DAA LAB

1. Find the min-max of the list of elements using DAC.

```
#include<stdio.h>
int a[100], n, I, max, min;
void minmax(int i, int j)
{
    int max1, min1, mid;
    if(i == j)
    {
        max = a[i];
        min = a[j];
    }
    else
    {
        if(i == (j - 1))
        {
            if(a[i] < a[j])
            {
                max = a[j];
                min = a[i];
            }
            else
            {
                max = a[i];
                min = a[j];
            }
        }
        else
        {
            mid = (i + j) / 2;
            minmax(i, mid);
            max1 = max;
            min1 = min;
            minmax(mid + 1, j);
            if(max < max1)
              max = max1;
            if(min > min1)
              min = min1;
        }
    }
```

```
}
void read()
{
   for(I = 0 ; I < n ; I++)
   {
      printf("\nEnter a[%d] : ", I);
      scanf("%d", &a[I]);
   }
}
void display()
{
   printf("\nArray values are : \n ");
   for(I = 0 ; I < n ; I++)
   {
      printf("%d   ", a[I]);
   }
}
void main()
{
   printf("\nEnter Array size : ");
   scanf("%d", &n);
   read();
   display();
   max = min = a[0];
   minmax(0, n - 1);
   printf("\nMaximum Element = %d \nMinimum Element = %d ", max, min);
}
```

Output -

```
Enter Array size : 5

Enter a[0] : 33
Enter a[1] : 22
Enter a[2] : 77
Enter a[3] : 11
Enter a[4] : 5

Array values are :
 33   22   77   11   5

Maximum Element = 77
Minimum Element = 5
```

2. Find the kth smallest element using DAC.

```
#include <stdio.h>
void read(int a[], int n)
{
   int i;
   for(i = 0 ; i < n ; i++)
   {
      printf("\nEnter a[%d] : ",i);
      scanf("%d",&a[i]);
   }
}
int partition(int a[], int lb, int ub)
{
   int lower = lb, upper = ub, t;
   int pivot = a[lb];
   while(lb < ub)
   {
      while(lb < upper && a[lb] <= pivot)
       lb++;
      while(ub > lower && a[ub] >= pivot)
       ub--;
      if(lb < ub)
      {
         t = a[lb];
         a[lb] = a[ub];
         a[ub] = t;
      }
   }
   a[lower] = a[ub];
   a[ub] = pivot;
   return ub;
}
int kthSmallest(int arr[], int lb, int ub, int k)
{
   int j;
   if(lb == ub)
   {
      if(lb == k - 1)
       return arr[k - 1];
      else
       return;
   }
   else
```

```
    {
        j = partition(arr, lb, ub);
        if(j == k - 1)
         return arr[k - 1];
        else if(j < k - 1)
         return kthSmallest(arr, j + 1, ub, k);
        else
         return kthSmallest(arr, lb, j - 1, k);
    }
}
void main()
{
    int i, n, k, a[100], kth_SmallestElement;
    printf("\nEnter number of elements : ");
    scanf("%d", &n);
    read(a, n);
    printf("\nEnter K'th element : ");
    scanf("%d", &k);
    kth_SmallestElement = kthSmallest(a, 0, n -1 , k);
    printf("\nThe K'th Smallest element = %d ", kth_SmallestElement);
}
```

Output -

```
Enter number of elements : 6

Enter a[0] : 10
Enter a[1] : 6
Enter a[2] : 3
Enter a[3] : 9
Enter a[4] : 7
Enter a[5] : 1

Enter K'th element : 5

The K'th Smallest element = 9
```

3. Calculate the optimal profit of a Knapsack using the Greedy method.

```
#include<stdio.h>
void knapsack(int n, float weight[], float profit[], float capacity)
{
    float x[30], tp = 0; //tp indicates Total Profit
```

```
    int i, j, u = capacity;
    for(i = 0 ; i < n ; i++)
    {
        x[i] = 0.0;
    }
    for(i = 0 ; i < n ; i++)
    {
        if(weight[i] > u)
        {
            break;
        }
        else
        {
            x[i] = 1.0;
            tp = tp + profit[i];
            u = u - weight[i];
        }
    }
    if(i < n)
    {
        x[i] = u / weight[i];
        tp = tp + ( x[i] * profit[i] );
    }
    printf("\nMaximum Profit = %.2f", tp);
}
void main()
{
    float weight[30], profit[30], temp, capacity, ratio[30];
    int n, i, j;
    printf("\nEnter Number of objects : ");
    scanf("%d", &n);
    printf("\nEnter Weight's and Profit's of Objects : ");
    for(i = 0 ; i < n ; i++)
    {
        printf("\nWeight[%d] & Profit[%d] : ", i, i);
        scanf("%f%f", &weight[i], &profit[i]);
    }
    printf("\nEnter the Capacity of Knapsack : ");
    scanf("%f", &capacity);
    for(i = 0 ; i < n ; i++)
    {
        ratio[i] = profit[i] / weight[i];
    }
    for(i = 0 ; i < n ; i++)
```

```
    {
        for(j = i + 1 ; j < n ; j++)
        {
            if(ratio[i] < ratio[j])
            {
                temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;

                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;

                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }
    knapsack(n, weight, profit, capacity);
}
```

Output -

```
Enter Number of objects : 7

Enter Weight's and Profit's of Objects :
Weight[0] & Profit[0] : 2 10
Weight[1] & Profit[1] : 3 5
Weight[2] & Profit[2] : 5 15
Weight[3] & Profit[3] : 7 7
Weight[4] & Profit[4] : 1 6
Weight[5] & Profit[5] : 4 18
Weight[6] & Profit[6] : 1 3

Enter the Capacity of Knapsack : 15

Maximum Profit = 55.33
```

4. Determine the path length from a source vertex to the other vertices in a given graph. (Dijkstra's algorithm)

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

```
#include<conio.h>
#define INFINITY 9999
#define MAX 25
int n, i, j, count, min_distance, start_node, next_node;
int adj[MAX][MAX], distance[MAX], visited[MAX], path[MAX];
void Find_Dijkstras()
{
        for(i = 0 ; i < n ; i++)
        {
           distance[i] = adj[start_node][i];
           path[i] = start_node;
           visited[i] = 0;
         }
         distance[start_node] = 0;
         visited[start_node] = 1;
         count = 1;
        while(count < n - 1)
        {
                min_distance = INFINITY;
                for(i = 0 ; i < n ; i++)
                {
                        if(distance[i] < min_distance && !visited[i])
                        {
                                min_distance = distance[i];
                                next_node = i;
                        }
                }
                visited[next_node] = 1;
                for(i = 0 ; i < n ; i++)
                {
                        if(!visited[i])
                        {
                                if(min_distance + adj[next_node][i] < distance[i])
                                {
                                        distance[i] = min_distance + adj[next_node][i];
                                         path[i] = next_node;
                                }
                }
         }
         count++;
        }
        for(i = 0 ; i < n ; i++)
        {
              if(i != start_node && distance[i] != INFINITY)
```

```
            {
                    printf("\nDistance of %d = %d", i, distance[i]);
                    printf("\nPath = %d", i);
                    j = i;
                    do
                    {
                          j = path[j];
                          printf(" <- %d", j);
                    }while(j != start_node);
            }
        }
}
void main()
{
   printf("\nEnter Number of vertices : ");
   scanf("%d", &n);
   printf("\nEnter the cost in the Adjacency Matrix : \n");
   for(i = 0 ; i < n ; i++)
   {
      for(j = 0 ; j < n ; j++)
      {
         scanf("%d", &adj[i][j]);
         if(adj[i][j] == 0)
            adj[i][j] = INFINITY;
      }
   }
   printf("\nEnter the Source node : ");
   scanf("%d", &start_node);
   printf("\nThe edges of the Minimum Cost Spanning Tree are : \n");
   Find_Dijkstras();
}
```

Output -

```
Enter Number of vertices : 5

Enter the cost in the Adjacency Matrix :
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0

Enter the Source node : 0
```

```
The edges of the Minimum Cost Spanning Tree are :
Distance of 1 = 10
Path = 1 <- 0
Distance of 2 = 50
Path = 2 <- 3 <- 0
Distance of 3 = 30
Path = 3 <- 0
Distance of 4 = 60
Path = 4 <- 2 <- 3 <- 0
```

5. Construct a minimum cost spanning tree for the given graph. (Kruskal's algorithm)

```
#include<stdio.h>
#define MAX 9999
int i, j, k, a, b, u, v, n, ne = 1;
int min, min_cost = 0, adj[30][30], parent[25];
int find(int i)
{
   while(parent[i])
   {
      i = parent[i];
   }
   return i;
}
int uni(int i, int j)
{
   if(i != j)
   {
      parent[j] = i;
      return 1;
   }
   return 0;
}
void Find_Kruskal()
{
   while(ne < n)
   {
      for(i = 1, min = MAX ; i <= n ; i++)
      {
         for(j = 1 ; j <= n ; j++)
```

```
            {
                if(adj[i][j] < min)
                {
                    min = adj[i][j];
                    a = u = i;
                    b = v = j;
                }
            }
        }
        u = find(u);
        v = find(v);
        if(uni(u, v))
        {
            printf("\n(%d, %d) = %d ", a, b, min);
            min_cost += min;
            ne++;
        }
        adj[a][b] = adj[b][a] = MAX;
    }
    printf("\nMinimum Cost = %d ", min_cost);
}
void main()
{
    printf("\nEnter Number of vertices : ");
    scanf("%d", &n);
    printf("\nEnter the cost in the Adjacency Matrix : \n");
    for(i = 1 ; i <= n ; i++)
    {
        for(j = 1 ; j <= n ; j++)
        {
            scanf("%d", &adj[i][j]);
            if(adj[i][j] == 0)
              adj[i][j] = MAX;
        }
    }
    printf("\nThe edges of the Minimum Cost Spanning Tree are : \n");
    Find_Kruskal();
}
```

Output -

Enter Number of vertices : 5

```
Enter the cost in the Adjacency Matrix :
0 8 5 0 0
8 0 9 11 0
5 9 0 15 10
0 11 15 0 7
0 0 10 7 0

The edges of the Minimum Cost Spanning Tree are :
(1, 3) = 5
(4, 5) = 7
(1, 2) = 8
(3, 5) = 10

Minimum Cost = 30
```

6. Determine the shortest path in a multi-stage graph using the forward and backward approach.

```
#include<stdio.h>
#define MAX 9999
int stages, n, min, i, j, c[30][30], cost[30], d[30], path[30];
void read()
{
   printf("\nEnter Number of vertices : ");
   scanf("%d", &n);
   printf("\nEnter Number of stages : ");
   scanf("%d", &stages);
   printf("\nEnter the cost in the Adjacency Matrix : \n");
   for(i = 1 ; i <= n ; i++)
   {
      for(j = 1 ; j <= n ; j++)
      {
         scanf("%d", &c[i][j]);
      }
   }
}
void Multi_Stage_Graph()
{
   cost[n] = 0;
   for(i = n - 1 ; i >= 1 ; i--)
   {
      min = MAX;
      for(j = i + 1 ; j <= n ; j++)
```

```
        {
            if(c[i][j] != 0 && c[i][j] + cost[j] < min)
            {
                min = c[i][j] + cost[j];
                d[i] = j;
            }
        }
        cost[i] = min;
    }
}
void Multi_Stage_Graph_Path()
{
    path[1] = 1, path[stages] = n;
    for(i = 2 ; i < stages ; i++)
    {
        path[i] = d[ path[i - 1] ];
    }

    printf("\nThe Shortest Path : ");
    for(i = 1 ; i < stages ; i++)
    {
        printf("%d -> ", path[i]);
    }
    printf("%d", path[i]);

    printf("\nThe Shortest Path value = %d ", cost[1]);
}
void main()
{
    read();
    Multi_Stage_Graph();
    Multi_Stage_Graph_Path();
}
```

<u>Output</u> -

```
Enter Number of vertices : 8

Enter Number of stages : 4

Enter the cost in the Adjacency Matrix :
0 2 1 3 0 0 0 0
0 0 0 0 2 3 0 0
0 0 0 0 6 7 0 0
```

```
0 0 0 0 6 8 9 0
0 0 0 0 0 0 0 6
0 0 0 0 0 0 0 4
0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 0

The Shortest Path : 1 -> 2 -> 6 -> 8
The Shortest Path value = 9
```

7. Find the Shortest path from any node to any other node (All-pairs Shortest path) within a graph.

```
#include<stdio.h>
#define INF 9999
int n, c[30][30], d[30][30], i, j, k;
void read()
{
   printf("\nEnter Number of vertices : ");
   scanf("%d", &n);
   printf("\nEnter the cost in the Adjacency Matrix : \n{ If there is no Self loop :- 0 & If
there is no edge :- (-1) }\n");
   for(i = 1 ; i <= n ; i++)
   {
      for(j = 1 ; j <= n ; j++)
      {
         scanf("%d", &c[i][j]);
         d[i][j] = c[i][j];
         if(c[i][j] == -1)
         {
            d[i][j] = c[i][j] = INF;
         }
      }
   }
}
int min(int a, int b)
{
   return ( (a < b) ? a : b );
}
void All_Pairs_Shortest_Path()
{
   for(k = 1 ; k <= n ; k++)
   {
```

```
        for(i = 1 ; i <= n ; i++)
        {
            for(j = 1 ; j <= n ; j++)
            {
                d[i][j] = min( d[i][j], d[i][k] + d[k][j] );
            }
        }
    }
}
void display()
{
    printf("\nThe Resultant Matrix : \n");
    for(i = 1 ; i <= n ; i++)
    {
        for(j = 1 ; j <= n ; j++)
        {
            if(j != 1)
            {
                printf("%7d", d[i][j]);
            }
            else
            {
                printf("%d", d[i][j]);
            }
        }
        printf("\n");
    }
}
void main()
{
    read();
    All_Pairs_Shortest_Path();
    display();
}
```

<u>Output</u> -

```
Enter Number of vertices : 7

Enter the cost in the Adjacency Matrix :
{ If there is no Self loop :- 0 & If there is no edge :- (-1) }
0 3 6 -1 -1 -1 -1
3 0 2 1 -1 -1 -1
6 2 0 1 4 2 -1
```

```
-1 1 1 0 2 -1 4
-1 -1 4 2 0 2 1
-1 -1 2 -1 2 0 1
-1 -1 -1 4 1 1 0

The Resultant Matrix :
0    3    5    4    6    7    7
3    0    2    1    3    4    4
5    2    0    1    3    2    3
4    1    1    0    2    3    3
6    3    3    2    0    2    1
7    4    2    3    2    0    1
7    4    3    3    1    1    0
```

8. Construct spanning trees using DFS and BFS graph traversals.

```
#include<stdio.h>
#include<stdlib.h>
int q[10], visited[100], visited1[100], n, G[100][100];
int front = -1;
int rear = -1;
void insert(int x)
{
   front = 0;
    rear++;
   q[rear] = x;
}
int delete()
{
   int x;
   front++;
   x = q[front];
   return x;
}
int isempty()
{
   if(front == -1 || front > rear)
   {
      return 1;
   }
   return 0;
}
```

```
void BFS(int v)
{
    int u = v, l, w, j;
    if(visited[v] == 0)
    printf("%d  ", u);
    visited[v] = 1;
    while(1)
    {
        for(int i = 1 ; i <= n ; i++)
        {
            if(G[u][i] == 1)
            {
                w=i;
                if(visited[w] == 0)
                {
                    insert(w);
                    printf("  %d  ", w);
                    visited[w] = 1;
                }
            }
        }
        l = isempty();
        if(l == 1)
         return ;
        u = delete();
    }
}
void BFT()
{
    for(int i = 1 ; i <= n ; i++)
    {
        BFS(i);
    }
}
void DFS(int v)
{
    int w;
    visited1[v] = 1;
    printf("%d    ", v);
    for(int i = 1 ; i <= n ; i++)
    {
        if(G[v][i] == 1)
        {
            w=i;
```

```
            if(visited1[w] == 0)
            {
                DFS(w);
            }
        }
    }
}
void main()
{
    int i, j;
    printf("\nEnter Number of vertices for graph 'G' : ");
    scanf("%d", &n);
    printf("\nEnter Adjacency Matrix : \n");
    for(i = 1 ; i <= n ; i++)
    {
        for(j = 1 ; j <= n ; j++)
        {
            scanf("%d", &G[i][j]);
        }
    }
    for(int i = 1 ; i <= n ; i++)
    {
        visited[i] = visited1[i] = 0;
    }
    printf("\nBreadth First Search Graph Traversal : \n");
    BFT();
    printf("\nDepth First Search Graph Traversal : \n");
    DFS(1);
}
```

<u>Output</u> -

Enter Number of vertices for graph 'G' : 8

Enter Adjacency Matrix :

0	1	1	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	0	0	0	1	0	0
0	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0
0	0	1	0	1	0	1	1
0	0	0	0	0	1	0	1
0	0	0	1	0	1	0	0

Breadth First Search Graph Traversal :

```
1   2   3   6   5   7   8   4

Depth First Search Graph Traversal :
1   2   4   8   6   3   5   7
```

10. Find the non-attacking positions of Queens in a given chess board using the backtracking technique.

```
#include<stdio.h>
#include<stdlib.h>
int board[20], count;
/* 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 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 "n x n" board
        {
            if(board[i] == j)
```

```
            printf("\tQ"); // Queen at "i, j" position
          else
            printf("\t-"); // Empty slot
        }
    }
}
// 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);
        }
    }
}
void main()
{
    int n, i, j;
    printf("\nEnter number of Queen's : ");
    scanf("%d", &n);
    queen(1, n);
}
```

<u>Output</u> -

```
Enter number of Queen's : 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    -    -
```

11. Color the nodes in a given graph such that no two adjacent can have the same color using the backtracking technique.

```
#include<stdio.h>
int n, graph[30][30];
void read()
{
   int i, j;
   printf("\nEnter Number of vertices : ");
   scanf("%d", &n);
   printf("\nEnter the Adjacency Matrix : \n");
   for(i = 0 ; i < n ; i++)
   {
      for(j = 0 ; j < n ; j++)
      {
         scanf("%d", &graph[i][j]);
      }
   }
}
void print(int color[])
{
   printf("\nSolution exists \nFollowing are the assigned colors : \n");
   for(int i = 0 ; i < n ; i++)
   {
      printf("%d  ", color[i]);
   }
}
```

```
// Check if the Colored, Graph is safe or not
int isSafe(int color[])
{
   // check for every edge
        for (int i = 0; i < n; i++)
                for (int j = i + 1; j < n; j++)
                        if (graph[i][j] && color[j] == color[i])
                                return 0;
        return 1;
}
int Graph_Coloring(int m, int i, int color[n])
{
   // if current index reached end
        if(i == n)
        {
                // if coloring is safe
                if( isSafe(color) )
                {
                        print(color);
                        return 1;
                }
           return 0;
        }
        // Assign each color from 1 to m
        for(int j = 1 ; j <= m ; j++)
        {
           color[i] = j;
           // Recur of the rest vertices
                if ( Graph_Coloring(m, i + 1, color))
                        return 1;
                color[i] = 0;
        }
        return 0;
}
void main()
{
   read();
   int color[n], m, i, j;
   printf("\nEnter Number of colors to be assigned : ");
   scanf("%d", &m);
   // Initialize all color values as 0
   for(i = 0 ; i < n ; i++)
   {
      color[i] = 0;
```

```
    }
    if( !Graph_Coloring(m, 0, color) )
    {
        printf("\nSolution does not exists");
    }
}
```

Output -

```
Enter Number of vertices : 4

Enter the Adjacency Matrix :
0 1 1 1
1 0 0 1
1 0 0 1
1 1 1 0

Enter Number of colors to be assigned : 3

Solution exists
Following are the assigned colors :
1  2  2  3
```

12. Calculate the optimal profit of a Knapsack using Dynamic programming.

```
#include<stdio.h>
int n, W, i, j, weight[30], value[30];
void read()
{
    printf("\nEnter Number of objects : ");
    scanf("%d", &n);
    printf("\nEnter Weight's and Value's of Objects : ");
    for(i = 0 ; i < n ; i++)
    {
        printf("\nWeight[%d] & Value[%d] : ", i, i);
        scanf("%d %d", &weight[i], &value[i]);
    }
    printf("\nEnter the Capacity of Knapsack : ");
    scanf("%d", &W);
}
int max(int a, int b)
{
```

```
    return ( (a > b) ? a : b );
}
void Knapsack()
{
    int i, j, knap[n+1][W+1];;
    for(i = 0 ; i <= n ; i++)
    {
        for(j = 0 ; j <= W ; j++)
        {
            if(i == 0 || j == 0)
            {
                knap[i][j] = 0;
            }
            else if(weight[i - 1] <= j)
            {
                knap[i][j] = max( value[i - 1] + knap[i - 1][j - weight[i - 1]], knap[i - 1][j] );
            }
            else
            {
                knap[i][j] = knap[i - 1][j];
            }
        }
    }
    printf("\nThe Solution = %d ", knap[n][W]);
}
void main()
{
    read();
    Knapsack();
}
```

Output -

```
Enter Number of objects : 4

Enter Weight's and Value's of Objects :
Weight[0] & Value[0] : 2 8
Weight[1] & Value[1] : 6 1
Weight[2] & Value[2] : 3 16
Weight[3] & Value[3] : 2 11

Enter the Capacity of Knapsack : 5

The Solution = 27
```