EXPERIMENT 1

Date:

Objective: Implement Recursive Binary search and determine the time taken to search an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.

Software Used: C++

Theory: Binary search is a fundamental algorithm used for efficiently searching a sorted list. The key idea behind binary search is that it repeatedly divides the search interval in half until the target value is found or the search interval becomes empty.

In the case of **recursive binary search**, the algorithm calls itself with a smaller range (either the left half or the right half of the list) based on the comparison between the middle element and the target element.

How Binary Search Works:

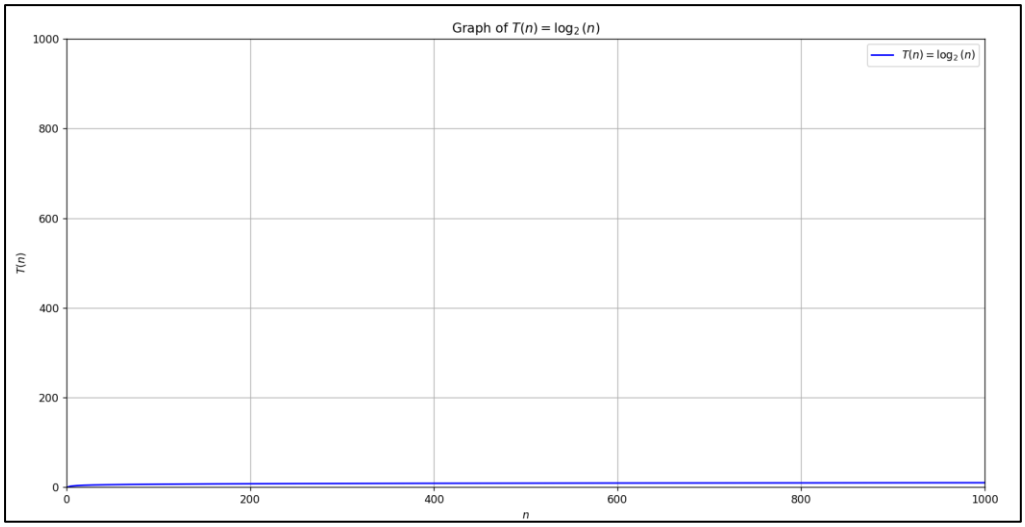
1. Start by selecting the middle element of the list.
2. If the middle element is the target, the search is complete.
3. If the target is smaller than the middle element, search only in the left half of the list.
4. If the target is larger than the middle element, search only in the right half of the list.
5. Repeat the process recursively until the target element is found or the search range is empty.

The time complexity of binary search is **O(log n)**, where n is the number of elements in the list. This is because, with each recursive call, the search space is halved, leading to a logarithmic number of steps. The logarithmic time complexity makes binary search much faster than linear search (O(n)), especially for large datasets. The recurrence relation for Recursive Binary Search would be:
$T(n) = T(n/2) + O(1)$
$T(n/2) = T(n/4) + O(1)$
$T(n/4) = T(n/8) + O(1)$
$T(n/8) = T(n/16) + O(1)$
and so on.

Substituting the values of T (n/2), T (n/4) and T (n/8) in T (n),
$T(n) = T(n/4) + 2 \cdot O(1)$
$T(n) = T(n/8) + 3 \cdot O(1)$
$T(n) = T(n/16) + 4 \cdot O(1)$
The general form for iterations :
$T(n) = T(n/2k) + k \cdot O(1)$

After sufficient number of iterations, the size of the last division would be 1.
Thus, $n/2k = 1$ is the limiting condition for the recurrence relation.

On further solving,
$2k = n$
$k = \log_2 n$
Therefore,
T (n) = T (1) + log2 n · O(1) T (n) = $O(\log_2 n)$
Therefore, T (n) = Θ(log(n))



Algorithm:

```
binarySearch(arr, x, low, high)
   if low > high
      return False
   else
      mid = (low + high) / 2
      if x == arr[mid]
         return mid
      else if x > arr[mid]        // x is on the right side
         return binarySearch(arr, x, mid + 1, high)
      else                        // x is on the left side
         return binarySearch(arr, x, low, mid - 1)
```

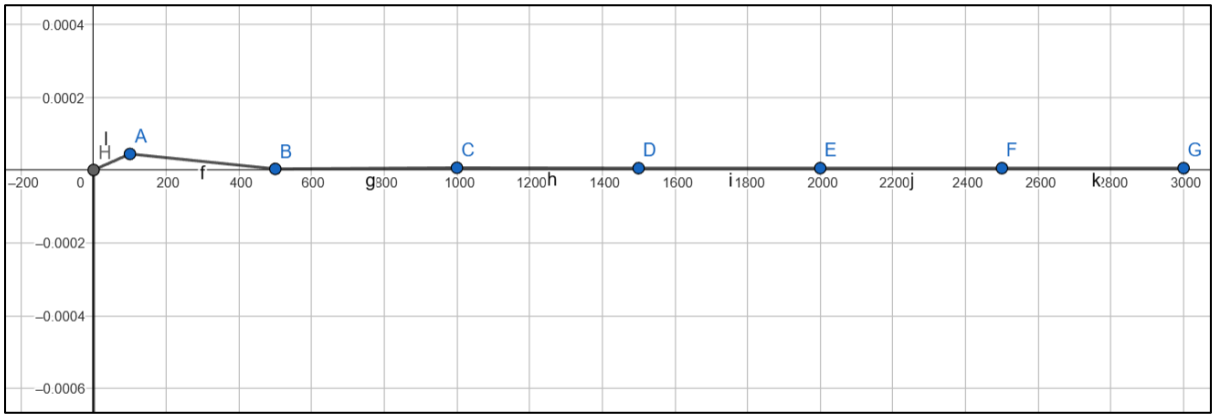
Code:

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

void binary_search(int array[], int search, int min_index, int max_index) {
   if (min_index > max_index) {
      cout << "Element not found in the list." << endl;
      return;
   }
   int mid_index = min_index + (max_index - min_index) / 2;
```

```
    int mid = array[mid_index];
    if (mid < search) {
        binary_search(array, search, mid_index + 1, max_index);
    }
    else if (mid > search) {
        binary_search(array, search, min_index, mid_index - 1);
    }
    else {
        cout << "Element found at index " << mid_index << endl;
    }
}

int main() {
    int sizes[] = {100, 500, 1000, 1500, 2000, 2500, 3000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        int* array = new int[n]; // Dynamically allocate array of size n
        // Fill the array with sorted elements from 1 to n
        for (int j = 0; j < n; j++) {
            array[j] = j + 1;
        }
        int search = n; // Search for the last element (worst-case scenario)
        // Start measuring time
        auto start = chrono::high_resolution_clock::now();
        binary_search(array, search, 0, n - 1);
        // End measuring time
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> time_taken = end - start;
        // Output the results
        cout << "Time taken for n = " << setw(7) << n << ": " << fixed << setprecision(6) <<
time_taken.count() << " seconds" << endl << endl;
        delete[] array; // Deallocate memory
    }
    return 0;
}
```

Output:

```
/tmp/MsrD4CFWrP.o
Element found at index 99
Time taken for n =     100: 0.000044 seconds

Element found at index 499
Time taken for n =     500: 0.000003 seconds

Element found at index 999
Time taken for n =    1000: 0.000006 seconds

Element found at index 1499
Time taken for n =    1500: 0.000005 seconds

Element found at index 1999
Time taken for n =    2000: 0.000005 seconds

Element found at index 2499
Time taken for n =    2500: 0.000005 seconds

Element found at index 2999
Time taken for n =    3000: 0.000005 seconds
```



Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 2

Date:

Objective: Sort a given set of elements using Quick sort method and determine the time taken 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.

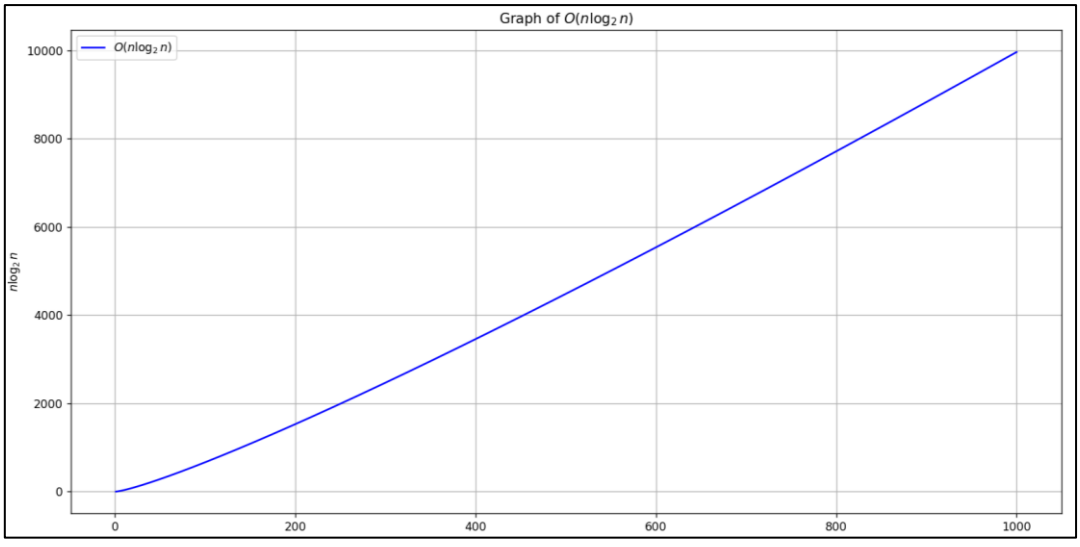
Software Used: C++

Theory: Sorting is one of the fundamental operations in computer science, widely used for organizing data. Among various sorting algorithms, Quick Sort is one of the most efficient algorithms, especially for large datasets. It follows a divide-and-conquer strategy and is commonly used due to its efficiency and simplicity.
Quick Sort is a comparison-based sorting algorithm that operates by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.
Steps Involved in Quick Sort:

1. Partitioning the Array: The pivot element is selected (typically the last element), and the array is rearranged so that all elements less than the pivot are on its left and all elements greater than the pivot are on its right.
2. Recursive Sorting: The sub-arrays formed by partitioning are recursively sorted using the same approach.
3. Combining the Results: After sorting, the pivot is placed in its correct position, and the process continues for each partition.

Time Complexity:

- Best and Average Case: O(nlogn)
- Worst Case: $O(n^2)$ (occurs when the smallest or largest element is consistently chosen as the pivot)



Algorithm:

```
mergeSort(arr, lb, ub) {
    if (lb < ub) {
        mid = (lb + ub) / 2
        mergeSort(arr, lb, mid)
        mergeSort(arr, mid + 1, ub);
        merge(arr, lb, mid, ub)
   }
}
merge(arr, lb, mid, ub) {
    I = lb; j = mid + 1; k = ub;
    while (i <= mid && j <= ub) {
        if (arr[i] <= arr[j]) {
            arr1[k] = arr[i];
            i++; k++;
        }
        else {
            arr1[k] = arr[j]
            j++; i++;
        }
        if (i > mid) {
            while (j <= ub) {
                    arr1[k] = arr[j];
                    j++; k++;
                }
        }
        else {
            while (i <= mid) {
                    arr1[k] = arr[i];
                    i++; k++;
                }
        }
}
```

Code:

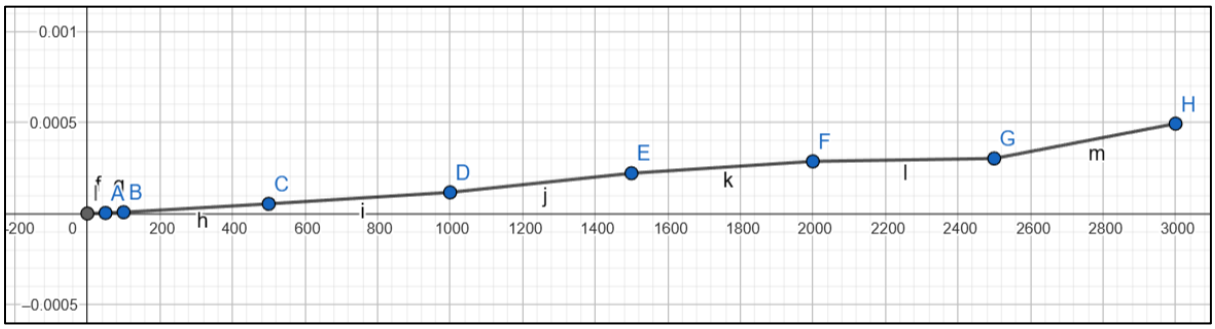
```
#include <iostream>
#include <chrono>
#include <iomanip>
#include <cstdlib>
using namespace std;

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

```
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

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

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

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    int* arr = new int[n]; // Dynamically allocate array of size n
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "Array before sorting: ";
    displayArray(arr, n);
    // Start measuring time
    auto start = chrono::high_resolution_clock::now();
    quickSort(arr, 0, n - 1);
    // End measuring time
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> time_taken = end - start;
    cout << "Array after sorting: ";
    displayArray(arr, n);
    cout << "Time taken to sort the inputted array: "
         << fixed << setprecision(6) << time_taken.count()
         << " seconds" << endl;
    delete[] arr; // Deallocate memory
    // Arrays of various sizes to demonstrate time taken
    int sizes[] = {50, 100, 500, 1000, 1500, 2000, 2500, 3000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    // Sorting automatically generated arrays of various sizes
    for (int i = 0; i < num_sizes; i++) {
        int size = sizes[i];
```

```
        int* autoArr = new int[size];
        // Fill the array with random elements
        for (int j = 0; j < size; j++) {
            autoArr[j] = rand() % 100000; // Generate random numbers between 0 and 99999
        }
        // Start measuring time
        start = chrono::high_resolution_clock::now();
        quickSort(autoArr, 0, size - 1);
        // End measuring time
        end = chrono::high_resolution_clock::now();
        time_taken = end - start;
        cout << "Time taken for n = " << setw(7) << size << ": "
             << fixed << setprecision(6) << time_taken.count()
             << " seconds" << endl;
        delete[] autoArr; // Deallocate memory
    }
    return 0;
}
```

Output:

```
Enter the size of the array: 6
Enter the elements of the array: 12 45 65 42 7 8
Array before sorting: 12 45 65 42 7 8
Array after sorting: 7 8 12 42 45 65
Time taken to sort the inputted array: 0.000001 seconds
Time taken for n =      50: 0.000003 seconds
Time taken for n =     100: 0.000007 seconds
Time taken for n =     500: 0.000053 seconds
Time taken for n =    1000: 0.000116 seconds
Time taken for n =    1500: 0.000221 seconds
Time taken for n =    2000: 0.000286 seconds
Time taken for n =    2500: 0.000302 seconds
Time taken for n =    3000: 0.000493 seconds
```



Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 3

Date:

Objective: Implement Knapsack Problem using Greedy Approach.

Software Used: C++

Theory: The Knapsack Problem is a combinatorial optimization problem. Given a set of items, each with a weight and a value, the goal is to determine the number of each item to include in a knapsack so that the total weight does not exceed a given capacity, and the total value is maximized.

The Fractional Knapsack Problem allows breaking items into smaller pieces, unlike the 0/1 Knapsack Problem where each item must be either taken entirely or left behind. The Greedy Approach is optimal for solving the Fractional Knapsack Problem.

Greedy Approach: The Greedy Approach involves selecting items based on the highest value-to-weight ratio. This strategy maximizes the value added to the knapsack while ensuring that the weight constraint is respected.

Steps Involved:

1. Compute the value-to-weight ratio for each item.
2. Sort the items based on this ratio in descending order.
3. Add the items to the knapsack in this order, either fully or fractionally, until the knapsack is full.

The Greedy Approach works efficiently for the Fractional Knapsack Problem due to the ability to break items into fractions, which ensures that the highest possible value is achieved.

Algorithm:
Input: Array of items with value and weight, Capacity W
Output: Maximum value that can be carried in the knapsack

1. Compute value/weight ratio for each item.
2. Sort items in descending order of value/weight ratio.
3. Initialize currentWeight = 0, finalValue = 0.
4. For each item in the sorted list:
    a. If adding the item does not exceed W:
        - Add the full item to the knapsack.
        - Update currentWeight and finalValue.
    b. Else:
        - Add the fractional part of the item that can fit.
        - Calculate the fraction and update finalValue.
        - Break the loop (no more items can be added).
5. Return finalValue as the maximum value that can be carried.

Code:

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

struct Item {
   int weight;
   int value;
};

bool compare(Item a, Item b) {
   double r1 = (double)a.value / a.weight;
   double r2 = (double)b.value / b.weight;
   return r1 > r2;
}

double knapsackGreedy(int W, Item items[], int n) {
   sort(items, items + n, compare);
   int currentWeight = 0;
   double finalValue = 0.0;
   for (int i = 0; i < n; i++) {
      if (currentWeight + items[i].weight <= W) {
         currentWeight += items[i].weight;
         finalValue += items[i].value;
      }
      else {
         int remainingWeight = W - currentWeight;
         finalValue += items[i].value * ((double)remainingWeight / items[i].weight);
         break;
      }
   }
   return finalValue;
}

int main() {
   int n, W;
   cout << "Enter the number of items: ";
   cin >> n;
   Item items[n];
   for (int i = 0; i < n; i++) {
      cout << "Enter weight and value for item " << i + 1 << ": ";
      cin >> items[i].weight >> items[i].value;
   }
   cout << "Enter the maximum weight capacity of the knapsack: ";
   cin >> W;
   double maxValue = knapsackGreedy(W, items, n);
   cout << "Maximum value we can obtain = " << maxValue << endl;
   return 0; }
```

Output:

```
Enter the number of items: 5
Enter weight and value for item 1: 10 34
Enter weight and value for item 2: 20 54
Enter weight and value for item 3: 5 23
Enter weight and value for item 4: 15 65
Enter weight and value for item 5: 25 39
Enter the maximum weight capacity of the knapsack: 55
Maximum value we can obtain = 183.8
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 4

Date:

Objective: Implement 0/1 Knapsack Problem using Dynamic Programming method.

Software Used: C++

Theory:
The dynamic programming approach for solving the 0/1 Knapsack Problem involves constructing a table `dp[i][w]` where `i` represents the items considered so far and `w` represents the current weight limit. The value at `dp[i][w]` is the maximum value that can be achieved with weight limit `w` using the first `i` items. The solution is built by examining two options: including the item or not, and taking the best of both.

Algorithm:
Algorithm Knapsack(values, weights, W, n):
1. Create a 2D array dp[n+1][W+1] where n is the number of items and W is the maximum weight capacity.
2. Initialize dp[0][w] = 0 for all 0 <= w <= W (base case: no items).
3. For i = 1 to n:
  a. For w = 0 to W:
    i. If weights[i-1] <= w:
      - dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
    ii. Else:
      - dp[i][w] = dp[i-1][w]
4. The value at dp[n][W] is the maximum value achievable with weight W.
5. End the algorithm.

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

void knapsack(int W, int weights[], int values[], int n, int kp[][100]) {
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            kp[i][w] = 0;
         else if (weights[i - 1] <= w)
            kp[i][w] = max(kp[i - 1][w], kp[i - 1][w - weights[i - 1]] + values[i - 1]);
         else
            kp[i][w] = kp[i - 1][w];
} } }

void findItems(int W, int weights[], int values[], int n, int kp[][100]) {
   int i = n, k = W;
   cout << "Items included in the knapsack:\n";
```

```
    while (i > 0 && k > 0) {
        if (kp[i][k] != kp[i - 1][k]) {
            cout << "Item " << i << " (Weight: " << weights[i - 1] << ", Value: " << values[i - 1]
<< ")\n";
            k -= weights[i - 1];
        }
        i--;
}}

int main() {
    int n, W;
    cout << "Enter the number of items: ";
    cin >> n;
    int weights[n], values[n];
    cout << "Enter the weight and value of each item:\n";
    for (int i = 0; i < n; i++) {
        cout << "Item " << i + 1 << " - Weight: ";
        cin >> weights[i];
        cout << "Item " << i + 1 << " - Value: ";
        cin >> values[i];
    }
    cout << "Enter the maximum capacity of the knapsack: ";
    cin >> W;
    int kp[100][100];
    knapsack(W, weights, values, n, kp);
    cout << "Maximum value that can be obtained: " << kp[n][W] << endl;
    findItems(W, weights, values, n, kp);
    return 0;
}
```

Output:

```
Enter the number of items: 3
Enter the weight and value of each item:
Item 1 - Weight: 1
Item 1 - Value: 2
Item 2 - Weight: 2
Item 2 - Value: 3
Item 3 - Weight: 3
Item 3 - Value: 4
Enter the maximum capacity of the knapsack: 3
Maximum value that can be obtained: 5
Items included in the knapsack:
Item 2 (Weight: 2, Value: 3)
Item 1 (Weight: 1, Value: 2)
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 5 (a)

Date:

Objective: From a given starting node in a digraph, print all the nodes reachable by using BFS method.

Software Used: C++

Theory: Breadth-First Search (BFS) is a fundamental graph traversal algorithm. It explores all the nodes at the present "depth" level before moving on to the nodes at the next depth level. BFS is especially useful for finding the shortest path in an unweighted graph and for exploring all reachable nodes from a given starting node.
In the context of a directed graph (digraph), BFS can be used to find and print all nodes that are reachable from a given starting node.

Working of BFS in a Directed Graph:

1. Start from the given node and explore all its neighbors.
2. Mark each visited node to avoid revisiting it.
3. Use a queue to manage the nodes to be explored next. This ensures that nodes are explored in the correct order (level by level).
4. Continue exploring until all reachable nodes have been visited.

Algorithm:
Algorithm BFS(graph, start_node):
1. Initialize an empty queue Q.
2. Initialize a visited array/array to mark all nodes as unvisited.
3. Mark the start_node as visited and enqueue it into Q.
4. While Q is not empty:
   a. Dequeue a node from Q and print it.
   b. For each neighbor of the dequeued node:
      i. If the neighbor is not visited:
         - Mark it as visited.
         - Enqueue it into Q.
5. End the algorithm.

Code:

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

void printAdjacencyMatrix(int adjMatrix[][5], int n) {
   cout << "Adjacency Matrix:" << endl;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         cout << adjMatrix[i][j] << " ";
```

```
        }
        cout << endl;
    }
}

void BFS(int startNode, int adjMatrix[][5], int n) {
    bool visited[n]; // Array to track visited nodes
    for (int i = 0; i < n; i++) {
        visited[i] = false; // Initialize all nodes as not visited
    }
    queue<int> q; // Queue for BFS
    visited[startNode] = true;
    q.push(startNode);
    cout << "Nodes reachable from node " << startNode << " using BFS:" << endl;
    while (!q.empty()) {
        int currentNode = q.front();
        q.pop();
        cout << currentNode << " ";
        // Explore all adjacent nodes of the current node
        for (int i = 0; i < n; i++) {
            if (adjMatrix[currentNode][i] == 1 && !visited[i]) { // If there is an edge and it's not
visited
                visited[i] = true;
                q.push(i);
            }
        }
    }
    cout << endl;
}

int main() {
    int n = 5;
    int adjMatrix[5][5] = {
        {0, 1, 1, 0, 0}, // Edges from node 0 to nodes 1 and 2
        {0, 0, 0, 1, 0}, // Edge from node 1 to node 3
        {0, 0, 0, 0, 1}, // Edge from node 2 to node 4
        {0, 0, 1, 0, 1}, // Edges from node 3 to nodes 2 and 4
        {1, 0, 0, 0, 0}  // Edge from node 4 to node 0
    };
    printAdjacencyMatrix(adjMatrix, n);
    int startNode;
    cout << "Enter the starting node (0 to 4): ";
    cin >> startNode;
    BFS(startNode, adjMatrix, n);
    return 0;
}
```

Output:

```
Adjacency Matrix:
0 1 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 1 0 1
1 0 0 0 0
Enter the starting node (0 to 4): 2
Nodes reachable from node 2 using BFS:
2 4 0 1 3
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 5 (b)

Date:

Objective: From a given starting node in a digraph, print all the nodes reachable by using DFS method.

Software Used: C++

Theory: Depth-First Search (DFS) is a fundamental graph traversal algorithm used to explore nodes and edges of a graph. It starts from a given starting node and explores as far as possible along each branch before backtracking. DFS is particularly useful for tasks such as finding connected components, detecting cycles, and solving puzzles where all possible paths need to be explored.

Working of DFS in a Directed Graph (Digraph):

1. Start from the Given Node: Begin traversal from the specified starting node.
2. Explore Each Node: Recursively explore each unvisited adjacent node. This is done by visiting a node, marking it as visited, and then moving to one of its unvisited neighbors.
3. Mark Nodes as Visited: To avoid revisiting nodes and to prevent infinite loops, maintain a record of visited nodes.
4. Use a Stack: The recursion stack (or an explicit stack in an iterative implementation) helps manage the nodes to be explored next, ensuring that nodes are explored as deep as possible before backtracking.
5. Backtrack When Necessary: If a node has no unvisited neighbors, backtrack to the previous node and continue exploring from there until all reachable nodes have been visited.

Algorithm:

DFS(graph, start_node):
1. Initialize a visited array to mark all nodes as unvisited.
2. Initialize an empty stack S.
3. Push start_node onto S and mark it as visited.
4. While S is not empty:
   a. Pop a node from S and print it.
   b. For each neighbor of the node:
      i. If the neighbor is not visited:
         - Mark the neighbor as visited.
         - Push the neighbor onto S.
5. End the algorithm.

Code:

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

void printAdjacencyMatrix(int adjMatrix[][6], int n) {
```

```
    cout << "Adjacency Matrix:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << adjMatrix[i][j] << " ";
        }
        cout << endl;
    }
}

void DFS(int startNode, int adjMatrix[][6], int n) {
    bool visited[n]; // Array to track visited nodes
    for (int i = 0; i < n; i++) {
        visited[i] = false; // Initialize all nodes as not visited
    }
    stack<int> s;
    visited[startNode] = true;
    s.push(startNode);
    cout << "Nodes reachable from node " << startNode << " using DFS: ";
    while (!s.empty()) {
        int currentNode = s.top();
        s.pop();
        cout << currentNode << " ";
        for (int i = 0; i < n; i++) {
            if (adjMatrix[currentNode][i] == 1 && !visited[i]) { // If there is an edge and it's not
visited
                visited[i] = true;
                s.push(i);
            }
        }
    }
}

int main() {
    int n = 6;
    int adjMatrix[6][6] = {
        {0, 1, 0, 1, 0, 0}, // Edges from node 0 to nodes 1 and 3
        {0, 0, 1, 1, 0, 0}, // Edges from node 1 to nodes 2 and 3
        {0, 0, 0, 0, 1, 0}, // Edge from node 2 to node 4
        {0, 0, 0, 0, 1, 1}, // Edges from node 3 to nodes 4 and 5
        {0, 0, 0, 0, 0, 1}, // Edge from node 4 to node 5
        {0, 0, 0, 0, 0, 0}  // Node 5 has no outgoing edges
    };
    printAdjacencyMatrix(adjMatrix, n);
    int startNode;
    cout << "Enter the starting node (0 to 5): ";
    cin >> startNode;
    DFS(startNode, adjMatrix, n);
    return 0; }
```

Output:

```
Adjacency Matrix:
0 1 0 1 0 0
0 0 1 1 0 0
0 0 0 0 1 0
0 0 0 0 1 1
0 0 0 0 0 1
0 0 0 0 0 0
Enter the starting node (0 to 5): 3
Nodes reachable from node 3 using DFS: 3 5 4
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 6 (a)

Date:

Objective: Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm

Software Used: C++

Theory: A Minimum Cost Spanning Tree (MCST) is a subset of edges in a connected, undirected graph such that the tree formed by these edges includes all the vertices of the graph with the minimum possible total edge weight. There are various algorithms to compute the MCST of a graph, including Kruskal's Algorithm and Prim's Algorithm. Prim's Algorithm is a greedy algorithm that builds the spanning tree by continuously adding the shortest edge connected to the growing tree.
Prim's Algorithm begins with a single vertex and grows the Minimum Spanning Tree (MST) by adding one edge at a time. At each step, the algorithm selects the smallest edge that connects a vertex inside the MST to a vertex outside of it, ensuring the tree remains connected and has minimal weight.

Algorithm:

1. Start with any arbitrary vertex. This vertex is added to the MST set.
2. Among the edges that connect vertices in the MST to vertices outside the MST, choose the one with the smallest weight.
3. Add the selected edge and its associated vertex to the MST.
4. Repeat the process until all vertices are included in the MST.

Code:

```
#include <iostream>
#include <climits> // For INT_MAX
using namespace std;
#define V 7 // Number of vertices in the graph (A, B, C, D, E, F, G)

// Function to find the vertex with the minimum key value, from the set of vertices not yet
included in MST
int minKey(int key[], bool mstSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}

// Function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V]) {
   cout << "Edge \tWeight\n";
   for (int i = 1; i < V; i++)
```

```
        cout << (char)(parent[i] + 'A') << " - " << (char)(i + 'A') << "\t" << graph[i][parent[i]]
<< " \n";
}

// Function to construct and print MST for a graph represented using adjacency matrix
representation
void primMST(int graph[V][V]) {
    int parent[V];  // Array to store constructed MST
    int key[V];     // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices not yet included in MST
    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
    // Always include the first vertex in MST.
    key[0] = 0;     // Make key 0 so that this vertex is picked as the first vertex
    parent[0] = -1; // First node is always the root of MST
    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not yet included in MST
        int u = minKey(key, mstSet);
        // Add the picked vertex to the MST Set
        mstSet[u] = true;
        // Update key value and parent index of the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not yet included in MST
        for (int v = 0; v < V; v++)
            // graph[u][v] is non zero only for adjacent vertices of u
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    // Print the constructed MST
    printMST(parent, graph);
}

int main() {
    int graph[V][V];
    cout << "Enter the adjacency matrix for the graph (enter 0 for no edge):\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cin >> graph[i][j];
        }
    }
    primMST(graph);
    return 0;
}
```

Output:

```
Enter the adjacency matrix for the graph (enter 0 for no edge):
0 3 6 0 0 0 0
3 0 2 4 0 0 0
6 2 0 1 4 2 0
0 4 1 0 2 0 4
0 0 4 2 0 2 1
0 0 2 0 2 0 1
0 0 0 4 1 1 0
Edge    Weight
A - B   3
B - C   2
C - D   1
D - E   2
G - F   1
E - G   1
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 6 (b)

Date:

Objective: Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm

Software Used: C++

Theory: Depth-First Search (DFS) is a fundamental graph traversal algorithm used to explore nodes and edges of a graph. It starts from a given starting node and explores as far as possible along each branch before backtracking. DFS is particularly useful for tasks such as finding connected components, detecting cycles, and solving puzzles where all possible paths need to be explored.

Working of DFS in a Directed Graph (Digraph):

6. **Start from the Given Node:** Begin traversal from the specified starting node.
7. **Explore Each Node:** Recursively explore each unvisited adjacent node. This is done by visiting a node, marking it as visited, and then moving to one of its unvisited neighbors.
8. **Mark Nodes as Visited:** To avoid revisiting nodes and to prevent infinite loops, maintain a record of visited nodes.
9. **Use a Stack:** The recursion stack (or an explicit stack in an iterative implementation) helps manage the nodes to be explored next, ensuring that nodes are explored as deep as possible before backtracking.
10. **Backtrack When Necessary:** If a node has no unvisited neighbors, backtrack to the previous node and continue exploring from there until all reachable nodes have been visited.

Algorithm:

DFS(graph, start_node):
1. Initialize a visited array to mark all nodes as unvisited.
2. Initialize an empty stack S.
3. Push start_node onto S and mark it as visited.
4. While S is not empty:
   a. Pop a node from S and print it.
   b. For each neighbor of the node:
      i. If the neighbor is not visited:
         - Mark the neighbor as visited.
         - Push the neighbor onto S.
5. End the algorithm.

Code:

```
#include <iostream>
#include <algorithm>
#include <iomanip>
using namespace std;
const int MAX = 20;
int parent[MAX];
```

```
// Structure to store edges of the graph
struct Edge {
    int u, v, weight;
};

// Find the root (with path compression)
int find(int i) {
    if (parent[i] == i)
        return i;
    else
        return parent[i] = find(parent[i]);
}

// Union of two subsets
void union_set(int u, int v) {
    parent[u] = v;
}

// Kruskal's Algorithm to find the Minimum Cost Spanning Tree
void kruskal(Edge edges[], int n, int m) {
    // Sort edges based on weight
    sort(edges, edges + m, [](Edge a, Edge b) {
        return a.weight < b.weight;
    });
    // Initialize the parent array
    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }
    int mst_weight = 0;
    cout << "Edges in the Minimum Cost Spanning Tree:\n";
    cout << "Edge \tWeight\n";
    for (int i = 0, count = 0; i < m && count < n - 1; i++) {
        int u_root = find(edges[i].u);
        int v_root = find(edges[i].v);

        if (u_root != v_root) {
            // Include this edge in MST
            cout << edges[i].u + 1 << " - " << edges[i].v + 1 << "\t" << edges[i].weight << endl;
            mst_weight += edges[i].weight;
            union_set(u_root, v_root);
            count++;
        }
    }
    cout << "Total weight of MST: " << mst_weight << endl;
}

int main() {
```

```
    int n;  // Number of vertices
    cout << "Enter the number of vertices (up to 20): ";
    cin >> n;
    int graph[MAX][MAX];
    cout << "Enter the adjacency matrix:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> graph[i][j];
        }
    }
    // Store the edges
    Edge edges[MAX * MAX];
    int edge_count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (graph[i][j] != 0) {
                edges[edge_count++] = {i, j, graph[i][j]};
            }
        }
    }
    kruskal(edges, n, edge_count);
    return 0;
}
```

Output:

```
Enter the number of vertices (up to 20): 10
Enter the adjacency matrix:
0 0 12 5 1 0 0 0 0 0
0 0 6 0 0 4 16 0 0 0
12 6 0 0 19 17 0 0 0 0
5 0 0 0 9 0 0 0 3 0
1 0 19 9 0 8 0 7 10 0
0 4 17 0 8 0 2 14 0 13
0 16 0 0 0 2 0 0 0 18
0 0 0 0 7 14 0 0 15 11
0 0 0 3 10 0 0 15 0 0
0 0 0 0 0 13 18 11 0 0
Edges in the Minimum Cost Spanning Tree:
Edge    Weight
1 - 5   1
6 - 7   2
4 - 9   3
2 - 6   4
1 - 4   5
2 - 3   6
5 - 8   7
5 - 6   8
8 - 10  11
Total weight of MST: 47
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 7

Date:

Objective: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

Software Used: C++

Theory: Dijkstra's Algorithm is a well-known algorithm used to find the shortest paths from a given source vertex to all other vertices in a weighted, connected graph. This algorithm only works when all the edge weights are non-negative. It is a greedy algorithm that progressively builds up the shortest path tree.

Algorithm:

1. Initialization:
   - Set the distance of the source vertex s to 0, and the distance of all other vertices to infinity (∞).
   - Mark all vertices as unvisited.
   - Use a priority queue to store vertices by their tentative distances.
2. Selecting Minimum Distance Vertex:
   - Select the unvisited vertex u with the smallest tentative distance from the priority queue. This vertex is now considered visited, and its shortest path from the source is finalized.
3. Relaxation of Adjacent Vertices:
   - For each neighbour v of the vertex u, update the tentative distance to v as:
   - distance(v) = min(distance(v), distance(u) + w(u, v))
   - If the tentative distance to v is updated, add v to the priority queue with the updated distance.
4. Repeat:
   - Continue the process until all vertices have been visited, and the shortest paths to all vertices have been finalized.

Code:

```
#include <iostream>
#include <climits> // For INT_MAX

using namespace std;

#define MAX 100   // Maximum number of vertices in the graph

int graph[MAX][MAX];   // Adjacency matrix representation of the graph
int dist[MAX];         // Array to store the shortest distance from source
bool visited[MAX];     // Array to check if a vertex is already visited
int parent[MAX];       // Array to store the path
```

```
// Function to find the vertex with the minimum distance value
int findMinDistance(int vertices) {
    int minDist = INT_MAX;
    int minIndex = -1;
    for (int i = 0; i < vertices; i++) {
        if (!visited[i] && dist[i] <= minDist) {
            minDist = dist[i];
            minIndex = i;
        }
    }
    return minIndex;
}

// Function to print the shortest path from source to a given vertex
void printPath(int target) {
    if (parent[target] == -1) {
        cout << target << " ";
        return;
    }
    printPath(parent[target]);
    cout << target << " ";
}

// Dijkstra's algorithm implementation
void dijkstra(int source, int vertices) {
    // Initialize all distances to infinity and visited to false
    for (int i = 0; i < vertices; i++) {
        dist[i] = INT_MAX;
        visited[i] = false;
        parent[i] = -1; // No parent initially
    }
    // Distance of source vertex from itself is always 0
    dist[source] = 0;
    // Loop to find the shortest path for all vertices
    for (int count = 0; count < vertices - 1; count++) {
        // Find the vertex with the minimum distance
        int u = findMinDistance(vertices);
        if (u == -1) break; // All remaining vertices are unreachable
        visited[u] = true;
        // Update the distance of adjacent vertices
        for (int v = 0; v < vertices; v++) {
            // Only update dist[v] if:
            // 1. Vertex v is not visited
            // 2. There is an edge from u to v (graph[u][v] > 0)
            // 3. Total weight of the path from source to v through u is smaller than the current
value of dist[v]
            if (!visited[v] && graph[u][v] != 0 && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
```

```
                dist[v] = dist[u] + graph[u][v];
                parent[v] = u; // Set u as the parent of v
            }
        }
    }
    // Print the shortest distances and paths
    cout << "Vertex\tDistance from Source\tPath" << endl;
    for (int i = 0; i < vertices; i++) {
        cout << i << "\t\t" << (dist[i] == INT_MAX ? -1 : dist[i]) << "\t\t\t\t\t";
        if (dist[i] != INT_MAX) {
            printPath(i);
        } else {
            cout << "No path";
        }
        cout << endl;
    }
}

int main() {
    int vertices, source;
    cout << "Enter the number of vertices: ";
    cin >> vertices;
    // Input the adjacency matrix
    cout << "Enter the adjacency matrix (use 0 for no direct path between vertices):" << endl;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            cin >> graph[i][j];
        }
    }
    cout << "Enter the source vertex: ";
    cin >> source;
    // Call Dijkstra's algorithm
    dijkstra(source, vertices);
    return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the adjacency matrix (use 0 for no direct path between vertices):
0 2 0 6 0
2 0 3 8 0
0 3 0 0 5
6 8 0 0 9
0 0 5 9 0
Enter the source vertex: 0
Vertex  Distance from Source    Path
0       0                      0
1       2                      0 1
2       5                      0 1 2
3       6                      0 3
4       10                     0 1 2 4
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 8

Date:

Objective: Consider the problem of N queen on an (NxN) chessboard. Two queens are said to attack each other if they are on the same row, column, or diagonal. Implements backtracking algorithm to solve the problem i.e. place N non-attacking queens on the board.

Software Used: C++

Theory: The N-Queens problem is a classic combinatorial problem where the goal is to place N queens on an N×N chessboard such that no two queens threaten each other. In chess, a queen can attack another queen if they are on the same row, column, or diagonal. Therefore, the problem asks us to place queens in such a way that none of them share these three attacking conditions.

Algorithm:

1. Initialize the board:
   - Create an empty N×N board (2D array) where all cells are initialized to '.' (empty).
2. Recursive Function (Solve N-Queens):
   - Define a recursive function solve_n_queens(board, row, n) that tries to place queens row by row.
3. Base Case (Stopping Condition):
   - If all queens are placed (i.e., row == N), print the current board configuration as a valid solution and return True.
4. Column Placement Loop:
   - For each column in the current row, do the following:
   - Check if placing the queen in the current cell (row, col) is safe by calling the helper function is_safe(board, row, col, n).
5. Safe Position Check (is_safe function):
   - The is_safe(board, row, col, n) function checks three conditions:
   - There is no other queen in the same column above the current row.
   - There is no other queen on the upper-left diagonal.
   - There is no other queen on the upper-right diagonal.
6. Place the Queen:
   - If the position (row, col) is safe, place a queen ('Q') in the current cell (row, col).
7. Recursive Call:
   - Make a recursive call to the function solve_n_queens(board, row+1, n) to place a queen in the next row.
8. Backtracking:
   - If placing the queen in the current column does not lead to a solution, remove the queen ('backtrack') from (row, col) and continue trying other columns in the same row.
9. No Solution:
   - If no valid position is found for the current row, return False to trigger backtracking in the previous row.
10. Driver Function:
    - Call the driver function n_queens(n) which initializes the board and calls the recursive solver function starting from the first row.

Code:

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

// Function to check if the queen can be placed at board[row][col]
bool isSafe(int queens[], int row, int col) {
    for (int i = 0; i < row; i++) {
        int qCol = queens[i];
        // Check same column or diagonal conflicts
        if (qCol == col || abs(qCol - col) == abs(i - row)) {
            return false;
        }
    }
    return true;
}

// Recursive function to solve the N-Queens problem
void solveNQueens(int queens[], int row, int n) {
    if (row == n) {
        // All queens are placed, print the solution
        cout << "<";
        for (int i = 0; i < n; i++) {
            cout << queens[i] + 1;         // Output column positions (1-based index)
            if (i != n - 1) cout << ", ";  // Add commas between positions
        }
        cout << ">" << endl;
        return;
    }
    // Try placing queen in each column of the current row
    for (int col = 0; col < n; col++) {
        if (isSafe(queens, row, col)) {
            queens[row] = col;                // Place the queen at (row, col)
            solveNQueens(queens, row + 1, n);  // Recur for the next row
        }
    }
}

// Driver function
int main() {
    int n;
    cout << "Enter the number of queens (N): ";
    cin >> n;
    int queens[n];  // Array to store the column positions of queens
    solveNQueens(queens, 0, n);
    return 0;
}
```

Output:

```
Enter the number of queens (N): 4
<2, 4, 1, 3>
<3, 1, 4, 2>
```

```
Enter the number of queens (N): 8
<1, 5, 8, 6, 3, 7, 2, 4>
<1, 6, 8, 3, 7, 4, 2, 5>
<1, 7, 4, 6, 8, 2, 5, 3>
<1, 7, 5, 8, 2, 4, 6, 3>
<2, 4, 6, 8, 3, 1, 7, 5>
<2, 5, 7, 1, 3, 8, 6, 4>
<2, 5, 7, 4, 1, 8, 6, 3>
<2, 6, 1, 7, 4, 8, 3, 5>
<2, 6, 8, 3, 1, 4, 7, 5>
<2, 7, 3, 6, 8, 5, 1, 4>
<2, 7, 5, 8, 1, 4, 6, 3>
<2, 8, 6, 1, 3, 5, 7, 4>
<3, 1, 7, 5, 8, 2, 4, 6>
<3, 5, 2, 8, 1, 7, 4, 6>
<3, 5, 2, 8, 6, 4, 7, 1>
<3, 5, 7, 1, 4, 2, 8, 6>
<3, 5, 8, 4, 1, 7, 2, 6>
<3, 6, 2, 5, 8, 1, 7, 4>
<3, 6, 2, 7, 1, 4, 8, 5>
<3, 6, 2, 7, 5, 1, 8, 4>
<3, 6, 4, 1, 8, 5, 7, 2>
<3, 6, 4, 2, 8, 5, 7, 1>
<3, 6, 8, 1, 4, 7, 5, 2>
```

```
<3, 6, 8, 1, 5, 7, 2, 4>
<3, 6, 8, 2, 4, 1, 7, 5>
<3, 7, 2, 8, 5, 1, 4, 6>
<3, 7, 2, 8, 6, 4, 1, 5>
<3, 8, 4, 7, 1, 6, 2, 5>
<4, 1, 5, 8, 2, 7, 3, 6>
<4, 1, 5, 8, 6, 3, 7, 2>
<4, 2, 5, 8, 6, 1, 3, 7>
<4, 2, 7, 3, 6, 8, 1, 5>
<4, 2, 7, 3, 6, 8, 5, 1>
<4, 2, 7, 5, 1, 8, 6, 3>
<4, 2, 8, 5, 7, 1, 3, 6>
<4, 2, 8, 6, 1, 3, 5, 7>
<4, 6, 1, 5, 2, 8, 3, 7>
<4, 6, 8, 2, 7, 1, 3, 5>
<4, 6, 8, 3, 1, 7, 5, 2>
<4, 7, 1, 8, 5, 2, 6, 3>
<4, 7, 3, 8, 2, 5, 1, 6>
<4, 7, 5, 2, 6, 1, 3, 8>
<4, 7, 5, 3, 1, 6, 8, 2>
<4, 8, 1, 3, 6, 2, 7, 5>
<4, 8, 1, 5, 7, 2, 6, 3>
<4, 8, 5, 3, 1, 7, 2, 6>
<5, 1, 4, 6, 8, 2, 7, 3>
<5, 1, 8, 4, 2, 7, 3, 6>
```

```
<5, 1, 8, 6, 3, 7, 2, 4>
<5, 2, 4, 6, 8, 3, 1, 7>
<5, 2, 4, 7, 3, 8, 6, 1>
<5, 2, 6, 1, 7, 4, 8, 3>
<5, 2, 8, 1, 4, 7, 3, 6>
<5, 3, 1, 6, 8, 2, 4, 7>
<5, 3, 1, 7, 2, 8, 6, 4>
<5, 3, 8, 4, 7, 1, 6, 2>
<5, 7, 1, 3, 8, 6, 4, 2>
<5, 7, 1, 4, 2, 8, 6, 3>
<5, 7, 2, 4, 8, 1, 3, 6>
<5, 7, 2, 6, 3, 1, 4, 8>
<5, 7, 2, 6, 3, 1, 8, 4>
<5, 7, 4, 1, 3, 8, 6, 2>
<5, 8, 4, 1, 3, 6, 2, 7>
<5, 8, 4, 1, 7, 2, 6, 3>
<6, 1, 5, 2, 8, 3, 7, 4>
<6, 2, 7, 1, 3, 5, 8, 4>
<6, 2, 7, 1, 4, 8, 5, 3>
<6, 3, 1, 7, 5, 8, 2, 4>
<6, 3, 1, 8, 4, 2, 7, 5>
<6, 3, 1, 8, 5, 2, 4, 7>
<6, 3, 5, 7, 1, 4, 2, 8>
<6, 3, 5, 8, 1, 4, 2, 7>
<6, 3, 7, 2, 4, 8, 1, 5>
```

```
<6, 3, 7, 2, 8, 5, 1, 4>
<6, 3, 7, 4, 1, 8, 2, 5>
<6, 4, 1, 5, 8, 2, 7, 3>
<6, 4, 2, 8, 5, 7, 1, 3>
<6, 4, 7, 1, 3, 5, 2, 8>
<6, 4, 7, 1, 8, 2, 5, 3>
<6, 8, 2, 4, 1, 7, 5, 3>
<7, 1, 3, 8, 6, 4, 2, 5>
<7, 2, 4, 1, 8, 5, 3, 6>
<7, 2, 6, 3, 1, 4, 8, 5>
<7, 3, 1, 6, 8, 5, 2, 4>
<7, 3, 8, 2, 5, 1, 6, 4>
<7, 4, 2, 5, 8, 1, 3, 6>
<7, 4, 2, 8, 6, 1, 3, 5>
<7, 5, 3, 1, 6, 8, 2, 4>
<8, 2, 4, 1, 7, 5, 3, 6>
<8, 2, 5, 3, 1, 7, 4, 6>
<8, 3, 1, 6, 2, 5, 7, 4>
<8, 4, 1, 3, 6, 2, 7, 5>
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 9

Date:

Objective: Implement Knapsack Problem based on Backtracking algorithm.

Software Used: C++

Theory: The Knapsack Problem is a classical optimization problem where the objective is to select a subset of items to include in a knapsack such that the total weight of the selected items does not exceed a given limit (capacity), and the total value of the selected items is maximized.

0/1 Knapsack Problem:
The objective is to select a subset of items such that:

- The total weight of the selected items is less than or equal to the knapsack capacity W.
- The total value of the selected items is maximized.

The solution involves deciding, for each item, whether to include it in the knapsack (1) or not include it (0). Thus, it's called a 0/1 decision problem.

Backtracking Approach to the Knapsack Problem:
Backtracking is a recursive approach that explores all potential solutions by trying all combinations of items. The idea is to build a solution incrementally by making choices at each step, and backtrack whenever a decision leads to an invalid solution.
In the 0/1 Knapsack Problem:

1. For each item, you have two choices: either include it or exclude it from the knapsack.
2. The algorithm explores both possibilities recursively:
    - Include the item: Add its value to the total value and its weight to the total weight.
    - Exclude the item: Skip the item and move to the next.
3. At each stage, the algorithm checks if the current weight exceeds the capacity or if the maximum value has been achieved.
4. If the solution is invalid (i.e., total weight exceeds capacity), it backtracks by removing the last included item and explores other possibilities.

Algorithm:

```
function Knapsack(i, current_weight, current_value):
   if i == n or current_weight > W:
      return current_value

   exclude = Knapsack(i + 1, current_weight, current_value)
   if current_weight + w[i] <= W:
      include = Knapsack(i + 1, current_weight + w[i], current_value + v[i])
   else:
      include = 0
   return max(include, exclude)
```

Code:

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

```
int n, W;  // Number of items and capacity of knapsack
int weights[100], values[100];  // Arrays to store weights and values of items

// Recursive function to solve the Knapsack problem using backtracking
int knapsack(int i, int current_weight, int current_value) {
    // Base case: If all items are considered or knapsack is full
    if (i == n || current_weight > W) {
        return current_value;
    }
    // Exclude the current item and move to the next item
    int exclude = knapsack(i + 1, current_weight, current_value);
    // Include the current item (if it doesn't exceed the capacity)
    int include = 0;
    if (current_weight + weights[i] <= W) {
        include = knapsack(i + 1, current_weight + weights[i], current_value + values[i]);
    }
    // Return the maximum value obtained by including or excluding the current item
    return max(include, exclude);
}

int main() {
    // Input number of items and knapsack capacity
    cout << "Enter the number of items: ";
    cin >> n;
    cout << "Enter the capacity of the knapsack: ";
    cin >> W;
    // Input the weights and values of each item
    cout << "Enter the weights of the items: ";
    for (int i = 0; i < n; i++) {
        cin >> weights[i];
    }
    cout << "Enter the values of the items: ";
    for (int i = 0; i < n; i++) {
        cin >> values[i];
    }
    int max_value = knapsack(0, 0, 0);
    cout << "The maximum value that can be obtained is: " << max_value << endl;
    return 0;
}
```

Output:

```
Enter the number of items: 5
Enter the capacity of the knapsack: 15
Enter the weights of the items: 5 23 10 6 1
Enter the values of the items: 12 83 64 21 8
The maximum value that can be obtained is: 76
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

EXPERIMENT 10

Date:

Objective: Implement TSP based on branch and bound technique.

Software Used: C++

Theory:
Branch and Bound (B&B) is an algorithm design paradigm for solving combinatorial and optimization problems. It systematically explores the solution space by dividing it into smaller subproblems (branching) and evaluating the bounds of possible solutions to eliminate subproblems that cannot yield better solutions than the current best (bounding).

- Node: Represents a partial solution or a state in the search tree.
- Branching: Dividing the problem into smaller subproblems, typically by making decisions about the next city to visit.
- Bounding: Calculating a lower bound on the cost of the solution that can be achieved from a particular node. If this bound is greater than the current best solution, that branch can be pruned (discarded).
- Solution Space: The set of all possible routes that can be taken to solve the TSP.

Algorithm:
```
function calculate_bound(node, cost_matrix):
   if node.level == n:
      return node.cost + cost_matrix[node.path[-1]][0]  # Complete the tour

   bound = node.cost
   for i from 0 to n-1:
      if i is not in node.path:
         min_edge = minimum cost_matrix[i][j] for all j not in node.path
         bound += min_edge

   return bound
```

Code:
```
#include <bits/stdc++.h>
using namespace std;

const int N = 4;
// final_path[] stores the final solution ie, the path of the salesman.
int final_path[N + 1];

// visited[] keeps track of the already visited nodes in a particular path
bool visited[N];

// Stores the final minimum weight of shortest tour.
```

```
int final_res = INT_MAX;

// Function to copy temporary solution to the final solution
void copyToFinal(int curr_path[]) {
	for (int i = 0; i < N; i++)
		final_path[i] = curr_path[i];
	final_path[N] = curr_path[0];
}

// Function to find the minimum edge cost having an end at the vertex i
int firstMin(int adj[N][N], int i) {
	int min = INT_MAX;
	for (int k = 0; k < N; k++)
		if (adj[i][k] < min && i != k)
			min = adj[i][k];
	return min;
}

// function to find the second minimum edge cost having an end at the vertex i
int secondMin(int adj[N][N], int i) {
	int first = INT_MAX, second = INT_MAX;
	for (int j = 0; j < N; j++) {
		if (i == j)
			continue;
		if (adj[i][j] <= first) {
			second = first;
			first = adj[i][j];
		}
		else if (adj[i][j] <= second &&
				adj[i][j] != first)
			second = adj[i][j];
	}
	return second;
}

// function that takes as arguments:
// curr_bound -> lower bound of the root node
// curr_weight-> stores the weight of the path so far
// level-> current level while moving in the search
//		space tree
// curr_path[] -> where the solution is being stored which
//			would later be copied to final_path[]
void TSPRec(int adj[N][N], int curr_bound, int curr_weight,
			int level, int curr_path[]) {
	// base case is when we have reached level N which
	// means we have covered all the nodes once
	if (level == N) {
		// check if there is an edge from last vertex in path back to the first vertex
```

```
            if (adj[curr_path[level-1]][curr_path[0]] != 0) {
                // curr_res has the total weight of the solution we got
                int curr_res = curr_weight +
                            adj[curr_path[level-1]][curr_path[0]];
                // Update final result and final path if current result is better.
                if (curr_res < final_res) {
                    copyToFinal(curr_path);
                    final_res = curr_res;
                }
            }
            return;
        }
        // for any other level iterate for all vertices to build the search space tree recursively
        for (int i = 0; i < N; i++) {
            // Consider next vertex if it is not same (diagonal entry in adjacency matrix
and not visited already)
            if (adj[curr_path[level-1]][i] != 0 && visited[i] == false) {
                int temp = curr_bound;
                curr_weight += adj[curr_path[level-1]][i];
                // different computation of curr_bound for level 2 from the other levels
                if (level == 1)
                curr_bound -= ((firstMin(adj, curr_path[level-1]) + firstMin(adj, i))/2);
                else
                curr_bound -= ((secondMin(adj, curr_path[level-1]) + firstMin(adj,
i))/2);
                // curr_bound + curr_weight is the actual lower bound for the node that
we have arrived on
                // If current lower bound < final_res, we need to explore the node
further
                if (curr_bound + curr_weight < final_res) {
                    curr_path[level] = i;
                    visited[i] = true;
                    // call TSPRec for the next level
                    TSPRec(adj, curr_bound, curr_weight, level + 1, curr_path);
                }
                // Else we have to prune the node by resetting all changes to
curr_weight and curr_bound
                curr_weight -= adj[curr_path[level-1]][i];
                curr_bound = temp;
                // Also reset the visited array
                memset(visited, false, sizeof(visited));
                for (int j = 0; j <= level - 1; j++)
                    visited[curr_path[j]] = true;
            }
        }
}

// This function sets up final_path[]
```

```
void TSP(int adj[N][N]) {
	int curr_path[N+1];

	// Calculate initial lower bound for the root node using the formula 1/2 * (sum of first
min + second min) for all edges.
	// Also initialize the curr_path and visited array
	int curr_bound = 0;
	memset(curr_path, -1, sizeof(curr_path));
	memset(visited, 0, sizeof(curr_path));
	// Compute initial bound
	for (int i=0; i<N; i++)
		curr_bound += (firstMin(adj, i) + secondMin(adj, i));
	// Rounding off the lower bound to an integer
	curr_bound = (curr_bound&1)? curr_bound/2 + 1 : curr_bound/2;
	// We start at vertex 1 so the first vertex in curr_path[] is 0
	visited[0] = true;
	curr_path[0] = 0;
	// Call to TSPRec for curr_weight equal to 0 and level 1
	TSPRec(adj, curr_bound, 0, 1, curr_path);
}

int main() {
	//Adjacency matrix for the given graph
	int adj[N][N] = { {0, 10, 15, 20},
		{10, 0, 35, 25},
		{15, 35, 0, 30},
		{20, 25, 30, 0}
	};
	TSP(adj);
	printf("Minimum cost : %d\n", final_res);
	printf("Path Taken : ");
	for (int i = 0; i <= N; i++)
		printf("%d ", final_path[i]);
	return 0;
}
```

Output:

```
Minimum cost : 80
Path Taken : 0 1 3 2 0
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment
Department of Computer Science & Engineering
Amity University, Noida (UP)

Program	B. Tech CSE	Course Name	Analysis and Design of Algorithm
Course Code	[CSE 303]	Semester	5
Student Name	Saumyaa Garg	Enrollment No.	A2305222336

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		