LAB - 10:

Dijkstra's Algorithm:

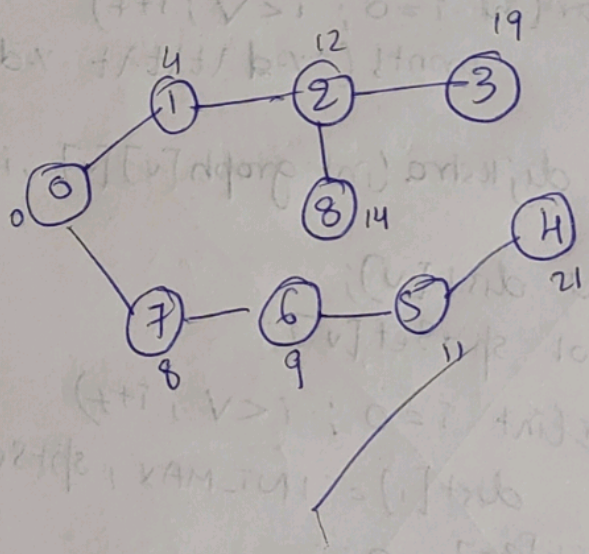
```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 6
int minDistance (int dist[], bool sptset[])
{ int min = INT_MAX, min_index;
  for (int v=0; v<V; v++)
      if (!sptset[v] && dist[v] <= min) {
          min = dist[v];
          min_index = v;
      }
  return min_index;
}
void printSolution (int dist[])
{ printf("Vertex \t\t Distance from source\n");
  for (int i=0; i<V; i++)
      printf("%d \t\t\t %d\n", i, dist[i]);
}
void dijkstra (int graph[V][V], int src)
{
  int dist[V];
  bool sptSet[V];
  for (int i=0; i<V; i++)
      dist[i] = INT_MAX, sptset[i] = false;
  dist[src] = 0;
  for (int count = 0; count < V-1; count++) {
      int u = minDistance (dist, sptSet);
      sptSet[u] = true;
      for (int v = 0; v<V; 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];
  }
```

```
int main()
{
    int graph[V][V];
    printf("Enter the values of adjacency matrix:\n");
    for(int i=0; i<V ; i++) {
        for(int j=0; j<V ; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    dijkstra(graph,0);
    return 0;
}
```

Output:

Enter the values of adjacency matrix:

```
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  14 10 0  2  0  0
0  0  0  0  0  2  0  1  6
8  11 0  0  0  0  1  0  7
0  0  2  0  0  0  6  7  0
```



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

Kruskal's Algorithm

```
#include <stdio.h>
#include <stdlib.h>
int comparator(const void *p1, const void *p2)
{
  const int (*x)[3] = p1;
  const int (*y)[3] = p2;
  return (*x)[2] - (*y)[2];
}
void makeset(int parent[], int rank[], int n)
{
  for(int i=0; i<n; i++){
    parent[i] = i;
    rank[i] = 0;
  }
}
int findParent(int parent[], int component)
{
  if(parent[component] != component){
    parent[component] = findParent(parent, parent[component]);
  }
  return parent[component];
}
void unionset(int u, int v, int parent[], int rank[])
{
  u = findParent(parent, u);
  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]++;
```

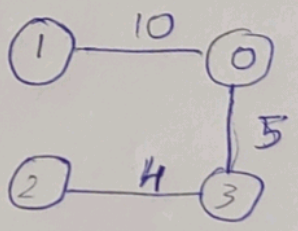
```
void kruskalAlgo (int n, int edges[][3], int e)
{
    qsort(edges, e, sizeof(edge[0]), comparator);
    int parent[n];
    int rank[n];
    makeSet(parent, rank, n);
    int minCost = 0;
    printf("Following are edges in the constructed MST\n");
    for(int i=0; i<e; i++){
        int u = edges[i][0];
        int v = edges[i][1];
        int wt = edges[i][2];

        int parent_u = findParent(parent, u);
        int parent_v = findParent(parent, v);
        if(parent_u != parent_v){
            unionSet(parent_u, parent_v, parent, rank);
            minCost += wt;
            printf("%d -- %d == %d\n", u, v, wt);
        }
    }
    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

int main()
{
    int n, e;
    printf("Enter no. of vertices: ");
    scanf("%d", &n);
    printf("Enter no. of edges: ");
    scanf("%d", &e);

    int edges[e][3];
    printf("Enter the edges (u v wt):\n");
```

```
for(int i=0; i<e; i++){
    scanf("%d %d %d", &edges[i][0], &edges[i][1], &
                      &edges[i][2]);
}
kruskalAlgo(n, edges, e);
return 0;
}
```

Output:

Enter the no. of vertices: 5
Enter the no. of edges: 5
Enter the edges (u v wt):

0 1 10
0 2 6
0 3 5
1 3 15
2 3 4



Following are the edges in the constructed MST

2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10

Minimum spanning Tree: 19

Greedy knapsack

```
#include<stdio.h>
void main(){
  int n;
  float m;
  printf("Enter the capacity\n");
  scanf("%f", &m);
  printf("Enter the no. of objects\n");
  scanf("%d", &n);
  printf("Enter the elements of Profit/weight of %d obj\n", n);
  float w[n], p[n], x[n];
  float ratio[n];
  for(int i=0; i<n; i++){
      scanf("%f%f", &p[i], &w[i]);
      x[i]=0;
      ratio[i]=p[i]/w[i];
  }
  for(int i=0; i<n-1; i++){
    for(int j=0; j<n-i-1; j++){
      if(ratio[j]<ratio[j+1]){
        float tp = p[j+1];
        p[j+1] = p[j];
        p[j] = tp;
        float tw = w[j+1];
        w[j+1] = w[j];
        w[j] = tw;
        float tr = ratio[j+1];
        ratio[j+1] = ratio[j];
        ratio[j] = tr;
      }
```

```
    float rc=m;
    float mp=0;
    for(int i=0; i<n; i++){
        if(w[i]<=rc){
            x[i]=1;
            rc-=w[i];
            mp+=p[i];
        }
        else{
            x[i]=rc/w[i];
            mp+=x[i]*p[i];
            break;
        }
    }
    printf("The seleded objects are:\n");
    for(int i=0; i<n; i++){
        if(x[i]){
            printf("object %d (fraction: %.2f)\n", i+1, x[i]);
        }
    }
    printf("The Maximum Profit is: %.2f\n", mp);
}
```

Output:

```
Enter the capacity: 10
Enter no. of objects: 3
Enter the elements of Profit/weight of 3 objects
10 20
10 25
35 10
The selected objects are:
Object 1 (fraction: 1.00)
object 2 (fraction: 1.00)
object 3 (fraction: 0.25)
The Maximum Profit is: 82.50
```

NQueens

```
#include<stdio.h>
#include<stdbool.h>
#define N 8
void printSolution(int board[N][N]) {
    for(int i=0; i<N; i++) {
        for(int j=0; j<N; j++) {
            printf("%2d ", board[i][j]);
        }
        printf("\n");
    }
}
bool isSafe(int board[N][N], int row, int col) {
    int i, j;
    for(i=0; i<col; i++)
        if(board[row][i])
            return false;

    for(i=row, j=col; i>=0 && j>=0; i--, j--)
        if(board[i][j])
            return false;

    for(i=row, j=col; j>=0 && i<N; i++, j--)
        if(board[i][j])
            return false;

    return true;
}
bool solveNQUtil(int board[N][N], int col) {
    if(col>=N)
        return true;

    for(int i=0; i<N; i++) {
        if(isSafe(board, i, col)) {
            board[i][col]=1;
            if(solveNQUtil(board, col+1))
                return true;
```

```
        board[i][col] = 0;
    }
  }
  return false;
}
bool solveNQ() {
  int board[N][N] = {0};
  if (!solveNQUtil(board, 0)) {
    printf("Solution does not exist");
    return false;
  }
  printSolution(board);
  return true;
}
int main() {
  solveNQ();
  return 0;
}
```

Output:

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