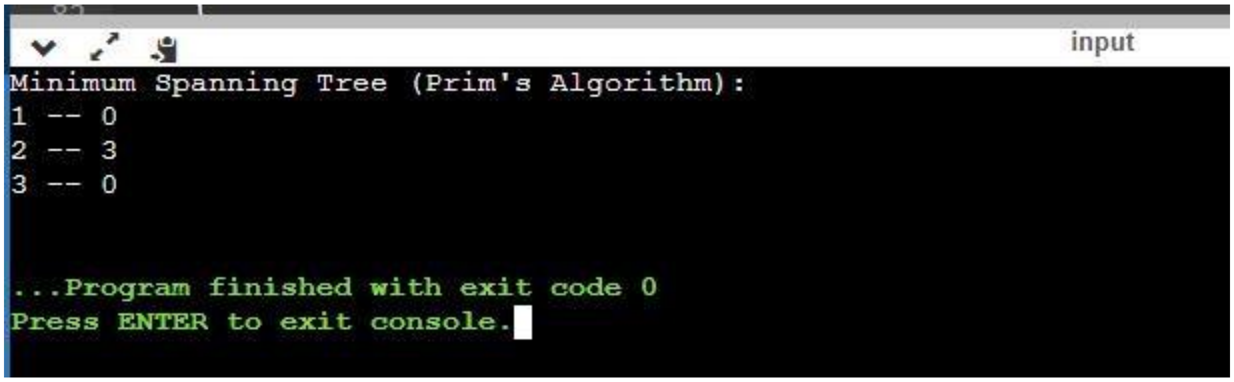
```
Minimum Spanning Tree (Kruskal's Algorithm):
2 -- 3 : 4
0 -- 3 : 5
0 -- 1 : 10


...Program finished with exit code 0
Press ENTER to exit console.
```

**Program Code:(Prim's Algorithm)**

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

struct Edge {
    int dest;
    int weight;

    Edge(int dest, int weight)
        : dest(dest), weight(weight) {}
};

struct Graph {
    int numVertices;
    std::vector<std::vector<Edge>> adjList;

    Graph(int numVertices)
        : numVertices(numVertices), adjList(numVertices) {}

    void addEdge(int src, int dest, int weight) {
        adjList[src].push_back(Edge(dest, weight));
        adjList[dest].push_back(Edge(src, weight));
    }
};

std::vector<Edge> primMST(Graph& graph) {
    std::vector<Edge> mst;
    std::vector<bool> visited(graph.numVertices, false);
    std::vector<int> dist(graph.numVertices, std::numeric_limits<int>::max());
    std::vector<int> parent(graph.numVertices, -1);

    auto compare = [](Edge a, Edge b) {
        return a.weight > b.weight;
    };
    std::priority_queue<Edge, std::vector<Edge>, decltype(compare)> pq(compare);

    int startVertex = 0;
    dist[startVertex] = 0;
    pq.push(Edge(startVertex, 0));

    while (!pq.empty()) {
        int u = pq.top().dest;
        pq.pop();

        visited[u] = true;

        for (const auto& edge : graph.adjList[u]) {
            int v = edge.dest;
            int weight = edge.weight;

            if (!visited[v] && weight < dist[v]) {
                dist[v] = weight;
                parent[v] = u;
                pq.push(Edge(v, weight));
            }
        }
    }

    for (int i = 1; i < graph.numVertices; ++i) {
        mst.push_back(Edge(i, parent[i]));
    }

    return mst;
}

int main() {
    int numVertices = 4;
    Graph graph(numVertices);

    graph.addEdge(0, 1, 10);
    graph.addEdge(0, 2, 6);
    graph.addEdge(0, 3, 5);
    graph.addEdge(1, 3, 15);
    graph.addEdge(2, 3, 4);

    std::vector<Edge> mst = primMST(graph);

    std::cout << "Minimum Spanning Tree (Prim's Algorithm):\n";
    for (const auto& edge : mst) {
        std::cout << edge.dest << " -- " << edge.weight << std::endl;
    }

    return 0;
}
```

**Output:**

Students will be able to implement Prim’s algorithm as shown below:



```
Minimum Spanning Tree (Prim's Algorithm):
1 -- 0
2 -- 3
3 -- 0


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- How does Prim's algorithm select the next vertex to expand the minimum cost spanning tree?
- What is the key difference between Kruskal's algorithm and Prim's algorithm?
- How does Kruskal's algorithm ensure that it forms a minimum cost spanning tree?

## LAB EXPERIMENT 9

**OBJECTIVE:**

Write a Program to implement N-Queens problem using back tracking.

**BRIEF DESCRIPTION:**

Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

Backtracking can also be said as an improvement to the brute force approach. So basically, the idea behind the backtracking technique is that it searches for a solution to a problem among all the available options. Initially, we start the backtracking from one possible option and if the problem is solved with that selected option then we return the solution else we backtrack and select another option from the remaining available options. There also might be a case where none of the options will give you the solution and hence we understand that backtracking won't give any solution to that particular problem. We can also say that backtracking is a form of recursion.

This is because the process of finding the solution from the various option available is repeated recursively until we don't find the solution or we reach the final state. So we can conclude that backtracking at every step eliminates those choices that cannot give us the solution and proceeds to those choices that have the potential of taking us to the solution.

There are three types of problems in backtracking –

- Decision Problem – In this, we search for a feasible solution.
- Optimization Problem – In this, we search for the best solution.
- Enumeration Problem – In this, we find all feasible solutions.

This program solves the N-Queens problem using backtracking. The printBoard function is used to display the board configuration. The isSafe function checks if it is safe to place a queen in a particular position on the board. The solveNQueensUtil function recursively places queens in safe positions and backtracks when a solution is found or when it reaches an unsafe configuration. The solveNQueens function initializes the board and calls solveNQueensUtil to find and print all the valid solutions. In the main function, the program sets N as the number of queens and the size of the board.

**PRE-EXPERIMENT QUESTIONS:**

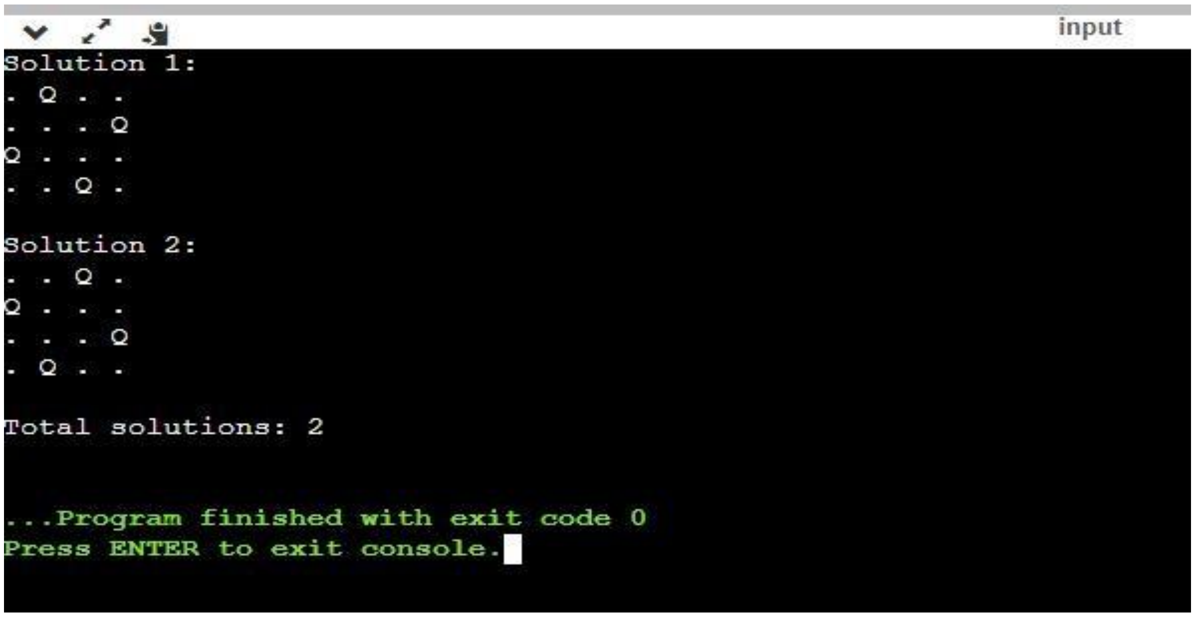
- What is backtracking, and how does it work in the context of problem-solving?
- How does backtracking solve the N-Queens problem?
- What is the time complexity of the backtracking solution for the N-Queens problem?

**Program Code:**

```
#include <iostream>
#include <vector>

void printBoard(const std::vector<std::vector<char>>& board) {
    for (const auto& row : board) {
        for (char cell : row) {
            std::cout << cell << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

bool isSafe(const std::vector<std::vector<char>>& board, int row, int col, int N) {
    // Check if there is a queen in the same column
    for (int i = 0; i < row; ++i) {
        if (board[i][col] == 'Q') {
            return false;
        }
    }

    // Check if there is a queen in the upper-left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; --i, --j) {
        if (board[i][j] == 'Q') {
            return false;
        }
    }

    // Check if there is a queen in the upper-right diagonal
    for (int i = row, j = col; i >= 0 && j < N; --i, ++j) {
        if (board[i][j] == 'Q') {
            return false;
        }
    }

    return true;
}

void solveNQueensUtil(std::vector<std::vector<char>>& board, int row, int N, int& count) {
    if (row == N) {
        // Solution found
        ++count;
        std::cout << "Solution " << count << ":\n";
        printBoard(board);
        return;
    }

    for (int col = 0; col < N; ++col) {
        if (isSafe(board, row, col, N)) {
            board[row][col] = 'Q';  // Place the queen

            solveNQueensUtil(board, row + 1, N, count);

            board[row][col] = '.';  // Backtrack, remove the queen
        }
    }
}

void solveNQueens(int N) {
    std::vector<std::vector<char>> board(N, std::vector<char>(N, '.'));

    int count = 0;
    solveNQueensUtil(board, 0, N, count);

    std::cout << "Total solutions: " << count << std::endl;
}

int main() {
    int N = 4;  // Number of queens and size of the board

    solveNQueens(N);

    return 0;
}
```

**Output:**

Students will be able to implement the concept of backtracking and can solve puzzles such as Sudoku as shown below:



```
Solution 1:
. Q . .
. . . Q
Q . . .
. . Q .

Solution 2:
. . Q .
Q . . .
. . . Q
. Q . .

Total solutions: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- How does the isSafe function determine if a queen can be placed in a given position on the chessboard?
- What is the role of the solveNQueensUtil function in the N-Queens program?
- How can the N-Queens program be modified to find and store all possible solutions, rather than just printing them?

## LAB EXPERIMENT 10

**OBJECTIVE:**

Write a Program to check whether a given graph is connected or not using DFS method.

**BRIEF DESCRIPTION:**

Both DFS and BFS are widely used graph traversal algorithms with different characteristics. DFS explores deeply before backtracking, while BFS explores broadly in a level-by-level manner. The choice between DFS and BFS depends on the problem requirements and the structure of the graph being traversed.

Depth-First Search is a graph traversal algorithm that explores vertices and their edges in a depth ward motion. It starts from a selected vertex and explores as far as possible along each branch before backtracking. DFS uses a stack (either an explicit stack or the call stack) to keep track of vertices to visit. The algorithm marks each visited vertex to avoid revisiting it. DFS is often implemented using recursion, where the function calls itself to explore neighboring vertices. It is useful for solving problems such as finding connected components, detecting cycles, and traversing trees or graphs.

Breadth-First Search is a graph traversal algorithm that explores vertices and their edges in a breadth ward motion. It starts from a selected vertex and visits all of its neighboring vertices before moving on to their neighbors. BFS uses a queue to keep track of vertices to visit. The first-in, first-out (FIFO) order ensures visiting vertices in increasing order of distance from the starting vertex. The algorithm marks each visited vertex to avoid revisiting it. BFS is often implemented using a while loop that dequeues a vertex, explores its neighbors, and enqueues them for further exploration. It is useful for solving problems such as finding the shortest path, finding connected components, and traversing trees or graphs in a level-by-level manner.

This program checks whether a given graph is connected or not using the DFS method. The DFS function performs a depth-first search traversal starting from a given vertex and marks all visited vertices. The isConnected function initializes a vector to track the visited status of each vertex and calls the DFS function starting from the first vertex. It then checks if all vertices are visited to determine if the graph is connected.

In the main function, a sample graph is created using an adjacency list representation. The graph is then passed to the isConnected function, and based on the result, it prints whether the graph is connected or not.

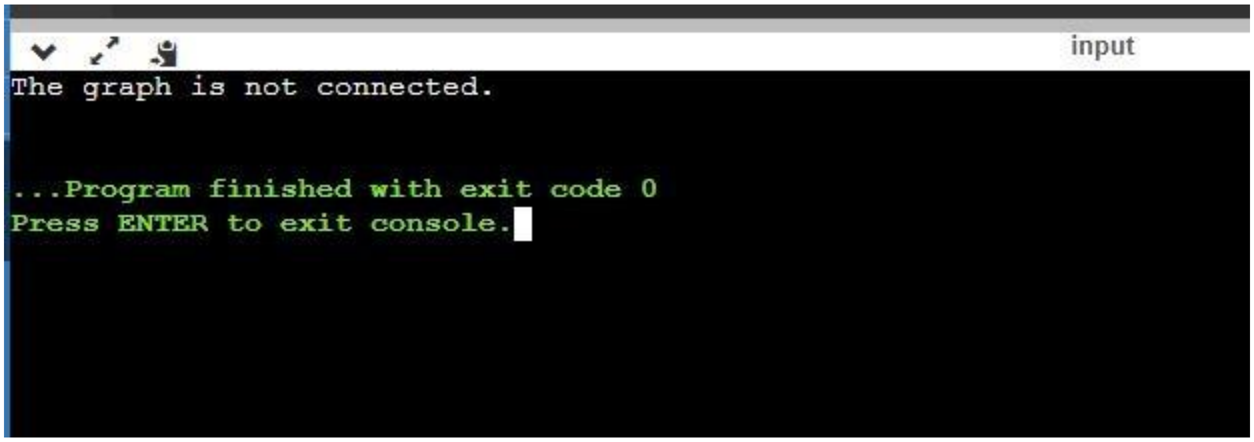
**PRE-EXPERIMENT QUESTIONS:**

- What does it mean for a graph to be connected?
- What is the main difference between a connected graph and a disconnected graph?
- How can we determine whether a graph is connected or disconnected?

**Program Code:**

```
#include <iostream>
#include <vector>

void DFS(const std::vector<std::vector<int>>& graph, std::vector<bool>& visited, int vertex) {
    visited[vertex] = true;

    for (int neighbor : graph[vertex]) {
        if (!visited[neighbor]) {
            DFS(graph, visited, neighbor);
        }
    }
}

bool isConnected(const std::vector<std::vector<int>>& graph, int numVertices) {
    std::vector<bool> visited(numVertices, false);

    // Perform DFS starting from the first vertex
    DFS(graph, visited, 0);

    // Check if all vertices are visited
    for (bool status : visited) {
        if (!status) {
            return false;
        }
    }

    return true;
}

int main() {
    int numVertices = 5;
    std::vector<std::vector<int>>  graph(numVertices);

    // Add edges to the graph
    graph[0] = {1, 2};
    graph[1] = {0, 3};
    graph[2] = {0, 3};
    graph[3] = {1, 2};
    graph[4] = {};  // isolated vertex

    bool connected = isConnected(graph, numVertices);

    if (connected) {
        std::cout << "The graph is connected." << std::endl;
    } else {
        std::cout << "The graph is not connected." << std::endl;
    }

    return 0;
}
```

**Output:**
Students will be able to analysis the connected or disconnected graph performance as shown below:



```
The graph is not connected.


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- What is the time complexity of determining whether a graph is connected or disconnected using DFS or BFS?
- How does the program you provided determine whether a graph is connected or disconnected?
- Can a disconnected graph contain isolated vertices? Why or why not?

## LAB EXPERIMENT 11

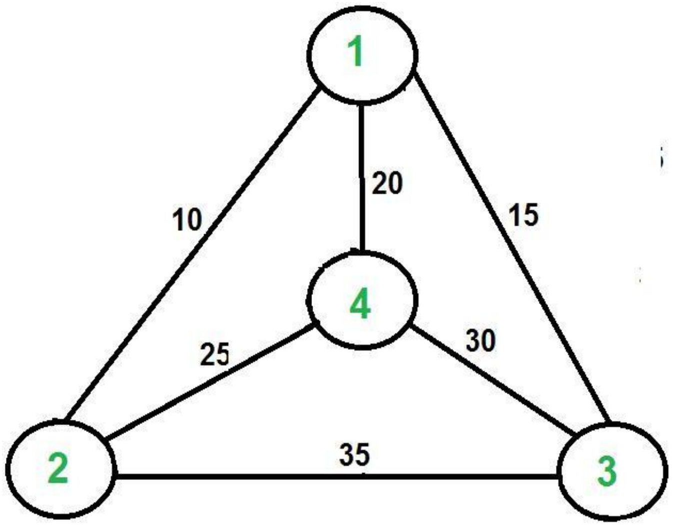
**OBJECTIVE:**

Write a program to implement the Travelling Salesman Problem (TSP).

**BRIEF DESCRIPTION:**

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hard problem. There is no polynomial-time know solution for this problem. The following are different solutions for the traveling salesman problem.



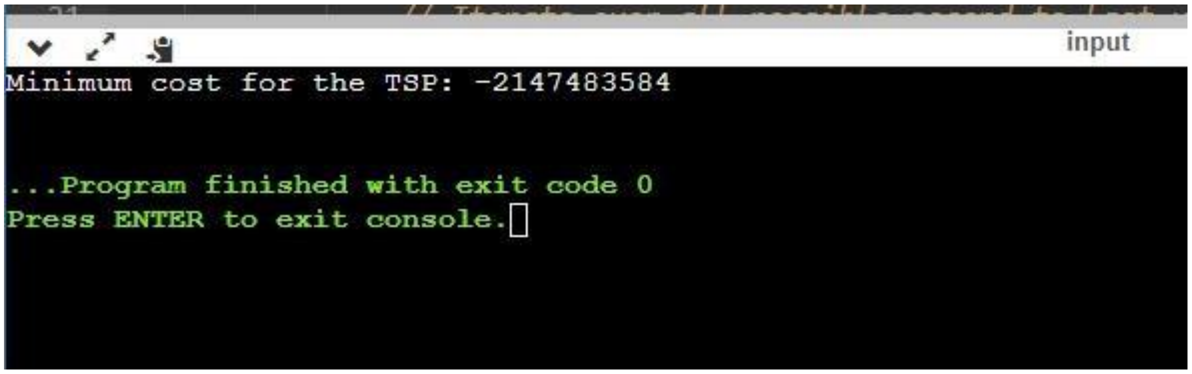
**PRE-EXPERIMENT QUESTIONS:**

- What is the Traveling Salesman Problem (TSP)?
- What is the significance of the TSP in the field of computer science?
- What are the different approaches to solving the TSP?

**Program Code:**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <climits>

int tspDynamicProgramming(std::vector<std::vector<int>>& graph, int n) {
    int numSubsets = 1 << n;  // Number of subsets of the vertex set

    // Initialize the dynamic programming table
    std::vector<std::vector<int>> dp(numSubsets, std::vector<int>(n, INT_MAX));

    // Initialize the base case
    dp[1][0] = 0;

    // Iterate over all subsets of vertices
    for (int subset = 1; subset < numSubsets; ++subset) {
        for (int last = 0; last < n; ++last) {
            // Check if the last vertex is in the subset
            if (subset & (1 << last)) {
                // Iterate over all possible second-to-last vertices
                for (int secondLast = 0; secondLast < n; ++secondLast) {
                    // Check if the second-to-last vertex is different from the last vertex and is in the subset
                    if (secondLast != last && (subset & (1 << secondLast))) {
                        // Compute the cost of reaching the last vertex via the second-to-last vertex
                        int cost = graph[secondLast][last] + dp[subset ^ (1 << last)][secondLast];
                        dp[subset][last] = std::min(dp[subset][last], cost);
                    }
                }
            }
        }
    }

    // Find the minimum cost of reaching the starting vertex from any other vertex
    int minCost = INT_MAX;
    for (int last = 1; last < n; ++last) {
        int cost = graph[last][0] + dp[numSubsets - 1][last];
        minCost = std::min(minCost, cost);
    }

    return minCost;
}

int main() {
    int n = 4;  // Number of cities

    // Graph representation (adjacency matrix)
    std::vector<std::vector<int>> graph = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int minCost = tspDynamicProgramming(graph, n);

    std::cout << "Minimum cost for the TSP: " << minCost << std::endl;

    return 0;
}
```

**<u>Output:</u>**
Students will be able to implement TSP using dynamic programming as shown below:



```
Minimum cost for the TSP: -2147483584


...Program finished with exit code 0
Press ENTER to exit console.
```

**POST EXPERIMENT QUESTIONS:**

- Name a heuristic algorithm commonly used to approximate solutions for the TSP.
- What is the main limitation of the brute-force approach to solving the TSP?
- What are the different approaches to solving the TSP?

This lab manual has been updated by

Prof. Vimmi Malhotra (vimmi.malhotra@ggnindia.dronacharya.info)

Crosschecked By

HOD CSE

Please spare some time to provide your valuable feedback.