

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

PROGRAM:

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

```
#define MAX_EDGES 1000
```

// Structure to represent an edge in the graph

```
typedef struct Edge {
    int src, dest, weight;
} Edge;
```

```
typedef struct Graph {
    int V, E;
    Edge edges[MAX_EDGES];
} Graph;
```

// Structure to represent a subset for union-find

```
typedef struct Subset {
    int parent, rank;
} Subset;
```

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

// Function to find the root of the set the vertex belongs to

```
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}
```

```
}
```

// Function to perform union of two sets

```
void Union(Subset subsets[], int x, int y) {  
    int xroot = find(subsets, x);  
    int yroot = find(subsets, y);
```

// Attach smaller rank tree under root of larger rank tree

```
    if (subsets[xroot].rank < subsets[yroot].rank) {  
        subsets[xroot].parent = yroot;  
    } else if (subsets[xroot].rank > subsets[yroot].rank) {  
        subsets[yroot].parent = xroot;  
    } else {  
        subsets[yroot].parent = xroot;  
        subsets[xroot].rank++;  
    }  
}
```

// Function to compare two edges by weight

```
int compare(const void* a, const void* b) {  
    Edge* a_edge = (Edge*) a;  
    Edge* b_edge = (Edge*) b;  
    return a_edge->weight - b_edge->weight;  
}
```

```
void kruskalMST(Graph* graph) {  
    Edge mst[graph->V];  
    int e = 0, i = 0;
```

//Sort the edges by weight

```
    qsort(graph->edges, graph->E, sizeof(Edge), compare);
```

// Allocate memory for subsets

```
    Subset* subsets = (Subset*) malloc(graph->V * sizeof(Subset));  
    for (int v = 0; v < graph->V; ++v) {  
        subsets[v].parent = v;  
        subsets[v].rank = 0;  
    }
```

```

while (e < graph->V - 1 && i < graph->E) {
    Edge next_edge = graph->edges[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);
    // Check if adding edge creates a cycle
    if (x != y) {
        mst[e++] = next_edge;
        Union(subsets, x, y);
    }
}

printf("Minimum Spanning Tree:\n");
for (i = 0; i < e; ++i) {
    printf("(%d, %d) -> %d\n", mst[i].src, mst[i].dest, mst[i].weight);
}
}

int main() {
    int V, E;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &V, &E);

    Graph* graph = createGraph(V, E);

    printf("Enter edges and their weights:\n");
    for (int i = 0; i < E; ++i) {
        scanf("%d %d %d", &graph->edges[i].src, &graph->edges[i].dest,
&graph->edges[i].weight);
    }

    kruskalMST(graph);

    return 0;
}

```

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

PROGRAM:

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

```
#define V_MAX 100 // Maximum number of vertices
```

```
// Function to find the vertex with the minimum key value, from the
set of vertices not yet included in the MST
```

```
int minKey(int key[], int mstSet[], int V)
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == 0 && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}
```

```
// Function to print the constructed MST stored in parent[]
```

```
void printMST(int parent[], int n, int graph[V_MAX][V_MAX], int V)
{
    printf("Edge  Weight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d   %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_MAX], int V)
{
    int parent[V_MAX]; // Array to store constructed MST
    int key[V_MAX]; // Key values used to pick minimum weight
edge in cut
    int mstSet[V_MAX]; // To represent set of vertices not yet included
in MST

    // Initialize all keys as INFINITE, mstSet[] as 0
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = 0;

    // Always include first 1st vertex in MST. Make key 0 so that this
vertex is picked as the first vertex
    key[0] = 0;
    parent[0] = -1; // First node is always the root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not yet
included in MST
        int u = minKey(key, mstSet, V);

        // Add the picked vertex to the MST set
        mstSet[u] = 1;

        // Update key value and parent index of the adjacent vertices of
the picked vertex
        // Consider only those vertices which are not yet included in the
MST
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // Print the constructed MST
    printMST(parent, V, graph, V);
}

```

```

int main() {
    int V, E;
    printf("Enter the number of vertices and edges: ");
    scanf("%d %d", &V, &E);

    // Create the graph as an adjacency matrix
    int graph[V_MAX][V_MAX];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            graph[i][j] = 0; // Initialize the graph with 0s
        }
    }

    // Prompt the user to enter the source vertex, destination vertex, and
    weight for each edge
    printf("Enter the source vertex, destination vertex, and weight for
    each edge:\n");
    for (int i = 0; i < E; i++)
    {
        int source, dest, weight;
        scanf("%d %d %d", &source, &dest, &weight);
        graph[source][dest] = weight;
        graph[dest][source] = weight; // Since the graph is undirected
    }

    // Print the MST using Prim's algorithm
    primMST(graph, V);

    return 0;
}

```

OUTPUT:

```
student@lenovo-ThinkCentre-M900:~$ gedit 2.c
student@lenovo-ThinkCentre-M900:~$ gcc 2.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of vertices and edges: 5
7
Enter the source vertex, destination vertex, and weight for each edge:
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

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

PROGRAM:

```
#include<stdio.h>

int min(int,int);
void floyds(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++)
                if(i==j)
                    p[i][j]=0; else
                    p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
}
int min(int a,int b) {
```

```

    if(a<b)
        return(a); else
        return(b);
}
void main() {
    int p[10][10],w,n,e,u,v,i,j;

    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    printf("\n Enter the number of edges:\n");
    scanf("%d",&e);
    for (i=1;i<=n;i++) {
        for (j=1;j<=n;j++)
            p[i][j]=999;
    }
    for (i=1;i<=e;i++) {
        printf("\n Enter the end vertices of edge%d with its weight\n",i);
        scanf("%d%d%d",&u,&v,&w);
        p[u][v]=w;
    }
    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");
    }
    floyds(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");
    }
    printf("\n The shortest paths are:\n");
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++) {
            if(i!=j)
                printf("\n <%d,%d>=%d",i,j,p[i][j]);

```


}

}

OUTPUT:

```
student@lenovo-ThinkCentre-M900:~$ gcc 3a.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
```

Enter the number of vertices:4

Enter the number of edges:
5

Enter the end vertices of edge1 with its weight
1 3 3

Enter the end vertices of edge2 with its weight
2 1 2

Enter the end vertices of edge3 with its weight
3 2 7

Enter the end vertices of edge4 with its weight
3 4 1

Enter the end vertices of edge5 with its weight
4 1 6

Matrix of input data:

999	999	3	999
2	999	999	999
999	7	999	1
6	999	999	999

Transitive closure:

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

The shortest paths are:

<1,2>=10
<1,3>=3
<1,4>=4
<2,1>=2
<2,3>=5
<2,4>=6
<3,1>=7

$\langle 3,2 \rangle = 7$
 $\langle 3,4 \rangle = 1$
 $\langle 4,1 \rangle = 6$
 $\langle 4,2 \rangle = 16$

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

PROGRAM:

```
#include<stdio.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");

}

}

```

OUTPUT:

```

student@lenovo-ThinkCentre-M900:~$ gedit 3b.c
student@lenovo-ThinkCentre-M900:~$ gcc 3b.c
student@lenovo-ThinkCentre-M900:~$ ./a.out

```

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

PROGRAM:

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

```
#define MAX_VERTICES 10 // Maximum number of vertices
#define INF INT_MAX
```

// A 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 V) {
    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;
}
```

// A utility function to print the constructed distance array

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

```
}
```

```
// Dijkstra's algorithm for adjacency matrix representation of the graph
```

```
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int  
src, int V)
```

```
{
```

```
    int dist[MAX_VERTICES];          // The output array. dist[i] will  
hold the shortest distance from src to i
```

```
    bool sptSet[MAX_VERTICES];      // 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;
```

```
dist[src] = 0;
```

```
// Find shortest path for all vertices
```

```
for (int count = 0; count < V - 1; count++) {
```

```
    int u = minDistance(dist, sptSet, V);
```

```
    sptSet[u] = true;
```

```
    for (int v = 0; v < V; v++)
```

```
        if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] +  
graph[u][v] < dist[v])
```

```
            dist[v] = dist[u] + graph[u][v];
```

```
    }
```

```
    printSolution(dist, V);
```

```
}
```

```
// Driver code
```

```
int main() {
```

```
    int V, E;
```

```
    printf("Enter the number of vertices: ");
```

```
    scanf("%d", &V);
```

```
    printf("Enter the number of edges: ");
```

```
    scanf("%d", &E);
```

```
    int graph[MAX_VERTICES][MAX_VERTICES] = {{0}};
```

```

    printf("Enter the source vertex, destination vertex, and weight for
each edge:\n");
    for (int i = 0; i < E; i++) {
        int source, dest, weight;
        scanf("%d %d %d", &source, &dest, &weight);
        graph[source][dest] = weight;
        graph[dest][source] = weight;    // Assuming undirected graph
    }

    dijkstra(graph, 0, V);
    return 0;
}

```

OUTPUT:

```

[3] - Stopped
student@lenovo-ThinkCentre-M900:~$ gedit 4.c
student@lenovo-ThinkCentre-M900:~$ gcc 4.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of vertices: 5
Enter the number of edges: 7
Enter the source vertex, destination vertex, and weight for each edge:
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9
Vertex          Distance from Source
0                0
1                2
2                5
3                6
4                7

```

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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

int graph[MAX][MAX]; // Adjacency matrix
int visited[MAX];    // Visited array
int stack[MAX];      // Stack to store the topological order
int stackIndex = 0;  // Stack index

void addEdge(int src, int dest) {
    graph[src][dest] = 1;
}

// A recursive function to perform DFS and push vertices to stack
void dfs(int v, int V) {
    visited[v] = 1;
    for (int i = 0; i < V; i++) {
        if (graph[v][i] && !visited[i]) {
            dfs(i, V);
        }
    }

    stack[stackIndex++] = v;
}

// The function to perform Topological Sort
void topologicalSort(int V) {
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            dfs(i, V);
        }
    }
    for (int i = stackIndex - 1; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}
```



```

int main() {
    int V = 6;           // Number of vertices
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            graph[i][j] = 0;    // Initialize graph adjacency matrix to 0
        }
        visited[i] = 0; // Initialize visited array to 0
    }

    addEdge(5, 2);
    addEdge(5, 0);
    addEdge(4, 0);
    addEdge(4, 1);
    addEdge(2, 3);
    addEdge(3, 1);
    printf("Topological sorting of the given graph is: \n");
    topologicalSort(V);
    return 0;
}

```

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

PROGRAM:

```

#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, j;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom-up manner
    for (i = 0; i <= n; i++) {

```

```

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

```

```

// K[n][W] contains the maximum value that can be put in a knapsack of
capacity W
return K[n][W];
}

```

```

int main() {
    int val[100], wt[100]; // Arrays to store values and weights
    int W, n; // Knapsack capacity and number of items
    printf("Enter the number of items: ");
    scanf("%d", &n);

    printf("Enter the values and weights of %d items:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Enter value and weight for item %d: ", i + 1);
        scanf("%d %d", &val[i], &wt[i]);
    }

    printf("Enter the knapsack capacity: ");
    scanf("%d", &W);

    printf("Maximum value that can be obtained: %d\n", knapsack(W, wt,
val, n));

    return 0;
}

```

OUTPUT:

```
student@lenovo-ThinkCentre-M900:~$ gcc 6.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of items: 4
Enter the values and weights of 4 items:
Enter value and weight for item 1: 42 7
Enter value and weight for item 2: 12 3
Enter value and weight for item 3: 40 4
Enter value and weight for item 4: 25 5
Enter the knapsack capacity: 10
Maximum value that can be obtained: 65
```

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

PROGRAM:

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

```
// Structure to represent items
```

```
struct Item {
    int value;
    int weight;
    double ratio; // Value-to-weight ratio for sorting
};
```

```
// Comparison function for sorting items based on ratio in descending order
```

```
int compare(const void *a, const void *b) {
    struct Item *item1 = (struct Item *)a;
    struct Item *item2 = (struct Item *)b;
    double ratio1 = item1->ratio;
    double ratio2 = item2->ratio;
    if (ratio1 > ratio2) return -1;
    else if (ratio1 < ratio2) return 1;
    else return 0;
}
```

```
// Function to solve discrete Knapsack problem
```

```
void discreteKnapsack(struct Item items[], int n, int capacity) {
    int i, j;
```

```

int dp[n + 1][capacity + 1];

// Initialize the DP table
for (i = 0; i <= n; i++) {
    for (j = 0; j <= capacity; j++) {
        if (i == 0 || j == 0)
            dp[i][j] = 0;
        else if (items[i - 1].weight <= j)
            dp[i][j] = (items[i - 1].value + dp[i - 1][j - items[i - 1].weight] >
dp[i - 1][j]) ?
                (items[i - 1].value + dp[i - 1][j - items[i - 1].weight]) :
                dp[i - 1][j];
        else
            dp[i][j] = dp[i - 1][j];
    }
}

printf("Total value obtained for discrete knapsack: %d\n",
dp[n][capacity]);
}

// Function to solve continuous Knapsack problem
void continuousKnapsack(struct Item items[], int n, int capacity) {
    int i;
    double totalValue = 0.0;
    int remainingCapacity = capacity;

    for (i = 0; i < n; i++) {
        if (remainingCapacity >= items[i].weight) {
            totalValue += items[i].value;
            remainingCapacity -= items[i].weight;
        } else {
            totalValue += (double)remainingCapacity / items[i].weight *
items[i].value;
            break;
        }
    }
}

```

```
    printf("Total value obtained for continuous knapsack: %.2lf\n",
totalValue);
}
```

```
int main() {
    int n, capacity, i;
    printf("Enter the number of items: ");
    scanf("%d", &n);

    struct Item items[n];

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

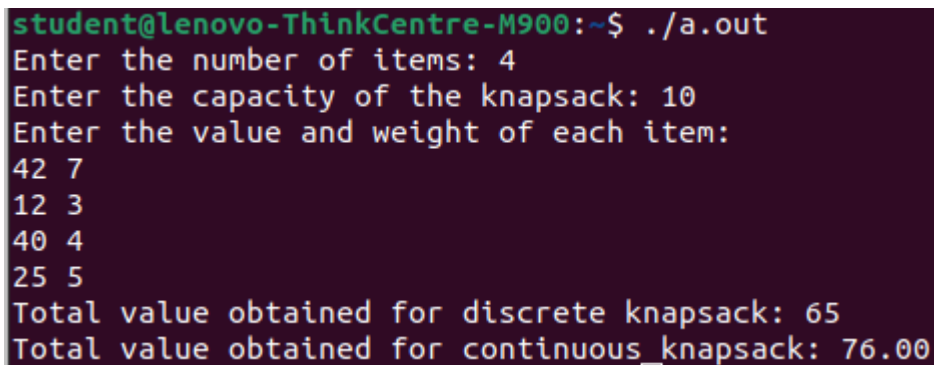
    printf("Enter the value and weight of each item:\n");
    for (i = 0; i < n; i++) {
        scanf("%d %d", &items[i].value, &items[i].weight);
        items[i].ratio = (double)items[i].value / items[i].weight;
    }

    // Sort items based on value-to-weight ratio
    qsort(items, n, sizeof(struct Item), compare);

    discreteKnapsack(items, n, capacity);
    continuousKnapsack(items, n, capacity);

    return 0;
}
```

OUTPUT:



```
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of items: 4
Enter the capacity of the knapsack: 10
Enter the value and weight of each item:
42 7
12 3
40 4
25 5
Total value obtained for discrete knapsack: 65
Total value obtained for continuous knapsack: 76.00
```

8.Design and implement C Program to find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d .

PROGRAM:

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

#define MAX_SIZE 100

// Function to find subset with given sum
void subsetSum(int set[], int subset[], int n, int subSize, int total, int
nodeCount, int sum)
{
    if (total == sum) {
        // Print the subset
        printf("Subset found: { ");
        for (int i = 0; i < subSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("}\n");
        return;
    } else {
        // Check the sum of the remaining elements
        for (int i = nodeCount; i < n; i++) {
            subset[subSize] = set[i];
            subsetSum(set, subset, n, subSize + 1, total + set[i], i + 1, sum);
        }
    }
}

int main() {
    int set[MAX_SIZE];
    int subset[MAX_SIZE];
    int n, sum;

    // Input the number of elements in the set
    printf("Enter the number of elements in the set: ");
```

```

scanf("%d", &n);

// Input the elements of the set
printf("Enter the elements of the set:\n");
for (int i = 0; i < n; i++) {
    scanf("%d", &set[i]);
}

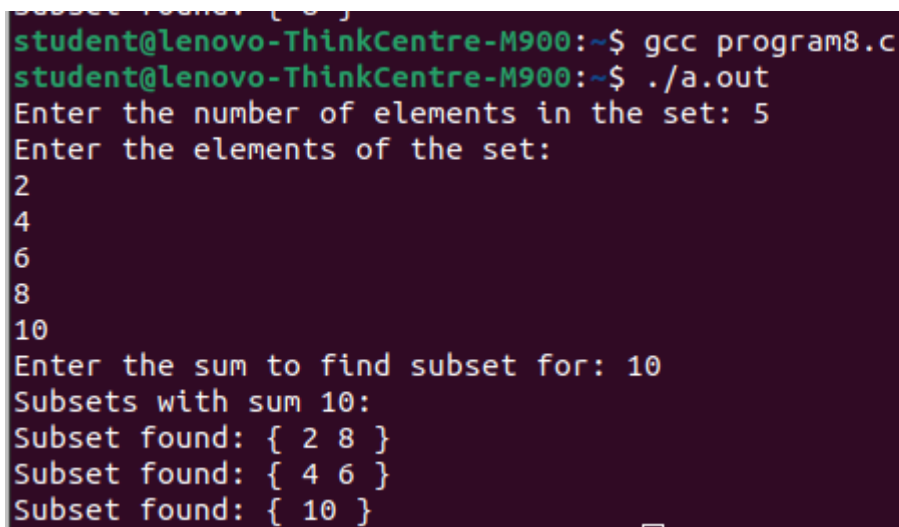
// Input the target sum
printf("Enter the sum to find subset for: ");
scanf("%d", &sum);

printf("Subsets with sum %d:\n", sum);
subsetSum(set, subset, n, 0, 0, 0, sum);

return 0;
}

```

OUTPUT:



```

student@lenovo-ThinkCentre-M900:~$ gcc program8.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements in the set: 5
Enter the elements of the set:
2
4
6
8
10
Enter the sum to find subset for: 10
Subsets with sum 10:
Subset found: { 2 8 }
Subset found: { 4 6 }
Subset found: { 10 }

```

9.Design and implement 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.

PROGRAM:

```
ALGORITHM SelectionSort( $A[0..n - 1]$ )  
  //Sorts a given array by selection sort  
  //Input: An array  $A[0..n - 1]$  of orderable elements  
  //Output: Array  $A[0..n - 1]$  sorted in nondecreasing order  
  for  $i \leftarrow 0$  to  $n - 2$  do  
     $min \leftarrow i$   
    for  $j \leftarrow i + 1$  to  $n - 1$  do  
      if  $A[j] < A[min]$   $min \leftarrow j$   
    swap  $A[i]$  and  $A[min]$ 
```

```
#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;  
    printf("Enter the number of elements: ");  
    scanf("%d", &n);  
  
    // Dynamic memory allocation for the array  
    int *arr = (int)malloc(n * sizeof(int));
```



```

// Generate random numbers for the array
srand(time(NULL));
for (int i = 0; i < n; i++) {
    arr[i] = rand() % 1000; // Generate random numbers between 0 and
999
}

clock_t start = clock(); // Start the timer
selectionSort(arr, n); // Sort the array
clock_t end = clock(); // End the timer
// Calculate the time taken
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Sorted array: ");
for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
}
printf("\n");
printf("Time taken: %f seconds\n", time_taken);
// Free dynamically allocated memory
free(arr);
return 0;
}

```

OUTPUT:

```

student@lenovo-ThinkCentre-M900:~$ gcc program9.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 5000
Time taken to sort 5000 elements: 0.028919 seconds
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 10000
Time taken to sort 10000 elements: 0.112973 seconds
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 15000
Time taken to sort 15000 elements: 0.250916 seconds
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 20000
Time taken to sort 20000 elements: 0.447036 seconds
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 25000
Time taken to sort 25000 elements: 0.693559 seconds

```

10.Design and implement 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.

PROGRAM:

```
ALGORITHM Quicksort( $A[l..r]$ )
    //Sorts a subarray by quicksort
    //Input: Subarray of array  $A[0..n - 1]$ , defined by its left and right
    //      indices  $l$  and  $r$ 
    //Output: Subarray  $A[l..r]$  sorted in nondecreasing order
    if  $l < r$ 
         $s \leftarrow \text{Partition}(A[l..r])$  //  $s$  is a split position
        Quicksort( $A[l..s - 1]$ )
        Quicksort( $A[s + 1..r]$ )
```

// C program to implement Quick Sort Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void quickSort(int arr[], int left, int right) {
    if (left >= right) return;
    int pivot = arr[right];
    int partitionIndex = left;
    for (int i = left; i < right; i++) {
        if (arr[i] < pivot) {
            int temp = arr[i];
            arr[i] = arr[partitionIndex];
            arr[partitionIndex] = temp;
            partitionIndex++;
        }
    }
    quickSort(arr, left, partitionIndex - 1);
    quickSort(arr, partitionIndex + 1, right);
}
```

```

    }
}
int temp = arr[partitionIndex];
arr[partitionIndex] = arr[right];
arr[right] = temp;
quickSort(arr, left, partitionIndex - 1);
quickSort(arr, partitionIndex + 1, right);
}

void generateRandomData(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100000; // Random numbers between 0 and 99999
    }
}

int main() {
    srand(time(0)); // Seed for random number generation

    printf("n \t Time (seconds)\n");
    printf("-----\n");

    for (int n = 5000; n <= 50000; n += 5000) {
        int *arr = (int *)malloc(n * sizeof(int));
        if (!arr) {
            printf("Memory allocation failed for %d elements.\n", n);
            return 1;
        }
        generateRandomData(arr, n);

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

        double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
        printf("%d \t %f\n", n, time_taken);

        free(arr);
    }

    return 0;
}

```

11.Design and implement 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.

PROGRAM:

```
ALGORITHM Mergesort( $A[0..n - 1]$ )
//Sorts array  $A[0..n - 1]$  by recursive mergesort
//Input: An array  $A[0..n - 1]$  of orderable elements
//Output: Array  $A[0..n - 1]$  sorted in nondecreasing order
if  $n > 1$ 
    copy  $A[0..\lfloor n/2 \rfloor - 1]$  to  $B[0..\lfloor n/2 \rfloor - 1]$ 
    copy  $A[\lfloor n/2 \rfloor..n - 1]$  to  $C[0..\lceil n/2 \rceil - 1]$ 
    Mergesort( $B[0..\lfloor n/2 \rfloor - 1]$ )
    Mergesort( $C[0..\lceil n/2 \rceil - 1]$ )
    Merge( $B, C, A$ ) //see below
```

```
ALGORITHM Merge( $B[0..p - 1], C[0..q - 1], A[0..p + q - 1]$ )
//Merges two sorted arrays into one sorted array
//Input: Arrays  $B[0..p - 1]$  and  $C[0..q - 1]$  both sorted
//Output: Sorted array  $A[0..p + q - 1]$  of the elements of  $B$  and  $C$ 
 $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$ 
while  $i < p$  and  $j < q$  do
    if  $B[i] \leq C[j]$ 
         $A[k] \leftarrow B[i]; i \leftarrow i + 1$ 
    else  $A[k] \leftarrow C[j]; j \leftarrow j + 1$ 
     $k \leftarrow k + 1$ 
if  $i = p$ 
    copy  $C[j..q - 1]$  to  $A[k..p + q - 1]$ 
else copy  $B[i..p - 1]$  to  $A[k..p + q - 1]$ 
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Function to merge two subarrays of arr[]
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    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 (i = 0; i < n1; i++)
```

```
    L[i] = arr[l + i];
```

```
for (j = 0; j < n2; j++)
```

```
    R[j] = arr[m + 1 + j];
```

```
// Merge the temporary arrays back into arr[l..r]
```

```
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++;
```

```
}
```

```
// Copy the remaining elements of L[], if there are any
```

```
while (i < n1) {
```

```
    arr[k] = L[i];
```

```
    i++;
```

```
    k++;
```

```
}
```

```
// Copy the remaining elements of R[], if there are any
```

```
while (j < n2) {
```

```
    arr[k] = R[j];
```

```
    j++;
```

```
    k++;
```

```
}
```

```
}
```

```
// Main function to implement Merge Sort
```

```

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

    // Dynamically allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));

    // Generate random numbers and fill the array
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100000; // Adjust range as needed
    }

    // Measure the time taken for sorting
    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

    // Free dynamically allocated memory
    free(arr);
    return 0;
}

```

OUTPUT:

```
student@lenovo-ThinkCentre-M900:~$ gcc 11.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 5000
Time taken to sort 5000 elements: 0.000691 seconds
student@lenovo-ThinkCentre-M900:~$ gcc 11.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 10000
Time taken to sort 10000 elements: 0.001521 seconds
student@lenovo-ThinkCentre-M900:~$ gcc 11.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 15000
Time taken to sort 15000 elements: 0.002262 seconds
student@lenovo-ThinkCentre-M900:~$ gcc 11.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 20000
Time taken to sort 20000 elements: 0.003134 seconds
student@lenovo-ThinkCentre-M900:~$ gcc 11.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
Enter the number of elements: 25000
Time taken to sort 25000 elements: 0.003956 seconds
```

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

PROGRAM:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int board[20],count;
```

```
int main()
{
int n,i,j;
void queen(int row, int n);
```

```
printf(" N Queens Problem Using Backtracking: ");
printf("\n\n Enter number of Queens:");
scanf("%d", &n);
queen(1,n);
return 0;
}
```

//function for printing the solution

```
void print(int n)
{
int i,j;
printf("\n\nSolution %d:\n\n",++count);
```

```
for(i=1;i<=n;++i)
    printf("\t%d",i);
```

```
for(i=1;i<=n;++i)
{
    printf("\n\n%d",i);
    for(j=1;j<=n;++j)        //for nxn board
    {
        if(board[i]==j)
            printf("\tQ");    //queen at i,j position
        else
            printf("\t-");    //empty slot
    }
}
}
```

/*function to check conflicts

//If no conflict for desired position returns 1 otherwise returns 0*/

```
int place(int row,int column)
{
int i;
for(i=1;i<=row-1;++i)
{
    //checking column and diagonal conflicts
    if(board[i]==column)
        return 0;
    else
        if(abs(board[i]-column)==abs(i-row))
            return 0;
}
}
```

```
return 1; //no conflicts
```



```
}
```

```
//function to check for proper positioning of queen
```

```
void queen(int row,int n)
```

```
{
```

```
int column;
```

```
for(column=1;column<=n;++column)
```

```
{
```

```
if(place(row,column))
```

```
{
```

```
board[row]=column; //no conflicts so place queen
```

```
if(row==n) //dead end
```

```
print(n); //printing the board configuration
```

```
else //try queen with next position
```

```
queen(row+1,n);
```

```
}
```

```
}
```

```
}
```

OUTPUT:

```
student@lenovo-ThinkCentre-M900:~$ gedit 12.c
student@lenovo-ThinkCentre-M900:~$ gcc 12.c
student@lenovo-ThinkCentre-M900:~$ ./a.out
- N Queens Problem Using Backtracking -

Enter number of Queens:4

Solution 1:
      1      2      3      4
1      -      Q      -      -
2      -      -      -      Q
3      Q      -      -      -
4      -      -      Q      -

Solution 2:
      1      2      3      4
1      -      -      Q      -
2      Q      -      -      -
3      -      -      -      Q
4      -      Q      -      -
-student@lenov
```

