VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Analysis and Design of Algorithms

Submitted by

TANMAY BHARADWAJ
(1BM22CS303)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
April-2024 to August-2024

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated to Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Analysis and Design of Algorithms**" carried out by **TANMAY BHARADWAJ (1BM22CS303),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester April-2024 to August-2024. The Lab report has been approved as it satisfies the academic requirements in respect of an **Analysis and Design of Algorithms (23CS4PCADA)** work prescribed for the said degree.

Dr. Kayarvizhy N
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations.
CO2	Apply various design techniques for the given problem.
CO3	Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete
CO4	Design efficient algorithms and conduct practical experiments to solve problems.

1.Write a program to obtain the Topological ordering of vertices in a given digraph using Source Removal method.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 100
typedef struct Queue
{
   int items[MAX_VERTICES];
   int front;
   int rear;
} Queue;
void enqueue(Queue *q, int value);
int dequeue(Queue *q);
int isEmpty(Queue *q);
void topologicalSort(int n, int graph[][MAX_VERTICES]);
int main()
{
   int n;
   printf("Enter the number of vertices in the graph: ");
   scanf("%d", &n);
   int graph[MAX_VERTICES][MAX_VERTICES] = {0};
   printf("Enter the adjacency matrix of the graph:\n");
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n; j++)
      {
         scanf("%d", &graph[i][j]);
      }
   }
```

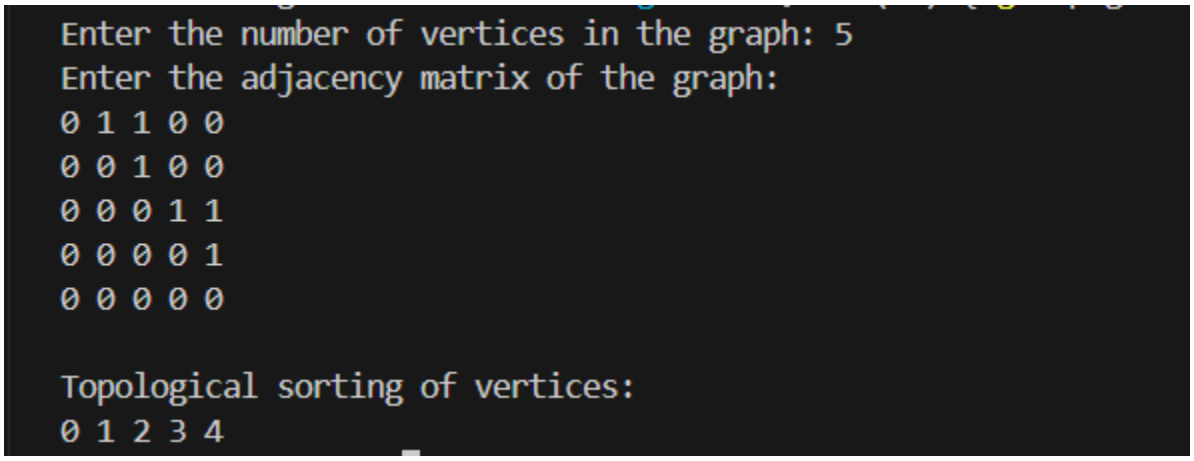
```
    printf("\nTopological sorting of vertices:\n");
    topologicalSort(n, graph);
    return 0;
}
void topologicalSort(int n, int graph[][MAX_VERTICES])
{
    int indegree[MAX_VERTICES] = {0};
    Queue q;
    q.front = -1;
    q.rear = -1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (graph[i][j] == 1)
            {
                indegree[j]++;
            }
        }
    }
    for (int i = 0; i < n; i++)
    {
        if (indegree[i] == 0)
        {
            enqueue(&q, i);
        }
    }
```

```
  while (!isEmpty(&q))
    {
        int vertex = dequeue(&q);
        printf("%d ", vertex);

        for (int i = 0; i < n; i++)
        {
            if (graph[vertex][i] == 1)
            {
                if (--indegree[i] == 0)
                {
                    enqueue(&q, i);
                }
            }
        }
    }
    printf("\n");
}
void enqueue(Queue *q, int value)
{
    if (q->rear == MAX_VERTICES - 1)
    {
        printf("Queue is full\n");
    }
    else{
        if (q->front == -1)
        {
            q->front = 0;
```

```
        }
        q->rear++;
        q->items[q->rear] = value;
    }
}
int dequeue(Queue *q)
{
    int item;
    if (isEmpty(q))
    {
        printf("Queue is empty\n");
        item = -1;
    }
    else
    {
        item = q->items[q->front];
        q -> front++;
        if (q->front > q->rear)
        {
            q->front = q->rear = -1;
        }
    }
    return item;
}

int isEmpty(Queue *q)
{
    if (q->rear == -1)
```

```
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

OUTPUT:



2.Implement Johnson Trotter algorithm to generate permutations

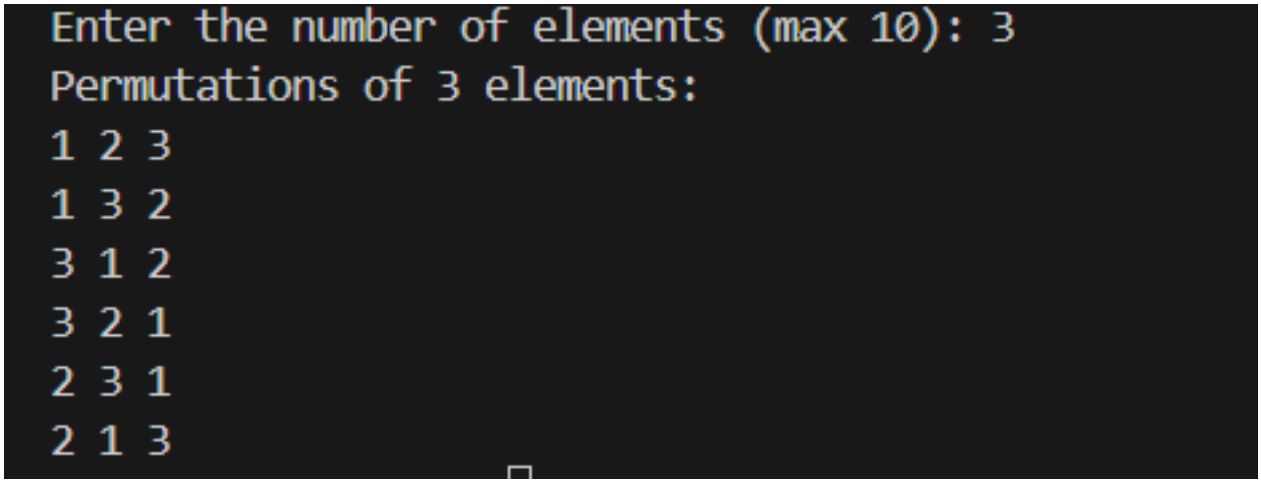
```
#include <stdio.h>
#include <stdbool.h>
#define MAX_N 10
#define LEFT 0
#define RIGHT 1
typedef struct
{
   int value;
   int direction;
} Element;
void printPermutations(int n);
void generatePermutations(Element permutation[], int n);
int findLargestMobile(Element permutation[], int n);
int main()
{
   int n;
   printf("Enter the number of elements (max %d): ", MAX_N);
   scanf("%d", &n);
   if (n > MAX_N || n <= 0)
   {
      printf("Invalid input size. Please enter a valid number between 1 and %d.\n", MAX_N);
      return 1;
   }
   printf("Permutations of %d elements:\n", n);
   printPermutations(n);
   return 0;
}
```

```
void printPermutations(int n)
{
   Element permutation[MAX_N];
   for (int i = 0; i < n; i++)
   {
      permutation[i].value = i + 1;
      permutation[i].direction = LEFT;
   }
   for (int i = 0; i < n; i++)
   {
      printf("%d ", permutation[i].value);
   }
   printf("\n");
   generatePermutations(permutation, n);
}
void generatePermutations(Element permutation[], int n)
{
   while (true)
   {
      int mobileIdx = findLargestMobile(permutation, n);

      if (mobileIdx == -1)
      {
         break;
      }
      int swapIdx = mobileIdx + (permutation[mobileIdx].direction == LEFT ? -1 : 1);
      Element temp = permutation[mobileIdx];
```

```
        permutation[mobileIdx] = permutation[swapIdx];
        permutation[swapIdx] = temp;
        for (int i = 0; i < n; i++)
        {
            if (permutation[i].value > permutation[swapIdx].value)
            {
                permutation[i].direction = (permutation[i].direction == LEFT) ? RIGHT : LEFT;
            }
        }
        for (int i = 0; i < n; i++)
        {
            printf("%d ", permutation[i].value);
        }
        printf("\n");   }    }
int findLargestMobile(Element permutation[], int n)
{
    int mobileIdx = -1;
    int maxMobileValue = -1;
    for (int i = 0; i < n; i++)
    {
        int direction = permutation[i].direction;
        int adjacentIdx = i + (direction == LEFT ? -1 : 1);
        if (adjacentIdx >= 0 && adjacentIdx < n &&
            permutation[i].value > permutation[adjacentIdx].value &&
            permutation[i].value > maxMobileValue)
        {
            mobileIdx = i;
            maxMobileValue = permutation[i].value;
```

```
        }
    }
    return mobileIdx;
}
```

OUTPUT:



3.Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void merge(int arr[], int left, int mid, int right)
{
   int n1 = mid - left + 1;
   int n2 = right - mid;
   int L[n1], R[n2];
   for (int i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (int j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];
   int i = 0;
   int j = 0;
   int k = left;
   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++;
    }
}
void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

```
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int n;
    clock_t start, end;
    double cpu_time_used;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Unsorted array: ");
    printArray(arr, n);
    start = clock();
    mergeSort(arr, 0, n - 1);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array: ");
    printArray(arr, n);
    printf("Time taken to sort: %f seconds\n", cpu_time_used);
    return 0;
}
```

OUTPUT:

```
Enter the number of elements: 6
Enter 6 integers:
3 4 1 6 3 2
Unsorted array: 3 4 1 6 3 2
Sorted array: 1 2 3 3 4 6
Time taken to sort: 0.000000 seconds
```

4.Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void swap(int *a, int *b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
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++;
         swap(&arr[i], &arr[j]);
      }
```

```
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
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);
    }
}
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int n;
    clock_t start, end;
    double cpu_time_used;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
```

```
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Unsorted array: ");
    printArray(arr, n);
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array: ");
    printArray(arr, n);
    printf("Time taken to sort: %f seconds\n", cpu_time_used);
    return 0;
}
```

OUTPUT:

```
Enter the number of elements: 10
Enter 10 integers:
9 7 5 2 4 6 8 1 10 0
Unsorted array: 9 7 5 2 4 6 8 1 10 0
Sorted array: 0 1 2 4 5 6 7 8 9 10
Time taken to sort: 0.000000 seconds
```

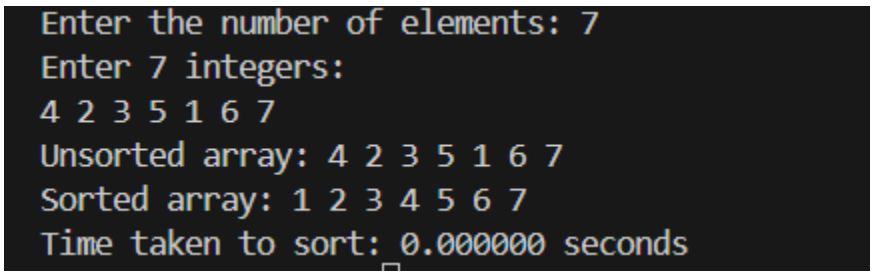
5.Sort a given set of N integer elements using Heap Sort technique and compute its time taken.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void heapify(int arr[], int n, int i)
{
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;
   if (l < n && arr[l] > arr[largest])
      largest = l;
   if (r < n && arr[r] > arr[largest])
      largest = r;
   if (largest != i)
   {
      int temp = arr[i];
      arr[i] = arr[largest];
      arr[largest] = temp;
      heapify(arr, n, largest);
   }
}
void heapSort(int arr[], int n)
{
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);
   for (int i = n - 1; i > 0; i--)
   {
      int temp = arr[0];
```

```
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int n;
    clock_t start, end;
    double cpu_time_used;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Unsorted array: ");
    printArray(arr, n);
    start = clock();
    heapSort(arr, n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
```

```
    printf("Sorted array: ");
    printArray(arr, n);
    printf("Time taken to sort: %f seconds\n", cpu_time_used);
    return 0;
}
```

OUTPUT:

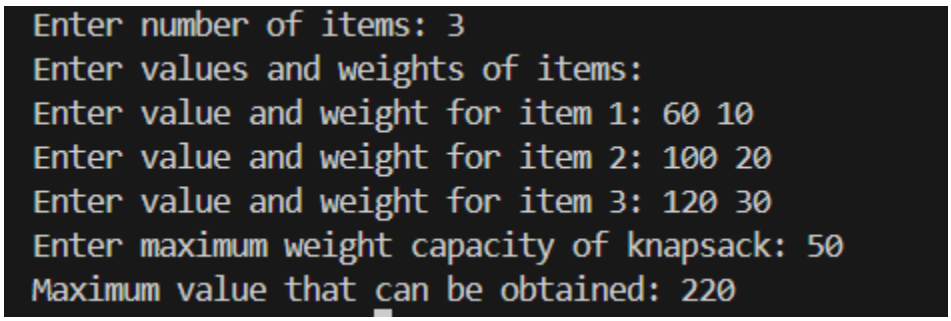


6.Implement 0/1 Knapsack problem using dynamic programming.

```
#include <stdio.h>
int max(int a, int b)
{
   return (a > b) ? a : b;
}
int knapsack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n + 1][W + 1];
   for (i = 0; i <= n; i++)
   {
      for (w = 0; w <= W; w++)
      {
         if (i == 0 || w == 0)
            K[i][w] = 0;
         else if (wt[i - 1] <= w)
            K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
         else
            K[i][w] = K[i - 1][w];
      }
   }
   return K[n][W];
}
int main()
{
   int n, W;
   printf("Enter number of items: ");
```

```
    scanf("%d", &n);
    int val[n], wt[n];
    printf("Enter values and weights of items:\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 maximum weight capacity of knapsack: ");
    scanf("%d", &W);
    int max_value = knapsack(W, wt, val, n);
    printf("Maximum value that can be obtained: %d\n", max_value);
    return 0;
}
```

OUTPUT:



7.Implement All Pair Shortest paths problem using Floyd's algorithm.

```
#include <stdio.h>
#include <limits.h>
#define V 4
void printSolution(int dist[][V])
{
   printf("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] == INT_MAX)
            printf("INF\t");
         else
            printf("%d\t", dist[i][j]);
      }
      printf("\n");
   }
}
void floydWarshall(int graph[][V])
{
   int dist[V][V];
   for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
         dist[i][j] = graph[i][j];
   for (int k = 0; k < V; k++)
   {
      for (int i = 0; i < V; i++)
```

```
        {
            for (int j = 0; j < V; j++)
            {
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    printSolution(dist);
}
int main()
{
    int graph[V][V] = {
        {0, 5, INT_MAX, 10},
        {INT_MAX, 0, 3, INT_MAX},
        {INT_MAX, INT_MAX, 0, 1},
        {INT_MAX, INT_MAX, INT_MAX, 0}};
    floydWarshall(graph);
    return 0;
}
```

OUTPUT:

```
Shortest distances between every pair of vertices:
0       5       8       9
INF     0       3       4
INF     INF     0       1
INF     INF     INF     0
```

8.(a) Find Minimum Cost Spanning Tree of a given undirected graph usingPrim's algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define V 5
int minKey(int key[], int mstSet[])
{
   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;
}
void 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]]);
}
void primMST(int graph[V][V])
{
   int parent[V];
   int key[V];
   int mstSet[V];
   for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = 0;
   key[0] = 0;
   parent[0] = -1;
```

```
    for (int count = 0; count < V - 1; count++)
    {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;
        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];
    }
    printMST(parent, graph);
}
int main()
{
    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
```

(b) Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 20
struct Edge
{
   int src, dest, weight;
};
void Union(int parent[], int rank[], int x, int y);
int find(int parent[], int i);
void KruskalMST(struct Edge *edges, int V, int E);
int find(int parent[], int i)
{
   if (parent[i] != i)
      parent[i] = find(parent, parent[i]);
   return parent[i];
}
void Union(int parent[], int rank[], int x, int y)
{
   int xroot = find(parent, x);
   int yroot = find(parent, y);
   if (rank[xroot] < rank[yroot])
      parent[xroot] = yroot;
   else if (rank[xroot] > rank[yroot])
      parent[yroot] = xroot;
   else
   {
      parent[yroot] = xroot;
```

```
        rank[xroot]++;
    }
}
int compareEdges(const void *a, const void *b)
{
    struct Edge *edge1 = (struct Edge *)a;
    struct Edge *edge2 = (struct Edge *)b;
    return edge1->weight - edge2->weight;
}
void KruskalMST(struct Edge *edges, int V, int E)
{
    struct Edge result[V];
    int e = 0;
    int i = 0;
    qsort(edges, E, sizeof(struct Edge), compareEdges);
    int parent[V];
    int rank[V];
    for (int v = 0; v < V; ++v)
    {
        parent[v] = v;
        rank[v] = 0;
    }
    while (e < V - 1 && i < E)
    {
        struct Edge next_edge = edges[i++];
        int u = find(parent, next_edge.src);
        int v = find(parent, next_edge.dest);
        if (u != v)
```

```
        {
            result[e++] = next_edge;
            Union(parent, rank, u, v);
        }
    }
    printf("Edges in the Minimum Spanning Tree:\n");
    for (i = 0; i < e; ++i)
    {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
    }
}
int main()
{
    int V, E;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    struct Edge edges[E];
    printf("Enter the source, destination, and weight of each edge:\n");
    for (int i = 0; i < E; ++i)
    {
        scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
    }
    KruskalMST(edges, V, E);
    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the source, destination, and weight of each edge:
0 1 10
0 2 6
0 3 5
1 3 15
1 2 6
Edges in the Minimum Spanning Tree:
0 -- 3 == 5
0 -- 2 == 6
1 -- 2 == 6
```

9.Implement Fractional Knapsack using Greedy technique.

```
#include <stdio.h>
#include <stdlib.h>
double fractionalKnapsack(int capacity, int weights[], int values[], int n)
{
   double valuePerWeight[n];
   for (int i = 0; i < n; ++i)
   {
      valuePerWeight[i] = (double)values[i] / weights[i];
   }
   int currentWeight = 0;
   double finalValue = 0.0;
   while (currentWeight < capacity)
   {
      int bestItem = -1;
      double bestRatio = 0.0;
      for (int i = 0; i < n; ++i)
      {
```

```
            if (weights[i] > 0 && (bestItem == -1 || valuePerWeight[i] > bestRatio))
            {
                bestItem = i;
                bestRatio = valuePerWeight[i];
            }
        }
        if (bestItem == -1)
        {
            break;
        }
        int takeWeight = (capacity - currentWeight < weights[bestItem]) ?
                                            (capacity currentWeight) : weights[bestItem];
        currentWeight += takeWeight;
        finalValue += takeWeight * valuePerWeight[bestItem];
        weights[bestItem] -= takeWeight;
    }
    return finalValue;
}
int main()
{
    int n;
    printf("Enter the number of items: ");
    scanf("%d", &n);
    int capacity;
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);
    int weights[n];
    int values[n];
```

```
    printf("Enter weight and value of each item:\n");
    for (int i = 0; i < n; ++i)
    {
        printf("Item %d:\n", i + 1);
        printf("Weight: ");
        scanf("%d", &weights[i]);
        printf("Value: ");
        scanf("%d", &values[i]);
    }
    double maxValue = fractionalKnapsack(capacity, weights, values, n);
    printf("Maximum value in Knapsack = %.2f\n", maxValue);
    return 0;
}
```

OUTPUT:

```
Enter the number of items: 3
Enter the capacity of the knapsack: 50
Enter weight and value of each item:
Item 1:
Weight: 20
Value: 100
Item 2:
Weight: 20
Value: 50
Item 3:
Weight: 20
Value: 90
Maximum value in Knapsack = 215.00
```

10.From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define MAX 100
int minDistance(int dist[], bool sptSet[], int V)
{
   int min = INT_MAX, 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;
}
void printSolution(int dist[], int V)
{
   printf("Vertex \t Distance from Source\n");
   for (int i = 0; i < V; i++)
      printf("%d \t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[MAX][MAX], int src, int V)
{
   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, V);
        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];
    }
    printSolution(dist, V);
}
int main()
{
    int V;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    int graph[MAX][MAX];
    printf("Enter the adjacency matrix (enter 0 if there is no edge between two vertices):\n");
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            scanf("%d", &graph[i][j]);
    int src;
    printf("Enter the source vertex: ");
    scanf("%d", &src);
    dijkstra(graph, src, V);
    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 5
Enter the adjacency matrix (enter 0 if there is no edge between two vertices):
0 10 0 0 5
10 0 1 0 2
0 1 0 4 0
0 0 4 0 3
5 2 0 3 0
Enter the source vertex: 0
Vertex   Distance from Source
0                0
1                7
2                8
3                8
4                5
```

12.Implement “N-Queens Problem” using Backtracking.

```
#include <stdio.h>

#include <stdbool.h>

#define MAX_N 10

void printSolution(int board[MAX_N][MAX_N], int N)

{

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

   {

      for (int j = 0; j < N; j++)

      {

         printf("%c ", board[i][j] ? 'Q' : '.');

      }

      printf("\n");

   }

}

bool isSafe(int board[MAX_N][MAX_N], int row, int col, int N)

{

   for (int i = 0; i < col; i++)

   {
```

```
        if (board[row][i])
        {
            return false;
        }
    }
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j])
        {
            return false;
        }
    }
    for (int i = row, j = col; i < N && j >= 0; i++, j--)
    {
        if (board[i][j])
        {
            return false;
        }
    }
    return true;
}
bool solveNQueens(int board[MAX_N][MAX_N], int col, int N)
{
    if (col >= N)
    {
        return true;
    }
```

```
    for (int i = 0; i < N; i++)
    {
        if (isSafe(board, i, col, N))
        {
            board[i][col] = 1;
            if (solveNQueens(board, col + 1, N))
            {
                return true;
            }
            board[i][col] = 0;
        }
    }
    return false;
}
void solveNQueensWrapper(int N)
{
    int board[MAX_N][MAX_N] = {0};
    if (solveNQueens(board, 0, N))
    {
        printf("Solution found:\n");
        printSolution(board, N);
    }
    else
    {
        printf("Solution does not exist for N = %d\n", N);
    }
}
```

```
int main()
{
   int N;
   printf("Enter the size of the chessboard (N): ");
   scanf("%d", &N);
   if (N <= 0 || N > MAX_N)
   {
      printf("Invalid input for N. Please enter a number between 1 and %d.\n", MAX_N);
      return 1;
   }
   solveNQueensWrapper(N);
   return 0;
}
```

OUTPUT: