ANALYSIS & DESIGN OF ALGORITHMS LAB

Subject code: BCSL404 **IA Marks:50**

Hour/week:02 **Total Hours:20**

Course Outcomes:

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

1. Develop programs to solve computational problems using suitable algorithm design strategy.
2. Compare algorithm design strategies by developing equivalent programs and observing running times for analysis (Empirical).
3. Make use of suitable integrated development tools to develop programs.
4. Choose appropriate algorithm design techniques to develop solution to the computational and complex problems.
5. Demonstrate and present the development of program, its execution and running time(s) and record the results/inferences.

Sl No.	List of Experiment	CO's	Page no
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	a. Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm. b. 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.	**CO5**	
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.	**CO5**	
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.	**CO5**	
12	Design and implement C/C++ Program for N Queen's problem using Backtracking.		

CONTENT BEYOND SYLLABUS

Sl No.	List of Experiment	CO's	Page no
1	Write a C program to find the length of the Longest Common Subsequence (LCS) between two given strings.		
2	Design and implement C Program to find he shortest paths from a single source vertex to all other vertices in a weighted graph, even in the presence of negative edge weights using Bellman- Ford Algorithm.		

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

CODE:

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

int comparator(const void* p1, const void* p2)          // Comparator function to use in sorting {
        const int(*x)[3] = p1;
        const int(*y)[3] = p2;
        return (*x)[2] - (*y)[2];
}

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

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



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

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

        }
```

```
}

void kruskalAlgo(int n, int edge[n][3])                    // Function to find the MST
{
        // First we sort the edge array in ascending order
        // so that we can access minimum distances/cost
        qsort(edge, n, sizeof(edge[0]), comparator);
        int parent[n];
        int rank[n];
        makeSet(parent, rank, n);              // Function to initialize parent[] and rank[]
          int minCost = 0;
        printf( "Following are the edges in the constructed MST\n");
        for (int i = 0; i < n; i++)
          {
                int v1 = findParent(parent, edge[i][0]);
                int v2 = findParent(parent, edge[i][1]);
                int wt = edge[i][2];

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

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

        kruskalAlgo(5, edge);
        return 0;
}
```

OUTPUT:

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

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

CODE:

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 5                     // Number of vertices in the graph


// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST

int minKey(int key[], bool mstSet[])
{
        // Initialize min value
        int min = INT_MAX, min_index;                   // Initialize min value
        for (int v = 0; v < V; v++)
                if (mstSet[v] == false && key[v] < min)
                        min = key[v], min_index = v;
        return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]

int printMST(int parent[], int graph[V][V])
{
        printf("Edge \tWeight\n");

                for (int i = 1; i < V; i++)
                printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// 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
        // To represent set of vertices included in MST
        bool mstSet[V];
        for (int i = 0; i < V; i++)                     // Initialize all keys as INFINITE
                key[i] = INT_MAX, mstSet[i] = false;
```

```
        // Always include first 1st vertex in MST.
        // Make key 0 so that this vertex is picked as first
        // vertex.
        key[0] = 0;
        parent[0] = -1;
        for (int count = 0; count < V - 1; count++)              // The MST will have V vertices
          {
                // Pick the minimum key vertex from the
                // set of vertices not yet included in MST
                int u = minKey(key, mstSet);
                mstSet[u] = true;                  // Add the picked vertex to the MST Set


                // 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 m 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];
        }
        printMST(parent, graph);       // print the constructed MST
}

int main()                              // Driver's code
{
        int graph[V][V] = { { 0, 2, 0, 6, 0 },
                            { 2, 0, 3, 8, 5 },
                            { 0, 3, 0, 0, 7 },
                            { 6, 8, 0, 0, 9 },
                            { 0, 5, 7, 9, 0 } };
        primMST(graph);
          return 0;
}
```

OUTPUT:

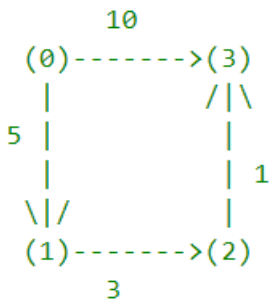
```
Edge          Weight
0 - 1           2
1 - 2           3
0 - 3           6
1 - 4           5
```

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

CODE:

```
#include <stdio.h>
#define V 4                    // Number of vertices in the graph


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

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm

void floydWarshall(int dist[][V])
{
        int i, j, k;


        for (k = 0; k < V; k++)
         {
                for (i = 0; i < V; i++)  // Pick all vertices as source one by one
                    {
                     for (j = 0; j < V; j++)   // Pick all vertices as destination for the above picked source
                        {
                                // If vertex k is on the shortest path from  i to j, then update the value of dist[i][j]
                                if (dist[i][k] + dist[k][j] < dist[i][j])
                                        dist[i][j] = dist[i][k] + dist[k][j];
                        }
                }
        }
        printSolution(dist);                // Print the shortest distance matrix
}


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

```
                else
                        printf("%7d", dist[i][j]);
            }
            printf("\n");
        }
}
int main()              // driver's code
{
        /* Let us create the following weighted graph
```




```
        int graph[V][V] = { { 0, 5, INF, 10 },
                            { INF, 0, 3, INF },
                            { INF, INF, 0, 1 },
                            { INF, INF, INF, 0 } };
        floydWarshall(graph);                   // Function call
        return 0;
}
```

OUTPUT:

The following matrix shows the shortest distances between every pair of vertices

```
0   5   8   9
INF 0   3   4
INF INF 0   1
INF INF INF 0
```

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

CODE:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int max(int, int);

void warshal(int p[10][10], int n)
{
   int i, j, k;
   for (k = 1; k <= n; k++)
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            p[i][j] = max(p[i][j], p[i][k] && p[k][j]);
}
int max(int a, int b)
{
   if (a > b)
      return (a);
   else
      return (b);
}
void main()
{
   int p[10][10] = { 0 }, n, e, u, v, i, j;

   printf("\n Enter the number of vertices:");
   scanf("%d", &n);
   printf("\n Enter the number of edges:");
   scanf("%d", &e);

   for (i = 1; i <= e; i++)
   {
      printf("\n Enter the end vertices of edge %d:", i);
      scanf("%d%d", &u, &v);
      p[u][v] = 1;
   }

   printf("\n Matrix of input data: \n");
   for (i = 1; i <= n; i++)
   {
      for (j = 1; j <= n; j++)
         printf("%d\t", p[i][j]);
      printf("\n");
   }



   warshal(p, n);
```

```
    printf("\n Transitive closure: \n");
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
            printf("%d\t", p[i][j]);
        printf("\n");
    }
    getch();
}
```

OUTPUT:

```
Enter the number of vertices: 5
Enter the number of edges: 11
Enter the end vertices of edge 1: 1 1
Enter the end vertices of edge 2: 1 4
Enter the end vertices of edge 3: 3 2
Enter the end vertices of edge 4: 3 3
Enter the end vertices of edge 5: 3 4
Enter the end vertices of edge 6: 4 2
Enter the end vertices of edge 7: 4 4
Enter the end vertices of edge 8: 5 2
Enter the end vertices of edge 9: 5 3
Enter the end vertices of edge 10: 5 4
Enter the end vertices of edge 11: 5 5

Matrix of input data:
1       0       0       1       0
0       0       0       0       0
0       1       1       1       0
0       1       0       1       0
0       1       1       1       1

Transitive closure:
1       1       0       1       0
0       0       0       0       0
0       1       1       1       0
0       1       0       1       0
0       1       1       1       1
```

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

CODE:

```
#include <stdio.h>
#include <limits.h>
 #define V 9                    // Number of vertices in the graph

// A utility function to find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree

int minDistance(int dist[], int sptSet[])
{
   int min = INT_MAX, min_index;              // Initialize min value
   int v;
   for (v = 0; v < V; v++)
      if (sptSet[v] == 0 && dist[v] <= min)
         min = dist[v], min_index = v;
      return min_index;
}

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


// Funtion that implements Dijkstra's single source shortest path algorithm

void dijkstra(int graph[V][V], int src)
{
   int dist[V];             // The output array.  dist[i] will hold the shortest distance from src to i
   int sptSet[V];                  // sptSet[i] will 1 if vertex i is included in shortest

   // path tree or shortest distance from src to i is finalized  Initialize all distances as INFINITE and       stpSet[] as 0

   int i, count, v;
   for (i = 0; i < V; i++)
      dist[i] = INT_MAX, sptSet[i] = 0;
      dist[src] = 0;                     // Distance of source vertex from itself is always 0
      for (count = 0; count < V - 1; count++)           // Find shortest path for all vertices
      {



      // Pick the minimum distance vertex from the set of vertices not yet processed. u is always equal to src in
```

first iteration.

```
        int u = minDistance(dist, sptSet);
        sptSet[u] = 1;                      // Mark the picked vertex as processed

        // Update dist value of the adjacent vertices of the picked vertex.
        for (v = 0; v < V; v++)
           // Update dist[v] only if is 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] != INT_MAX && dist[u]
                        + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist, V);                         // print the constructed distance array
}


int main()                          // driver program to test above function
{
    int graph[V][V] =  {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                        {4, 0, 8, 0, 0, 0, 0, 11, 0},
                        {0, 8, 0, 7, 0, 4, 0, 0, 2},
                        {0, 0, 7, 0, 9, 14, 0, 0, 0},
                        {0, 0, 0, 9, 0, 10, 0, 0, 0},
                        {0, 0, 4, 0, 10, 0, 2, 0, 0},
                        {0, 0, 0, 14, 0, 2, 0, 1, 6},
                        {8, 11, 0, 0, 0, 0, 1, 0, 7},
                        {0, 0, 2, 0, 0, 0, 6, 7, 0};
    dijkstra(graph, 0);
    return 0;
}
```

OUTPUT:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

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

CODE:

```
#include<stdio.h>

const int MAX = 10;
void fnTopological(int a[MAX][MAX], int n);
int main(void)
{
   int a[MAX][MAX],n;
   int i,j;

   printf("Topological Sorting Algorithm -\n");
   printf("\nEnter the number of vertices : ");
    scanf("%d",&n);

   printf("Enter the adjacency matrix -\n");
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
         scanf("%d",&a[i][j]);

   fnTopological(a,n);
   printf("\n");
   return 0;
}

void fnTopological(int a[MAX][MAX], int n)
{
   int in[MAX], out[MAX], stack[MAX], top=-1;
   int i,j,k=0;

   for (i=0;i<n;i++)
   {
      in[i] = 0;
      for (j=0; j<n; j++)
         if (a[j][i] == 1)
            in[i]++;
   }

   while(1)
   {
      for (i=0;i<n;i++)
      {
         if (in[i] == 0)
         {
            stack[++top] = i;
            in[i] = -1;
         }
      }
```

```
        if (top == -1)
            break;

        out[k] = stack[top--];

        for (i=0;i<n;i++)
        {
            if (a[out[k]][i] == 1)
                in[i]--;
        }
        k++;
    }

    printf("Topological Sorting (JOB SEQUENCE) is:- \n");
    for (i=0;i<k;i++)
        printf("%d ",out[i] + 1);
}
```

OUTPUT:

Input Graph : 5 vertices 0 0 1 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0

Topological Sorting (JOB SEQUENCE) is:- 2 1 3 4 5

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

CODE:

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

int knapSackRec(int W, int wt[], int val[], int index, int** dp)    // Returns the value of maximum profi
{
   if (index < 0)                              // base condition
      return 0;
   if (dp[index][W] != -1)
      return dp[index][W];

   if (wt[index] > W)
   {
      // Store the value of function call stack in table before return
      dp[index][W] = knapSackRec(W, wt, val, index - 1, dp);
      return dp[index][W];
   }
   else {
      // Store value in a table before return
      dp[index][W] = max(val[index]
                  + knapSackRec(W - wt[index], wt, val,
                              index - 1, dp),
               knapSackRec(W, wt, val, index - 1, dp));

      // Return value of table after storing
      return dp[index][W];
   }
}

int knapSack(int W, int wt[], int val[], int n)
{
   // double pointer to declare the  table dynamically
   int** dp;
   dp = new int*[n];

   for (int i = 0; i < n; i++)                              // loop to create the table dynamically
      dp[i] = new int[W + 1];

   for (int i = 0; i < n; i++)                             // loop to initially filled the table with -1
      for (int j = 0; j < W + 1; j++)
         dp[i][j] = -1;
   return knapSackRec(W, wt, val, n - 1, dp);
}
```

```
int main()                              // Driver Code
```

```
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << knapSack(W, weight, profit, n);
    return 0;
}
```

OUTPUT:
220

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

CODE:

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

// Structure to represent items
struct Item
{
   int value;
   int weight;
};

// Comparison function for qsort

int compare(const void *a, const void *b)
{
   struct Item *item1 = (struct Item *)a;
   struct Item *item2 = (struct Item *)b;
   double ratio1 = (double)item1->value / item1->weight;
   double ratio2 = (double)item2->value / item2->weight;
   return (ratio2 > ratio1) - (ratio2 < ratio1);
}

// Function to solve the discrete Knapsack problem using greedy approximation

void discreteKnapsack(struct Item items[], int n, int capacity)
{
   // Sort items based on value-to-weight ratio in non-increasing order
   qsort(items, n, sizeof(struct Item), compare);

   int currentWeight = 0;
   double totalValue = 0.0;

   // Iterate through items and add them to the knapsack
   for (int i = 0; i < n; ++i)
   {

      if (currentWeight + items[i].weight <= capacity)
      {
         currentWeight += items[i].weight;
         totalValue += items[i].value;
      }

     else
      {
         int remainingCapacity = capacity - currentWeight;
         totalValue += items[i].value * ((double)remainingCapacity / items[i].weight);
         break;
      }
   }

   // Print the total value obtained
   printf("Total value obtained in discrete Knapsack: %.2f\n", totalValue);
```

```
}

// Function to solve the continuous Knapsack problem using greedy approximation

void continuousKnapsack(struct Item items[], int n, int capacity)
{
   // Sort items based on value-to-weight ratio in non-increasing order
   qsort(items, n, sizeof(struct Item), compare);

   double totalValue = 0.0;

   // Iterate through items and add them to the knapsack fractionally
   for (int i = 0; i < n; ++i)
   {
      if (items[i].weight <= capacity)
      {
         totalValue += items[i].value;
         capacity -= items[i].weight;
      }
      else
      {
         totalValue += items[i].value * ((double)capacity / items[i].weight);
         break;
      }
   }

   // Print the total value obtained
   printf("Total value obtained in continuous Knapsack: %.2f\n", totalValue);
}


int main()
{
   // Example items for testing

   struct Item items[] = {{60, 10}, {100, 20}, {120, 30}};
   int n = sizeof(items) / sizeof(items[0]);
   int capacity = 50;

   // Solve discrete Knapsack problem
   discreteKnapsack(items, n, capacity);

   // Solve continuous Knapsack problem
   continuousKnapsack(items, n, capacity);

   return 0;
}
```

OUTPUT:

Total value obtained in discrete Knapsack: 240.00
Total value obtained in continuous Knapsack: 280.00

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.

CODE:

```
#include<stdio.h>
#include<conio.h>
int s[10],d,n,set[10],count=0;

void display(int);
int flag = 0;

void main()
{
        int subset(int,int);
        int i;

        printf("ENTER THE NUMBER OF THE ELEMENTS IN THE SET : ");
        scanf("%d",&n);
        printf("ENTER THE SET OF VALUES : ");

        for(i=0;i<n;i++)
          scanf("%d",&s[i]);

        printf("ENTER THE SUM : ");
        scanf("%d",&d);

        printf("THE PROGRAM OUTPUT IS: ");
        subset(0,0);

        if(flag == 0)
        printf("There is no solution");
        }
        int subset(int sum,int i)
        {
                if(sum == d)
                {
                        flag = 1;
                        display(count);
                        return;
                }
        if(sum>d || i>=n)return;
        else
        {
                set[count]=s[i];
                count++;
                subset(sum+s[i],i+1);
                count--;
                subset(sum,i+1);
        }


        }
        void display(int count)
        {
```

```
        int i;
        printf("\t{");
        for(i=0;i<count;i++)
        printf("%d,",set[i]);
        printf("}");
}
```

OUTPUT:

ENTER THE NUMBER OF ELEMETS IN THE SET: 5
ENTER THE SET OF VALUES: 6 5 4 3 2 1
ENTER THE SUM: 5
THE PROGRAM OUTPUT IS : {4,1} {3,2}

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.

CODE:

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

// Function to perform Selection Sort

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

      // Swap the found minimum element with the first element

      int temp = arr[minIndex];
      arr[minIndex] = arr[i];
      arr[i] = temp;
   }
}


int main()
{
   // Open file to store time complexity data

   FILE *timeData;
   timeData = fopen("time_complexity_data.txt", "w");
   if (timeData == NULL) {
      printf("Error opening file.\n");
      return 1;
   }

   // Varying values of n
   int nValues[] = {1000, 2000, 3000, 4000, 5000};
   int numNValues = sizeof(nValues) / sizeof(nValues[0]);



   // Perform sorting and compute time complexity for each n
   for (int i = 0; i < numNValues; ++i)
   {
      int n = nValues[i];
```

```
        int arr[n];

        // Generate random integers

        srand(time(NULL));
        for (int j = 0; j < n; ++j)
        {
            arr[j] = rand() % 1000; // Range of random numbers: 0 to 999
        }

        // Record start time
        clock_t start = clock();

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

        // Record end time
        clock_t end = clock();

        // Calculate time taken for sorting
        double timeTaken = ((double) (end - start)) / CLOCKS_PER_SEC;

        // Print time taken and write to file
        printf("Time taken to sort for n = %d: %.6f seconds\n", n, timeTaken);
        fprintf(timeData, "%d %.6f\n", n, timeTaken);
    }

    // Close file
    fclose(timeData);

    return 0;
}
```


To plot a graph of the time taken versus n, you can use plotting libraries like Matplotlib in Python. Below is an example Python code to read the data from the file and plot the graph:

```
import matplotlib.pyplot as plt

# Read time complexity data from file
with open("time_complexity_data.txt", "r") as file:
    data = file.readlines()

n_values = []
time_taken = []
for line in data:
    n, time = map(float, line.split())
    n_values.append(n)
    time_taken.append(time)

# Plot graph
plt.plot(n_values, time_taken, marker='o')
plt.xlabel('n (Number of elements)')
plt.ylabel('Time taken (seconds)')
plt.title('Time Complexity of Selection Sort')
plt.grid(True)
plt.show()
```

OUTPUT:

The output of the C program will display the time taken to sort for each value of n, as well as record this data in a file named "time_complexity_data.txt".

```
Time taken to sort for n = 1000: 0.000451 seconds
Time taken to sort for n = 2000: 0.001441 seconds
Time taken to sort for n = 3000: 0.003301 seconds
Time taken to sort for n = 4000: 0.005901 seconds
Time taken to sort for n = 5000: 0.010507 seconds
```

This output indicates the time taken to sort for each value of n, measured in seconds. Additionally, the data will be written to the "time_complexity_data.txt" file in the format:

```
1000    0.000451
2000    0.001441
3000    0.003301
4000    0.005901
5000    0.010507
```

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.

CODE:

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

// Function to partition the array
int partition(int arr[], int low, int high)
{
   int pivot = arr[high];
   int i = low - 1;
   for (int j = low; j <= high - 1; j++)
   {
      if (arr[j] < pivot)
      {
         i++;
         int temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
      }
   }

   int temp = arr[i + 1];
   arr[i + 1] = arr[high];
   arr[high] = temp;
   return (i + 1);
}

// Function to perform Quick Sort
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);
   }
}




int main()
{
   // Open file to store time complexity data

   FILE *timeData;
```

```
timeData = fopen("time_complexity_data.txt", "w");
if (timeData == NULL)
{
   printf("Error opening file.\n");
   return 1;
}

// Varying values of n

int nValues[] = {1000, 2000, 3000, 4000, 5000};
int numNValues = sizeof(nValues) / sizeof(nValues[0]);

// Perform sorting and compute time complexity for each n
for (int i = 0; i < numNValues; ++i)
{
   int n = nValues[i];
   int arr[n];

   // Generate random integers

   srand(time(NULL));
   for (int j = 0; j < n; ++j)
   {
      arr[j] = rand() % 1000; // Range of random numbers: 0 to 999
   }

   // Record start time
   clock_t start = clock();

   // Perform Quick Sort
   quickSort(arr, 0, n - 1);

   // Record end time
   clock_t end = clock();

   // Calculate time taken for sorting
   double timeTaken = ((double) (end - start)) / CLOCKS_PER_SEC;

   // Print time taken and write to file
   printf("Time taken to sort for n = %d: %.6f seconds\n", n, timeTaken);
   fprintf(timeData, "%d %.6f\n", n, timeTaken);
}

// Close file
fclose(timeData);

return 0;
```

To plot a graph of the time taken versus n, you can use plotting libraries like Matplotlib in Python. Below is an example Python code to read the data from the file and plot the graph:

```
import matplotlib.pyplot as plt

# Read time complexity data from file
with open("time_complexity_data.txt", "r") as file:
    data = file.readlines()

n_values = []
time_taken = []
for line in data:
    n, time = map(float, line.split())
    n_values.append(n)
    time_taken.append(time)

# Plot graph
plt.plot(n_values, time_taken, marker='o')
plt.xlabel('n (Number of elements)')
plt.ylabel('Time taken (seconds)')
plt.title('Time Complexity of Selection Sort')
plt.grid(True)
plt.show()
```

OUTPUT:

```
1000    0.000006
2000    0.000012
3000    0.000019
4000    0.000026
5000    0.000033
```


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.

CODE:

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

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

    int L[n1], R[n2];

    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];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }

        else
        {
            arr[k] = R[j];
            j++;
        }

        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }


    while (j < n2)
    {
        arr[k] = R[j];
```

```
        j++;
        k++;
    }
}

// Function to perform Merge Sort
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int main()
{
    // Open file to store time complexity data

    FILE *timeData;
    timeData = fopen("time_complexity_data.txt", "w");
    if (timeData == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    // Varying values of n

    int nValues[] = {1000, 2000, 3000, 4000, 5000};
    int numNValues = sizeof(nValues) / sizeof(nValues[0]);

    // Perform sorting and compute time complexity for each n
    for (int i = 0; i < numNValues; ++i)
    {
        int n = nValues[i];
        int arr[n];

        // Generate random integer  srand(time(NULL));
        for (int j = 0; j < n; ++j)
        {
            arr[j] = rand() % 1000; // Range of random numbers: 0 to 999
        }




        // Record start time
        clock_t start = clock();
```

```
        // Perform Merge Sort
        mergeSort(arr, 0, n - 1);

        // Record end time
        clock_t end = clock();

        // Calculate time taken for sorting
        double timeTaken = ((double) (end - start)) / CLOCKS_PER_SEC;

        // Print time taken and write to file
        printf("Time taken to sort for n = %d: %.6f seconds\n", n, timeTaken);
        fprintf(timeData, "%d %.6f\n", n, timeTaken);
    }

    // Close file
    fclose(timeData);

    return 0;
}
```

To plot a graph of the time taken versus n, you can use plotting libraries like Matplotlib in Python. Below is an example Python code to read the data from the file and plot the graph:

```
import matplotlib.pyplot as plt

# Read time complexity data from file
with open("time_complexity_data.txt", "r") as file:
    data = file.readlines()

n_values = []
time_taken = []
for line in data:
    n, time = map(float, line.split())
    n_values.append(n)
    time_taken.append(time)

# Plot graph
plt.plot(n_values, time_taken, marker='o')
plt.xlabel('n (Number of elements)')
plt.ylabel('Time taken (seconds)')
plt.title('Time Complexity of Selection Sort')
plt.grid(True)
plt.show()
```

OUTPUT:

1000	0.000006
2000	0.000012
3000	0.000019
4000	0.000026
5000	0.000033

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

CODE:

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

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

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

// A recursive utility function to solve N  Queen problem

bool solveNQUtil(int board[N][N], int col)
{
        // Base case: If all queens are placed then return true
        if (col >= N)
                return true;




// Consider this column and try placing this queen in all rows one by one

        for (int i = 0; i < N; i++)
```

```
        {
                // Check if the queen can be placed on board[i][col]
                if (isSafe(board, i, col))
                {
                        // Place this queen in board[i][col]
                        board[i][col] = 1;
                        // Recur to place rest of the queens
                        if (solveNQUtil(board, col + 1))
                                return true;

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

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

// This function solves the N Queen problem using Backtracking. It mainly uses solveNQUtil() to solve the problem. It returns false if queen cannot be placed, otherwise, return true and prints placement of queens in the form of 1s. Please note that there may be more than one solutions, this function prints one of the feasible solutions.

```
bool solveNQ()
{
        int board[N][N] = { { 0, 0, 0, 0 },
                            { 0, 0, 0, 0 },
                            { 0, 0, 0, 0 },
                            { 0, 0, 0, 0 } };

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

        printSolution(board);
        return true;
}

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

OUTPUT:

```
. . Q .
Q . . .
. Q . .
. . . Q
```


CONTENT BEYOND SYLLABUS

1. Write a C program to find the length of the Longest Common Subsequence (LCS) between two given strings.

CODE:

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

// Function to find the length of the Longest Common Subsequence (LCS)
int lcsLength(char* X, char* Y, int m, int n)
{
    int L[m + 1][n + 1];

    // Build the L[m+1][n+1] table in bottom-up manner
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;
            else
                L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];
        }
    }

    return L[m][n];
}

int main() {
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of Longest Common Subsequence is %d\n", lcsLength(X, Y, m, n));

    return 0;
}
```

OUTPUT:

Length of Longest Common Subsequence is 4

2. Design and implement C Program to find he shortest paths from a single source vertex to all other vertices in a weighted graph, even in the presence of negative edge weights using Bellman- Ford Algorithm.

CODE:

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

// Structure to represent a weighted edge in the graph
struct Edge
{
   int src, dest, weight;
};

// Structure to represent a directed graph with V vertices and E edges
struct Graph
{
   int V, E;
   struct Edge* edge;
};

// Function to create a new graph with V vertices and E edges

struct Graph* createGraph(int V, int E)
{
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->E = E;
   graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
   return graph;
}

// Function to print the distance array

void printDistances(int dist[], int n)
{
   printf("Vertex   Distance from Source\n");
   for (int i = 0; i < n; ++i)
      printf("%d \t\t %d\n", i, dist[i]);
}

// Bellman-Ford algorithm function

void bellmanFord(struct Graph* graph, int src)
{
   int V = graph->V;
   int E = graph->E;
   int dist[V];



   // Initialize distances from source to all vertices as INFINITE
```

```
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Relax all edges |V| - 1 times
    for (int i = 1; i <= V - 1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Check for negative-weight cycles

    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;

        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
        {
            printf("Graph contains negative-weight cycle\n");
            return;
        }
    }

    // Print the distance array
    printDistances(dist, V);
}

// Main function

int main()
{
    int V = 5; // Number of vertices
    int E = 8; // Number of edges
    struct Graph* graph = createGraph(V, E);

    // Add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;



    // Add edge 0-2
```

```
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // Add edge 1-2
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // Add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // Add edge 1-4
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // Add edge 3-2
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // Add edge 3-1
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // Add edge 4-3
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    int src = 0;                                    // Source vertex
    bellmanFord(graph, src);                        // Call Bellman-Ford function

    return 0;
}
```


OUTPUT:

Vertex	Distance from Source
0	**0**
1	**-1**
2	**2**
3	**-2**
4	**1**