IMPACT College of Engineering & Applied Sciences

ICEAS

LAB MANUAL

AY: 2024 – 25 (EVEN SEMESTER)

SEMESTER: 4

DEPT(s): CSE-ALLIED BRANCHES

SUBJECT NAME: ANALYSIS AND DESIGN OF ALGORITHMS

SUBJECT CODE : BCDL404

Syllabus

Analysis & Design of Algorithms Lab		Semester	4
Course Code	BCDL404	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0:0:2:0	SEE Marks	50
Credits	01	Exam Hours	2
Examination type (SEE)	Practical		

Course objectives:

To design and implement various algorithms in C/C++ programming using suitable development tools to address different computational challenges.

To apply diverse design strategies for effective problem-solving.

To Measure and compare the performance of different algorithms to determine their efficiency and suitability for specific tasks.

Sl.No	Experiments
1	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.
2	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.
3	Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm. Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.
4	Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.
5	Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.
6	Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.
7	Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.
8	Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.
9	Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
10	Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

11	Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
12	Design and implement C/C++ Program for N Queen's problem using Backtracking.

Course outcomes (Course Skill Set):

At the end of the course the student will be able to:

Develop programs to solve computational problems using suitable algorithm design strategy.

Compare algorithm design strategies by developing equivalent programs and observing running times for analysis (Empirical).

Make use of suitable integrated development tools to develop programs

Choose appropriate algorithm design techniques to develop solution to the computational and complex problems. Demonstrate and present the development of program, its execution and running time(s) and record the results/inferences.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

Continuous Internal Evaluation (CIE):

CIE marks for the practical course are **50 Marks**.

The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.

Each experiment is to be evaluated for conduction with an observation sheet and record write-up. Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session.

Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks.

Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks).

Weightage to be given for neatness and submission of record/write-up on time.

Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus.

In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce.

The suitable rubrics can be designed to evaluate each student's performance and learning ability.

The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).

The Sum of scaled-down marks scored in the report write-up/journal and marks of a test is the total CIE marks scored by the student.

Semester End Evaluation (SEE):

SEE marks for the practical course are 50 Marks.

1. Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.

```
// Kruskal's algorithm in C

#include <stdio.h>

#define MAX 30

typedef struct edge {
  int u, v, w;
} edge;

typedef struct edge_list {
  edge data[MAX];
  int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

// Applying Krushkal Algo
void kruskalAlgo() {
  int belongs[MAX], i, j, cno1, cno2;
  elist.n = 0;

  for (i = 1; i < n; i++)
    for (j = 0; j < i; j++) {
      if (Graph[i][j] != 0) {
        elist.data[elist.n].u = i;
        elist.data[elist.n].v = j;
        elist.data[elist.n].w = Graph[i][j];
        elist.n++;
      }
    }
  sort();

  for (i = 0; i < n; i++)
    belongs[i] = i;

  spanlist.n = 0;

  for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2) {
      spanlist.data[spanlist.n] = elist.data[i];
      spanlist.n = spanlist.n + 1;
      applyUnion(belongs, cno1, cno2);
    }
  }
}

int find(int belongs[], int vertexno) {
  return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
  int i;

  for (i = 0; i < n; i++)
    if (belongs[i] == c2)
      belongs[i] = c1;
}

// Sorting algo
void sort() {
  int i, j;
  edge temp;

  for (i = 1; i < elist.n; i++)
    for (j = 0; j < elist.n - 1; j++)
      if (elist.data[j].w > elist.data[j + 1].w) {
        temp = elist.data[j];
        elist.data[j] = elist.data[j + 1];
        elist.data[j + 1] = temp;
      }
}
```

```
// Printing the result
void print() {
  int i, cost = 0;

  for (i = 0; i < spanlist.n; i++) {
    printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
    cost = cost + spanlist.data[i].w;
  }

  printf("\nSpanning tree cost: %d", cost);
}

int main() {
  int i, j, total_cost;

  n = 6;

  Graph[0][0] = 0;
  Graph[0][1] = 4;
  Graph[0][2] = 4;
  Graph[0][3] = 0;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 4;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 0;
  Graph[1][6] = 0;

  Graph[2][0] = 4;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 3;
  Graph[2][4] = 4;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 0;
  Graph[3][1] = 0;
  Graph[3][2] = 3;
  Graph[3][3] = 0;
  Graph[3][4] = 3;
  Graph[3][5] = 0;
  Graph[3][6] = 0;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 4;
  Graph[4][3] = 3;
  Graph[4][4] = 0;
  Graph[4][5] = 0;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 0;
  Graph[5][2] = 2;
  Graph[5][3] = 0;
  Graph[5][4] = 3;
  Graph[5][5] = 0;
  Graph[5][6] = 0;

  kruskalAlgo();
  print();
}
```

OUTPUT

```
2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14

=== Code Execution Successful ===
```

Explanation

1.Header File and Macro Definitions:

```
#include <stdio.h>
#define MAX 30
```

- The code includes the standard input-output library (stdio.h) for input/output operations.
- A macro MAX is defined to represent the maximum number of vertices or edges in the graph.

2. Structures for Edge and Edge List:

```
typedef struct edge {
  int u, v, w;
} edge;

typedef struct edge_list {
  edge data[MAX];
  int n;
} edge_list;
```

- The structure edge represents an edge in the graph with attributes u (source vertex), v (destination vertex), and w (weight).
- The structure edge_list is used to store a list of edges (data) and the number of edges (n) in the list.

3. Global Variables and Function Prototypes:

```
edge_list elist;
int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();
```

- elist is used to store the list of all edges in the graph.
- Graph is a 2D array representing the graph's adjacency matrix.
- n represents the number of vertices in the graph.
- spanlist stores the edges of the Minimum Spanning Tree (MST).
- Function prototypes for the Kruskal's algorithm, helper functions, sorting function, and print function are declared.

4. Kruskal's Algorithm Implementation:

```
void kruskalAlgo() {
  int belongs[MAX], i, j, cno1, cno2;
  elist.n = 0;

  // Constructing a list of all edges in the graph
  for (i = 1; i < n; i++)
    for (j = 0; j < i; j++) {
      if (Graph[i][j] != 0) {
        elist.data[elist.n].u = i;
        elist.data[elist.n].v = j;
        elist.data[elist.n].w = Graph[i][j];
        elist.n++;
      }
    }
  sort(); // Sort the edges based on their weights

  // Initialize an array to keep track of the sets each vertex belongs to
  for (i = 0; i < n; i++)
    belongs[i] = i;

  spanlist.n = 0;

  // Apply Kruskal's algorithm
  for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2) {
      spanlist.data[spanlist.n] = elist.data[i];
      spanlist.n++;
      applyUnion(belongs, cno1, cno2);
    }
  }
}
```

- The kruskalAlgo() function implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) of the graph.
- It constructs a list of all edges in the graph, sorts them based on their weights, and then applies Kruskal's algorithm to select the edges for the MST.
- The belongs[] array keeps track of which set each vertex belongs to.
- The applyUnion() function is called to merge two sets when adding an edge to the MST.

5. Helper Functions: find() and applyUnion() :

```
  int find(int belongs[], int vertexno) {
    return (belongs[vertexno]);
  }

  void applyUnion(int belongs[], int c1, int c2) {
    int i;

    for (i = 0; i < n; i++)
      if (belongs[i] == c2)
        belongs[i] = c1;
  }
```

- The find() function returns the set to which a given vertex belongs.
- The applyUnion() function merges two sets by updating the belongs[] array.

6. Sorting Function sort() :

```
  void sort() {
    int i, j;
    edge temp;

    for (i = 1; i < elist.n; i++)
      for (j = 0; j < elist.n - 1; j++)
        if (elist.data[j].w > elist.data[j + 1].w) {
          temp = elist.data[j];
          elist.data[j] = elist.data[j + 1];
          elist.data[j + 1] = temp;
        }
  }
```

- The sort() function sorts the edges in the elist based on their weights in non-decreasing order using bubble sort.

7. Printing the Result:

```
  void print() {
    int i, cost = 0;

    for (i = 0; i < spanlist.n; i++) {
      printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
      cost = cost + spanlist.data[i].w;
    }

    printf("\nSpanning tree cost: %d", cost);
  }
```

- The print() function prints the edges of the Minimum Spanning Tree (MST) along with their weights.
- It also calculates and prints the total cost of the MST.

8. Main Function:

```
  int main() {
    // Graph initialization
    n = 6;
    // Graph adjacency matrix initialization
    // (omitted for brevity)
    kruskalAlgo(); // Apply Kruskal's algorithm
    print(); // Print the MST
  }
```

- The main() function initializes the graph and calls the kruskalAlgo() function to find the MST using Kruskal's algorithm.
- Finally, it calls the print() function to print the MST and its total cost.

This code implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a given graph. It constructs a list of edges, sorts them by weight, and then selects edges one by one while avoiding cycles to form the MST.

2. Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm

```
// Prim's Algorithm in C

#include<stdio.h>
#include<stdbool.h>

#define INF 9999999

// number of vertices in graph
#define V 5

// create a 2d array of size 5x5
//for adjacency matrix to represent graph
int G[V][V] = {
  {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}};

int main() {
  int no_edge;  // number of edge

  // create a array to track selected vertex
  // selected will become true otherwise false
  int selected[V];

  // set selected false initially
  memset(selected, false, sizeof(selected));

  // set number of edge to 0
  no_edge = 0;

  // the number of egde in minimum spanning tree will be
  // always less than (V -1), where V is number of vertices in
  //graph

  // choose 0th vertex and make it true
  selected[0] = true;

  int x;  //  row number
  int y;  //  col number
  // print for edge and weight
  printf("Edge : Weight\n");

  while (no_edge < V - 1) {
    //For every vertex in the set S, find the all adjacent vertices
    // , calculate the distance from the vertex selected at step 1.
    // if the vertex is already in the set S, discard it otherwise
    //choose another vertex nearest to selected vertex  at step 1.

    int min = INF;
    x = 0;
    y = 0;

    for (int i = 0; i < V; i++) {
      if (selected[i])
{
        for (int j = 0; j < V; j++)
{
          if (!selected[j] && G[i][j]) {  // not in selected and there is an edge
            if (min > G[i][j])
{
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    printf("%d - %d : %d\n", x, y, G[x][y]);
    selected[y] = true;
    no_edge++;
  }

  return 0;
}
```

OUTPUT

```
Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51
```

```
=== Code Execution Successful ===
```

Explanation

1. Header Files and Macros:

```c
#include<stdio.h>
#include<stdbool.h>

#define INF 9999999
#define V 5
```

- The code includes standard input-output library (stdio.h) for input/output operations and a header for handling boolean values (stdbool.h).
- INF is defined as a very large value to represent infinity, used for initialization purposes.
- V is defined as 5, representing the number of vertices in the graph.

2. Graph Representation:

```
int G[V][V] = {
  {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}
};
```

- The graph is represented as a 2D array G[][] , where G[i][j] represents the weight of the edge between vertices i and j .
- Here, G represents a weighted undirected graph with 5 vertices.

3. Main Function:

```
int main() {
  int no_edge;
  int selected[V];
  memset(selected, false, sizeof(selected));
  no_edge = 0;
  selected[0] = true;
  int x, y;
  printf("Edge : Weight\n");
```

- main() function starts execution.
- no_edge is initialized to keep track of the number of edges in the MST.
- selected[] array is initialized to keep track of vertices that are already included in the MST.

- Memory is set to false for all elements of selected[] array using memset() .
- no_edge is set to 0, indicating no edges are selected initially.
- The 0th vertex is marked as selected since the MST starts with this vertex.
- Variables x and y are declared to track the edge being added to the MST.
- The message "Edge : Weight" is printed to indicate the start of printing MST edges.

4. Prim's Algorithm Execution:

```
  while (no_edge < V - 1) {
    int min = INF;
    x = 0;
    y = 0;
    for (int i = 0; i < V; i++) {
      if (selected[i]) {
        for (int j = 0; j < V; j++) {
          if (!selected[j] && G[i][j]) {
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
printf("%d - %d : %d\n", x, y, G[x][y]);
    selected[y] = true;
    no_edge++;
  }
```

- The loop continues until the number of edges in the MST is V - 1 .
- Inside the loop, the minimum weight edge that connects a vertex in the MST to a vertex outside the MST is found.
- For each vertex i that is already in the MST (selected[i] == true), the algorithm checks all vertices j that are not yet in the MST (!selected[j]) and have an edge connecting to i .
- Among these edges, the one with minimum weight (min) is selected.
- The selected edge (x, y) is printed along with its weight.
- Vertex y is marked as selected (selected[y] = true) since it's added to the MST.
- no_edge is incremented to track the progress of edge selection.

5. Conclusion:

```
  return 0;
}
```

- The main() function ends, and the program returns 0 to the operating system, indicating successful execution.

This program effectively implements Prim's algorithm to find the Minimum Spanning Tree (MST) of a given graph. It traverses the graph and selects the minimum weight edges until the MST is formed.

3. a). Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.

a)

```
#include <stdio.h>

#define INF 99999
#define V 4 // Number of vertices in the graph

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

// Floyd Warshall algorithm
void floydWarshall(int graph[][V]) {
  int dist[V][V];

  // Initialize distance matrix
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      dist[i][j] = graph[i][j];

  // Update distance matrix considering all intermediate vertices
  for (int k = 0; k < V; k++) {
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        if (dist[i][k] + dist[k][j] < dist[i][j])
          dist[i][j] = dist[i][k] + dist[k][j];
      }
    }
  }

  // Print the solution
  printSolution(dist);
}

int main() {
   int graph[V][V] = {
      {0, INF, 3, INF},
      {2, 0, INF, INF},
      {INF, 7, 0, 1},
      {6, INF, INF, 0}
   };

   floydWarshall(graph);
   return 0;
}
```

OUTPUT

```
The following matrix shows the shortest distances between every pair of vertices:
0       10      3       4
2       0       5       6
7       7       0       1
6       16      9       0
```

EXPLANATION

1. Header and Definitions:
   - The program includes standard input-output library (stdio.h).
   - It defines INF (infinity) to represent infinite distance and V as the number of vertices in the graph.
2. Function Prototypes:
   - printSolution() : Prints the solution matrix.
   - floydWarshall() : Implementation of Floyd Warshall algorithm to find the shortest paths between all pairs of vertices.
3. Function Definitions:
   - printSolution() : Iterates through the solution matrix and prints the shortest distances between all pairs of vertices. It prints "INF" for unreachable pairs.
   - floydWarshall() : Implements Floyd Warshall algorithm. It initializes the distance matrix with the given graph. Then, it updates the distance matrix by considering all intermediate vertices. Finally, it calls printSolution() to print the solution matrix.
4. Main Function:
   - Initializes the input graph with the provided distances between vertices.
   - Calls floydWarshall() with the input graph to find the shortest paths between all pairs of vertices.

This program efficiently computes the shortest paths between all pairs of vertices in a given graph using Floyd's algorithm.

b). Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.

```
b) #include <stdio.h>

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

// Function to print the transitive closure matrix
void printTransitiveClosure(int closure[][V]) {
  printf("Transitive Closure Matrix:\n");
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      printf("%d ", closure[i][j]);
    }
    printf("\n");
  }
}

// Warshall's algorithm to find the transitive closure
void transitiveClosure(int graph[][V]) {
  int closure[V][V];

  // Initialize the closure matrix with the given graph
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      closure[i][j] = graph[i][j];
    }
  }

  // Update closure matrix by considering all intermediate vertices
  for (int k = 0; k < V; k++) {
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        closure[i][j] = closure[i][j] || (closure[i][k] && closure[k][j]);
      }
    }
  }

  // Print the transitive closure matrix
  printTransitiveClosure(closure);
}
```

```
int main() {
   int graph[V][V] = {
      {1, 1, 0, 1},
      {0, 1, 1, 0},
      {0, 0, 1, 1},
      {0, 0, 0, 1}
   };

   // Find and print the transitive closure
   transitiveClosure(graph);

   return 0;
}
```

OUTPUT

```
/tmp/goOugZR4N3.o
Transitive Closure Matrix:
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
```

EXPLANATION

Header and Definitions:
The program includes standard input-output library (stdio.h).
It defines V as the number of vertices in the graph.
Function Prototypes:
printTransitiveClosure(): Prints the transitive closure matrix.
transitiveClosure(): Implementation of Warshall's algorithm to find the transitive closure.
Function Definitions:
printTransitiveClosure(): Iterates through the transitive closure matrix and prints its elements.
transitiveClosure(): Implements Warshall's algorithm. It initializes the closure matrix with the given graph. Then, it updates the closure matrix by considering all intermediate vertices. Finally, it calls printTransitiveClosure() to print the transitive closure matrix.
Main Function:
Initializes the input graph with the given adjacency matrix representing the directed graph.
Calls transitiveClosure() with the input graph to find the transitive closure.
This program efficiently computes the transitive closure of a directed graph using Warshall's algorithm

4．Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm

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

#define V 6 // Number of vertices in the graph
#define INF 99999 // Infinity value for representing infinite distance

// Function to find the vertex with the minimum distance value, from the set of vertices
// not yet included in the shortest path tree
int minDistance(int dist[], bool sptSet[]) {
   int min = INF, min_index;

   for (int v = 0; v < V; v++) {
      if (sptSet[v] == false && dist[v] <= min) {
         min = dist[v];
         min_index = v;
      }
   }

   return min_index;
}

// Function to print the constructed distance array
void printSolution(int dist[], int src) {
   printf("Vertex   Distance from Source %d\n", src);
   for (int i = 0; i < V; i++)
      printf("%d \t\t %d\n", i, dist[i]);
}

// Function to implement Dijkstra's algorithm for a given graph
void dijkstra(int graph[V][V], int src) {
   int dist[V]; // Array to store the shortest distance from src to i
   bool sptSet[V]; // sptSet[i] will be true if vertex i is included in the shortest path tree

   // Initialize all distances as INFINITE and sptSet[] as false
   for (int i = 0; i < V; i++) {
      dist[i] = INF;
      sptSet[i] = false;
```

```
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not yet processed
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            // Update dist[v] only if it's not in sptSet, there is an edge from u to v,
            // and total weight of path from src to v through u is smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
        }
    }

    // Print the constructed distance array
    printSolution(dist, src);
}

int main() {
    // Given graph in adjacency matrix representation
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0},
        {4, 0, 8, 0, 0, 0},
        {0, 8, 0, 7, 0, 4},
        {0, 0, 7, 0, 9, 14},
        {0, 0, 0, 9, 0, 10},
        {0, 0, 4, 14, 10, 0}
    };

    int src = 0; // Source vertex

    printf("Shortest paths from vertex %d:\n", src);
    dijkstra(graph, src);

    return 0;
```

```
}
```

OUTPUT

```
Shortest paths from vertex 0:
Vertex   Distance from Source 0
0                    0
1                    4
2                    12
3                    19
4                    26
5                    16
```

EXPLANATION

1. Header Files and Definitions:
   - The program includes standard input-output library (stdio.h) and a header for handling boolean values (stdbool.h).
   - V is defined as the number of vertices in the graph.
   - INF is defined as a very large value to represent infinite distance.

2. Function Prototypes:
   - minDistance() : Finds the vertex with the minimum distance value from the set of vertices not yet included in the shortest path tree.
   - printSolution() : Prints the shortest distances from the source vertex to all other vertices.
   - dijkstra() : Implements Dijkstra's algorithm to find the shortest paths from the source vertex to all other vertices.
3. Function Definitions:
   - minDistance() : Finds the vertex with the minimum distance value among the vertices not yet included in the shortest path tree (sptSet[]).
   - printSolution() : Prints the shortest distances from the source vertex to all other vertices.
   - dijkstra() : Implements Dijkstra's algorithm. It initializes dist[] with INF and sptSet[] with false . It then iteratively finds the shortest path for all vertices by selecting the vertex with the minimum distance, marking it as processed, and updating the distances of its adjacent vertices if a shorter path is found.
4. Main Function:
   - Initializes the given graph in adjacency matrix representation.
   - Calls dijkstra() with the graph and the source vertex to find the shortest paths from the source vertex to all other vertices.

This program efficiently computes the shortest paths from a given vertex to all other vertices in a weighted connected graph using Dijkstra's algorithm.

5. Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

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

#define MAX_VERTICES 100

// Stack data structure
typedef struct {
    int arr[MAX_VERTICES];
    int top;
} Stack;

// Initialize stack
void initialize(Stack *s) {
    s->top = -1;
}

// Check if stack is empty
bool isEmpty(Stack *s) {
    return s->top == -1;
}

// Push element onto stack
void push(Stack *s, int data) {
    s->arr[++s->top] = data;
}

// Pop element from stack
int pop(Stack *s) {
    if (!isEmpty(s))
        return s->arr[s->top--];
    else {
        printf("Stack underflow!\n");
        exit(EXIT_FAILURE);
    }
}

// Graph data structure
typedef struct {
    int vertices;
    int   adjMatrix;
} Graph;
```

```
// Create graph with 'vertices' vertices
Graph* createGraph(int vertices) {
   Graph *graph = (Graph*)malloc(sizeof(Graph));
   graph->vertices = vertices;

   graph->adjMatrix = (int  )malloc(vertices * sizeof(int*));
   for (int i = 0; i < vertices; ++i)
      graph->adjMatrix[i] = (int*)calloc(vertices, sizeof(int));

   return graph;
}

// Add edge to graph
void addEdge(Graph *graph, int src, int dest) {
   graph->adjMatrix[src][dest] = 1;
}

// Depth First Search (DFS) traversal
void DFS(Graph *graph, int vertex, bool *visited, Stack *s) {
   visited[vertex] = true;

   for (int i = 0; i < graph->vertices; ++i) {
      if (graph->adjMatrix[vertex][i] && !visited[i])
         DFS(graph, i, visited, s);
   }

   push(s, vertex); // Push the vertex onto the stack after visiting all its adjacent vertices
}

// Topological sort using Depth First Search (DFS)
void topologicalSort(Graph *graph) {
   Stack s;
   initialize(&s);

   bool visited[MAX_VERTICES] = {false};

   for (int i = 0; i < graph->vertices; ++i) {
      if (!visited[i])
         DFS(graph, i, visited, &s);
   }

   // Print topological order
   printf("Topological ordering of vertices: ");
   while (!isEmpty(&s))
      printf("%d ", pop(&s));
   printf("\n");
}

int main() {
   int vertices = 6; // Number of vertices in the graph

   // Create a directed graph
   Graph *graph = createGraph(vertices);

   // Add edges to the graph
   addEdge(graph, 5, 2);
   addEdge(graph, 5, 0);
   addEdge(graph, 4, 0);
   addEdge(graph, 4, 1);
   addEdge(graph, 2, 3);
   addEdge(graph, 3, 1);

   // Obtain and print topological ordering
   topologicalSort(graph);

   return 0;
}
```

OUTPUT

```
Topological ordering of vertices: 5 4 2 3 1 0
```

EXPLANATION

Explanation of the program:

1. Stack Data Structure:
   - We implement a stack data structure to assist in topological ordering.

2. Graph Data Structure:
   - We define a graph data structure to represent the directed graph.

3. Functions:
   - initialize() , isEmpty() , push() , and pop() : Functions to handle the stack operations.
   - createGraph() : Function to create a graph with the specified number of vertices.
   - addEdge() : Function to add a directed edge between two vertices.
   - DFS() : Function to perform Depth First Search traversal recursively. It marks vertices as visited and explores them recursively.
   - topologicalSort() : Function to perform topological sorting using Depth First Search. It iterates through all vertices, performs DFS on unvisited vertices, and pushes the vertices onto the stack after visiting all their adjacent vertices.

4. Main Function:
   - We create a directed graph with a specified number of vertices.
   - We add directed edges to the graph.
   - We call topologicalSort() to obtain the topological ordering of vertices and print it.

This program efficiently computes the topological ordering of vertices in a given directed graph.

6. Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method

```
#include <stdio.h>

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

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

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

   // Return maximum value
   return K[n][W];
}

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

   printf("Maximum value that can be obtained: %d\n", knapSack(W, wt, val, n));
   return 0;
}
```

OUTPUT

```
Maximum value that can be obtained: 220
```

EXPLANATION

Explanation of the program:

1. max() Function:
   - The max() function returns the maximum of two integers.

2. knapSack() Function:
   - The knapSack() function takes four arguments:
     - W : The maximum capacity of the knapsack.
     - wt[] : An array containing weights of items.
     - val[] : An array containing values of items.
     - n : The number of items.
   - It uses dynamic programming to solve the 0/1 Knapsack problem. It builds a table K[][] where K[i][w] represents the maximum value that can be obtained with a knapsack capacity of w and i items.
   - It iterates through all possible combinations of items and knapsack capacities and fills in the table according to the optimal substructure of the problem.
   - The final value at K[n][W] represents the maximum value that can be obtained with the given constraints.

3. main() Function:
   - In the main() function, we initialize values and weights of items and the capacity of the knapsack.
   - We call the knapSack() function with these parameters and print the maximum value that can be obtained.

This program efficiently solves the 0/1 Knapsack problem using Dynamic Programming.

7. Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

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

// Structure to represent items
struct Item {
   int weight;
   int value;
   double density; // Density is value/weight ratio
};

// Function to solve discrete knapsack problem using greedy approximation method
int discreteKnapsack(struct Item items[], int n, int capacity) {
   // Calculate density for each item
   for (int i = 0; i < n; i++) {
      items[i].density = (double)items[i].value / items[i].weight;
   }

   // Sort items based on their density in non-increasing order using insertion sort
   for (int i = 1; i < n; i++) {
      struct Item temp = items[i];
      int j = i - 1;
      while (j >= 0 && items[j].density < temp.density) {
         items[j + 1] = items[j];
         j--;
      }
      items[j + 1] = temp;
   }

   // Fill the knapsack with items greedily
   int totalValue = 0;
   for (int i = 0; i < n && capacity > 0; i++) {
      if (items[i].weight <= capacity) {
         totalValue += items[i].value;
         capacity -= items[i].weight;
      }
   }

   return totalValue;
}
```

```
// Function to solve continuous knapsack problem using greedy approximation method
double continuousKnapsack(struct Item items[], int n, int capacity) {
   // Calculate density for each item
   for (int i = 0; i < n; i++) {
      items[i].density = (double)items[i].value / items[i].weight;
   }

   // Sort items based on their density in non-increasing order using insertion sort
   for (int i = 1; i < n; i++) {
      struct Item temp = items[i];
      int j = i - 1;
      while (j >= 0 && items[j].density < temp.density) {
         items[j + 1] = items[j];
         j--;
      }
      items[j + 1] = temp;
   }

   // Fill the knapsack with items greedily
   double totalValue = 0.0;
   for (int i = 0; i < n && capacity > 0; i++) {
      if (items[i].weight <= capacity) {
         totalValue += items[i].value;
         capacity -= items[i].weight;
      } else {
         totalValue += (capacity * items[i].density);
         capacity = 0;
      }
   }

   return totalValue;
}

int main() {
   // Sample data for discrete knapsack problem
   struct Item discreteItems[] = {{10, 60}, {20, 100}, {30, 120}};
   int n1 = sizeof(discreteItems) / sizeof(discreteItems[0]);
   int discreteCapacity = 50;

   // Sample data for continuous knapsack problem
   struct Item continuousItems[] = {{10, 60}, {20, 100}, {30, 120}};
   int n2 = sizeof(continuousItems) / sizeof(continuousItems[0]);
   int continuousCapacity = 50;

   // Compute and print solutions for discrete and continuous knapsack problems
```

```
    printf("Discrete Knapsack Problem:\n");
    int discreteResult = discreteKnapsack(discreteItems, n1, discreteCapacity);
    printf("Maximum value obtained: %d\n\n", discreteResult);

    printf("Continuous Knapsack Problem:\n");
    double continuousResult = continuousKnapsack(continuousItems, n2, continuousCapacity);
    printf("Maximum value obtained: %.2f\n", continuousResult);

    return 0;
}
```

OUTPUT

```
Discrete Knapsack Problem:
Maximum value obtained: 240

Continuous Knapsack Problem:
Maximum value obtained: 240.00
```

EXPLANATION

1. Item Structure:
   - Represents items with weight and value.

2. discreteKnapsack Function:
   - Solves the discrete knapsack problem using the greedy approximation method.
   - Calculates the density (value/weight ratio) for each item and sorts items based on their density in non-increasing order.
   - Iterates through sorted items and selects items greedily until the capacity is exhausted.

3. continuousKnapsack Function:
   - Solves the continuous knapsack problem using the greedy approximation method.
   - Calculates the density (value/weight ratio) for each item and sorts items based on their density in non-increasing order.
   - Iterates through sorted items and selects items greedily, allowing fractional parts of items to be selected.

4. Main Function:
   - Initializes sample data for both discrete and continuous knapsack problems.
   - Computes and prints solutions for both problems.

8. Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.

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

#define MAX_SIZE 100

// Function to find a subset of a given set whose sum is equal to a given positive integer
void findSubset(int set[], int n, int sum, int subset[], int subSize, int totalSum, int index) {
   if (totalSum == sum) {
      // Print the subset
      printf("Subset found: { ");
      for (int i = 0; i < subSize; i++)
         printf("%d ", subset[i]);
      printf("}\n");
      return;
   }

   if (index >= n || totalSum > sum)
      return;

   // Include the current element in the subset
   subset[subSize] = set[index];
   findSubset(set, n, sum, subset, subSize + 1, totalSum + set[index], index + 1);

   // Exclude the current element and move to the next one
   findSubset(set, n, sum, subset, subSize, totalSum, index + 1);
}

int main() {
   int set[] = {12, 4, 5, 6, 7, 2, 3, 8, 9}; // Given set
   int n = sizeof(set) / sizeof(set[0]); // Number of elements in the set
   int sum = 15; // Given sum
   int subset[MAX_SIZE]; // Array to store the subset
   int subSize = 0; // Size of the subset

   printf("Finding subset(s) with sum %d:\n", sum);
   findSubset(set, n, sum, subset, subSize, 0, 0);

   return 0;
}
```

OUTPUT

```
Finding subset(s) with sum 15:
Subset found: { 12 3 }
Subset found: { 4 5 6 }
Subset found: { 4 6 2 3 }
Subset found: { 4 2 9 }
Subset found: { 4 3 8 }
Subset found: { 5 7 3 }
Subset found: { 5 2 8 }
Subset found: { 6 7 2 }
Subset found: { 6 9 }
Subset found: { 7 8 }
```

EXPLANATION

1. findSubset Function:
   - This function recursively finds a subset of the given set whose sum is equal to the given positive integer.
   - It takes parameters:
     - set[] : The given set of positive integers.
     - n : The number of elements in the set.
     - sum : The target sum to achieve.
     - subset[] : An array to store the elements of the subset.
     - subSize : The current size of the subset.
     - totalSum : The current sum of the elements in the subset.
     - index : The index of the current element being considered.
   - It uses backtracking to explore all possible combinations of elements to form the subset.
   - When the totalSum becomes equal to the given sum , it prints the subset.

2. Main Function:
   - Initializes a sample set of positive integers and a target sum.
   - Calls the findSubset function to find the subset(s) with the given sum.

This program will find and print the subset(s) of the given set whose sum is equal to the given positive integer.

9. Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

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

void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int n, i;
    clock_t start, end;
    double time_used;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    printf("Enter %d integers:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    start = clock();
    selectionSort(arr, n);
    end = clock();

    printf("Sorted array: \n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nTime taken: %f seconds\n", time_used);

    free(arr);

    return 0;
}
```

OUTPUT

```
Enter the number of elements: 4
Enter 4 integers:
5 3 1 9
Sorted array:
1 3 5 9
Time taken: 0.000000 seconds

Process returned 0 (0x0)   execution time : 4.791 s
Press any key to continue.
```

(Second Part)

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


// Function to generate random integers in the range [min, max]
void generateRandomArray(int arr[], int n, int min, int max) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % (max - min + 1) + min;
    }
}

// Function to perform selection sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        // Swap arr[i] with the minimum element
        int temp = arr[i];
        arr[i] = arr[min_index];
        arr[min_index] = temp;
    }
}

int main() {
    FILE *fp;
    fp = fopen("sorting_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(fp, "n,Time taken (ms)\n");

    srand(time(NULL)); // Seed the random number generator

    int max_n = 10000; // Maximum value of n
    int min = 1; // Minimum value of generated integers
    int max = 10000; // Maximum value of generated integers

    clock_t start, end;
    double cpu_time_used;

    for (int n = 1000; n <= max_n; n += 1000) {
        int arr[n];

        // Generate random array
        generateRandomArray(arr, n, min, max);

        // Start the timer
        start = clock();

        // Perform selection sort
        selectionSort(arr, n);

        // Stop the timer
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC * 1000; // Time taken in
milliseconds

        // Print time taken
        printf("Time taken to sort %d elements: %.2f ms\n", n, cpu_time_used);

        // Write to file
        fprintf(fp, "%d,%.2f\n", n, cpu_time_used);
    }

    fclose(fp);

    printf("Data saved to sorting_times.csv\n");

    return 0;
}
```

OUTPUT

```
Time taken to sort 1000 elements: 0.00 ms
Time taken to sort 2000 elements: 15.00 ms
Time taken to sort 3000 elements: 16.00 ms
Time taken to sort 4000 elements: 15.00 ms
Time taken to sort 5000 elements: 47.00 ms
Time taken to sort 6000 elements: 47.00 ms
Time taken to sort 7000 elements: 78.00 ms
Time taken to sort 8000 elements: 78.00 ms
Time taken to sort 9000 elements: 122.00 ms
Time taken to sort 10000 elements: 141.00 ms
Data saved to sorting_times.csv

Process returned 0 (0x0)   execution time : 0.591 s
Press any key to continue.
```

Plot a graph.

10. Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

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

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

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
   int pivot = arr[high]; // Pivot element
   int i = (low - 1); // Index of smaller element

   for (int j = low; j <= high - 1; j++) {
      // If current element is smaller than the pivot
      if (arr[j] < pivot) {
         i++; // Increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}

// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      // Partitioning index
      int pi = partition(arr, low, high);

      // Sort elements before partition and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}

int main() {
   int n, i;
   clock_t start, end;
   double time_used;

   printf("Enter the number of elements: ");
   scanf("%d", &n);

   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL) {
      printf("Memory allocation failed\n");
      return 1;
   }

   printf("Enter %d integers:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   start = clock();
   quickSort(arr, 0, n - 1);
   end = clock();

   printf("Sorted array: \n");
   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }

   time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("\nTime taken: %lf seconds\n", time_used);

   free(arr);

   return 0;
}
```

OUTPUT

```
 Enter the number of elements: 5
Enter 5 integers:
3 6 1 7 9
Sorted array:
```

```
1 3 6 7 9
Time taken: 0.000000 seconds

Process returned 0 (0x0)   execution time : 6.946 s
Press any key to continue.
```

(Second Part)

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

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

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
   int pivot = arr[high]; // Pivot element
   int i = (low - 1); // Index of smaller element

   for (int j = low; j <= high - 1; j++) {
      // If current element is smaller than the pivot
      if (arr[j] < pivot) {
         i++; // Increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}

// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      // Partitioning index
      int pi = partition(arr, low, high);

      // Sort elements before partition and after partition
      quickSort(arr, low, pi - 1);
```

```
        quickSort(arr, pi + 1, high);
    }
}

// Function to generate random integers in the range [min, max]
void generateRandomArray(int arr[], int n, int min, int max) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % (max - min + 1) + min;
    }
}

int main() {
    FILE *fp;
    fp = fopen("sorting_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(fp, "n,Time taken (ms)\n");

    srand(time(NULL)); // Seed the random number generator

    int max_n = 10000; // Maximum value of n
    int min = 1; // Minimum value of generated integers
    int max = 10000; // Maximum value of generated integers

    clock_t start, end;
    double cpu_time_used;

    for (int n = 5000; n <= max_n; n += 500) {
        int arr[n];

        // Generate random array
        generateRandomArray(arr, n, min, max);

        // Start the timer
        start = clock();

        // Perform quick sort
        quickSort(arr, 0, n - 1);

        // Stop the timer
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC * 1000; // Time taken in
milliseconds
```

```
        // Print time taken
        printf("Time taken to sort %d elements: %.2f ms\n", n, cpu_time_used);

        // Write to file
        fprintf(fp, "%d,%.2f\n", n, cpu_time_used);
    }

    fclose(fp);

    printf("Data saved to sorting_times.csv\n");

    return 0;
}
```

OUTPUT

```
Time taken to sort 1000 elements: 1.00 ms
Time taken to sort 2000 elements: 0.00 ms
Time taken to sort 3000 elements: 0.00 ms
Time taken to sort 4000 elements: 1.00 ms
Time taken to sort 5000 elements: 0.00 ms
Time taken to sort 6000 elements: 2.00 ms
Time taken to sort 7000 elements: 0.00 ms
Time taken to sort 8000 elements: 0.00 ms
Time taken to sort 9000 elements: 2.00 ms
Time taken to sort 10000 elements: 2.00 ms
Data saved to sorting_times.csv

Process returned 0 (0x0)   execution time : 0.398 s
Press any key to continue.
```

Draw the graph

11. Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

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

// Merge function to merge two subarrays
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into arr[l..r]
    int i = 0; // Initial index of first subarray
    int j = 0; // Initial index of second subarray
    int k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

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

// Merge Sort function
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Find the middle point
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

int main() {
    int n;
    clock_t start, end;
    double time_used;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    start = clock();
    mergeSort(arr, 0, n - 1);
    end = clock();

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

    time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nTime taken: %lf seconds\n", time_used);

    free(arr);

    return 0;
}
```

OUTPUT

```
Enter the number of elements: 4
Enter 4 integers:
4 8 1 9
Sorted array:
1 4 8 9
Time taken: 0.000000 seconds

Process returned 0 (0x0)   execution time : 25.761 s
Press any key to continue.
```

(Second Part)

*

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

// Merge function to merge two subarrays
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temporary arrays
    int L[n1], R[n2];
    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into arr[l..r]
    int i = 0; // Initial index of first subarray
    int j = 0; // Initial index of second subarray
    int k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

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

// Merge Sort function
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Find the middle point
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

// Function to generate random integers in the range [min, max]
void generateRandomArray(int arr[], int n, int min, int max) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % (max - min + 1) + min;
    }
}

int main() {
    FILE *fp;
    fp = fopen("sorting_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(fp, "n,Time taken (ms)\n");

    srand(time(NULL)); // Seed the random number generator

    int max_n = 10000; // Maximum value of n
    int min = 1; // Minimum value of generated integers
    int max = 10000; // Maximum value of generated integers

    clock_t start, end;
    double cpu_time_used;

    for (int n = 5000; n <= max_n; n += 500) {
        int arr[n];

        // Generate random array
        generateRandomArray(arr, n, min, max);

        // Start the timer
        start = clock();

        // Perform merge sort
        mergeSort(arr, 0, n - 1);

        // Stop the timer
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC * 1000; // Time taken in milliseconds

        // Print time taken
        printf("Time taken to sort %d elements: %.2f ms\n", n, cpu_time_used);

        // Write to file
        fprintf(fp, "%d,%.2f\n", n, cpu_time_used);
    }

    fclose(fp);

    printf("Data saved to sorting_times.csv\n");

    return 0;
}
```

OUTPUT

```
Time taken to sort 1000 elements: 0.00 ms
Time taken to sort 1500 elements: 0.00 ms
Time taken to sort 2000 elements: 0.00 ms
Time taken to sort 2500 elements: 0.00 ms
Time taken to sort 3000 elements: 15.00 ms
Time taken to sort 3500 elements: 0.00 ms
Time taken to sort 4000 elements: 0.00 ms
Time taken to sort 4500 elements: 0.00 ms
Time taken to sort 5000 elements: 0.00 ms
Time taken to sort 5500 elements: 0.00 ms
Time taken to sort 6000 elements: 16.00 ms
Time taken to sort 6500 elements: 0.00 ms
Time taken to sort 7000 elements: 0.00 ms
Time taken to sort 7500 elements: 0.00 ms
Time taken to sort 8000 elements: 0.00 ms
Time taken to sort 8500 elements: 0.00 ms
Time taken to sort 9000 elements: 0.00 ms
Time taken to sort 9500 elements: 21.00 ms
Time taken to sort 10000 elements: 0.00 ms
Data saved to sorting_times.csv

Process returned 0 (0x0)   execution time : 0.147 s
Press any key to continue.
```

Draw graph

12. Design and implement C/C++ Program for N Queen's problem using Backtracking

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

#define N 8 // Define the size of the chessboard

// Function to print the solution
void printSolution(int board[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         printf("%d ", board[i][j]);
      }
      printf("\n");
   }
}

// Function to check if a queen can be placed at board[row][col]
bool isSafe(int board[N][N], int row, int col) {
   int i, j;

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

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

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

   return true;
}

// Function to solve N Queens problem using backtracking
bool solveNQueens(int board[N][N], int col) {
   if (col >= N)
      return true;
```

```
    // Consider this column and try placing this queen in all rows
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1; // Place the queen

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

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

    return false; // If queen can't be placed in any row in this column, return false
}

// Main function to solve the N Queens problem and print the solution
void solveNQueensProblem() {
    int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0} };

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

    printSolution(board);
}

int main() {
    solveNQueensProblem();
    return 0;
}
```

OUTPUT

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
```

```
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```