
NAME: JUNAID GIRKAR

SUBJECT: ANALYSIS OF ALGORITHM

ANNUAL YEAR: 2020 - 2021

SAP ID: 60004190057

BRANCH: COMPUTER ENGINEERING

DIVISION: A (A-3)

SEMESTER: 4

INDEX

SR NO.	EXPERIMENT NO.	EXPERIMENT NAME	PAGE NO.
1	1.1	Selection Sort	2
2	1.2	Insertion Sort	6
3	2.1	Bellman-Ford Algorithm	10
4	2.2	Floyd Warshall Algorithm	17
5	3.1	Dijkstra's Algorithm	21
6	3.2	0/1 Knapsack Problem	26
7	4.1	N Queen Problem	31
8	4.2	KMP Algorithm	36

MODULE - 1

MODULE NAME: Introduction to analysis of algorithm Divide and Conquer approach

EXPERIMENT - 1.1

AIM: Selection sort

COMPLEXITY:

- Worst Case Time Complexity [Big-O]: $O(n^2)$
- Best Case Time Complexity [Big-omega]: $O(n^2)$
- Average Time Complexity [Big-theta]: $O(n^2)$
- Space Complexity: $O(1)$

THEORY:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

ALGORITHM:

```
Step 1 - Set MIN to location 0
Step 2 - Search the minimum element in the list
Step 3 - Swap with value at location MIN
Step 4 - Increment MIN to point to next element
Step 5 - Repeat until list is sorted
```

PSEUDOCODE:

```
procedure selection sort
    list  : array of items
    n     : size of list

    for i = 1 to n - 1
        /* set current element as minimum*/
        min = i

        /* check the element to be minimum */

        for j = i+1 to n
            if list[j] < list[min] then
                min = j;
            end if
        end for

        /* swap the minimum element with the current element*/
        if indexMin != i then
            swap list[min] and list[i]
        end if
    end for

end procedure
```

CODE:

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        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
        swap(&arr[min_idx], &arr[i]);
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

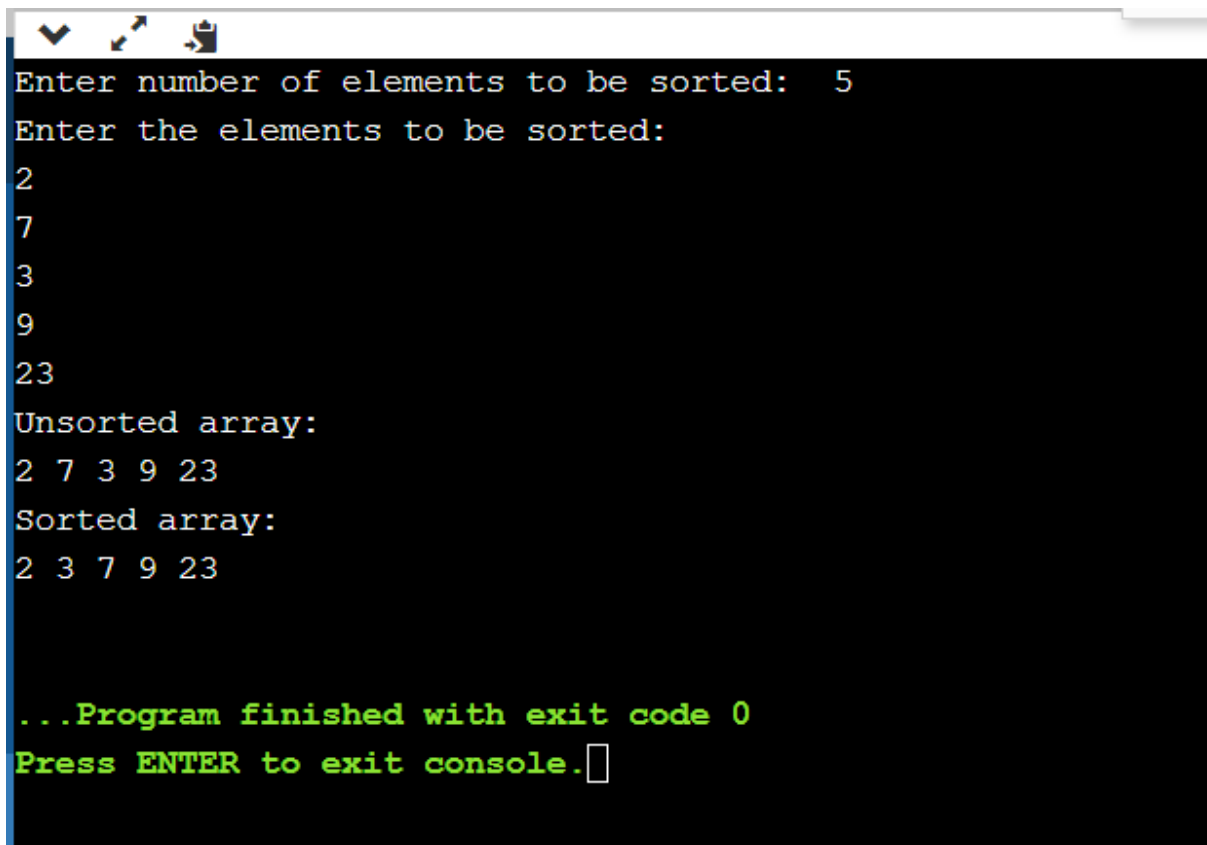
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Unsorted array: \n");
    printArray(arr,n);

    selectionSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}

```

OUTPUT:



```
Enter number of elements to be sorted: 5
Enter the elements to be sorted:
2
7
3
9
23
Unsorted array:
2 7 3 9 23
Sorted array:
2 3 7 9 23

...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION: We learnt about Selection Sort and implemented that in a C Program.

EXPERIMENT - 1.2

AIM: Insertion Sort

COMPLEXITY:

Worst-case time complexity: $O(n^2)$

Best-case time complexity: $O(n)$

Average time complexity: $O(n^2)$

Space complexity: $O(1)$

THEORY:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

ALGORITHM:

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 - If it is the first element, it is already sorted. return 1;
Step 2 - Pick next element
Step 3 - Compare with all elements in the sorted sub-list
Step 4 - Shift all the elements in the sorted sub-list that is greater than the
           value to be sorted
Step 5 - Insert the value
Step 6 - Repeat until list is sorted
```

PSEUDOCODE:

```
procedure insertionSort( A : array of items )
    int holePosition
    int valueToInsert

    for i = 1 to length(A) inclusive do:

        /* select value to be inserted */
        valueToInsert = A[i]
        holePosition = i

        /*locate hole position for the element to be inserted */

        while holePosition > 0 and A[holePosition-1] > valueToInsert do:
            A[holePosition] = A[holePosition-1]
            holePosition = holePosition - 1
        end while

        /* insert the number at hole position */
        A[holePosition] = valueToInsert

    end for

end procedure
```

CODE:

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

void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
    }
}
```



```

        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

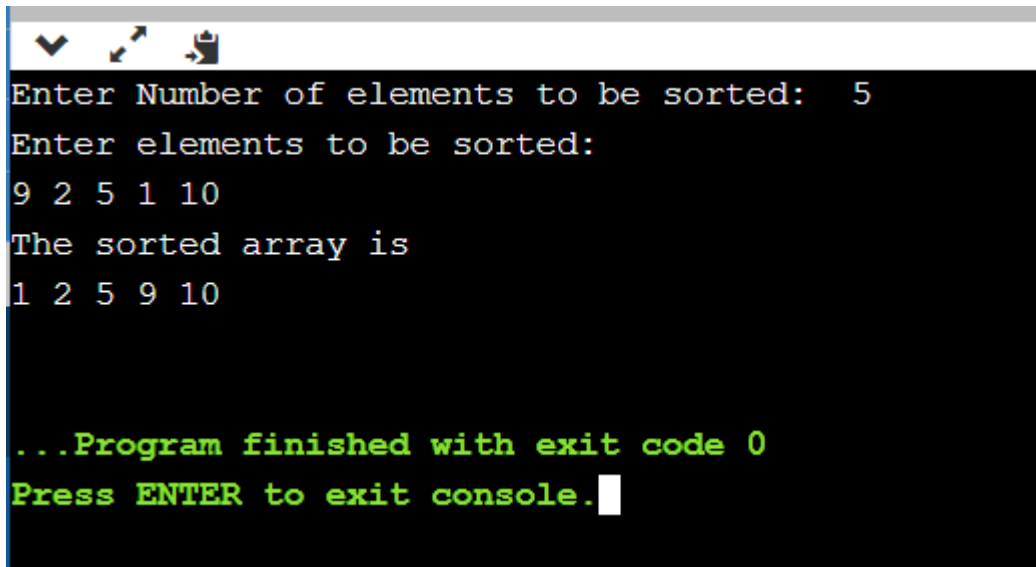
int main()
{
    int n, i;
    printf("Enter Number of elements to be sorted: \t");
    scanf("%d",&n);
    printf("Enter elements to be sorted: \n");
    int arr[n];
    for(i=0;i<n;i++)
    {
        scanf("%d",&arr[i]);
    }

    insertionSort(arr, n);
    printf("The sorted array is \n");
    printArray(arr, n);

    return 0;
}

```

OUTPUT:

A terminal window with a dark background and light green text. The window has a title bar with standard OS icons. The text inside shows a C program running. It prompts for the number of elements to be sorted (5), then the elements themselves (9 2 5 1 10). It then displays the sorted array (1 2 5 9 10). Finally, it shows the program finished with exit code 0 and prompts the user to press ENTER to exit the console.

```
Enter Number of elements to be sorted: 5
Enter elements to be sorted:
9 2 5 1 10
The sorted array is
1 2 5 9 10

...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION: We learnt about Insertion Sort and implemented that in a C Program.

MODULE - 2

MODULE NAME: Dynamic Programming Approach

EXPERIMENT - 2.1

AIM: Single Source Shortest Path (Bellman-Ford Algorithm)

COMPLEXITY:

- Worst case time complexity: $\theta(VE)$
- Average case time complexity: $\theta(VE)$
- Best case time complexity: $\theta(E)$
- Space complexity: $\theta(V)$

THEORY:

The Bellman-Ford algorithm is a graph search algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. This algorithm can be used on both weighted and unweighted graphs.

Like Dijkstra's shortest path algorithm, the Bellman-Ford algorithm is guaranteed to find the shortest path in a graph. Though it is slower than Dijkstra's algorithm, Bellman-Ford is capable of handling graphs that contain negative edge weights, so it is more versatile. It is worth noting that if there exists a negative cycle in the graph, then there is no shortest path. Going around the negative cycle an infinite number of times would continue to decrease the cost of the path (even though the path length is increasing). Because of this, Bellman-Ford can also detect negative cycles which is a useful feature.

ALGORITHM:

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

.....a) Do following for each edge u-v

.....If dist[v] > dist[u] + weight of edge uv, then update dist[v]

.....dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

.....If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

PSEUDOCODE :

```
Bellman-Ford(int v)
    d[i] = inf for each vertex i
    d[v] = 0
    for step = 1 to n
        for all edges like e
            i = e.first // first end
            j = e.second // second end
            w = e.weight
            if d[j] > d[i] + w
                if step == n
                    then return "Negative cycle found"
                d[j] = d[i] + w
```

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

struct Edge

{

    int src, dest, weight;

};

struct Graph

{

    // V-> Number of vertices, E-> Number of edges

    int V, E;

    struct Edge* edge;

};

// Creates a 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( graph->E * sizeof( struct Edge
) );

return graph;
}

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

        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to all other
// vertices using Bellman-Ford algorithm. The function also detects
// negative

// weight cycle

void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;

    int E = graph->E;

```

```

int dist[V];

int i,j;

// Step 1: Initialize distances from src to all other vertices as
INFINITE

for (i = 0; i < V; i++)

    dist[i]    = INT_MAX;

dist[src] = 0;


// Step 2: Relax all edges |V| - 1 times. A simple shortest path
from src

// to any other vertex can have at-most |V| - 1 edges

for (i = 1; i <= V-1; i++)
{
    for (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] + weight < dist[v])

            dist[v] = dist[u] + weight;
    }
}


// Step 3: check for negative-weight cycles. The above step
guarantees

```

```

    // shortest distances if graph doesn't contain negative weight
    cycle.

    // If we get a shorter path, then there is a cycle.

    for (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] + weight < dist[v])

            printf("Graph contains negative weight cycle");

    }

    printArr(dist, V);

    return;
}

int main()
{
    int V = 10; // Number of vertices in graph

    int E = 18; // Number of edges in graph

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

```



```

    struct Graph* graph = createGraph(V, E);

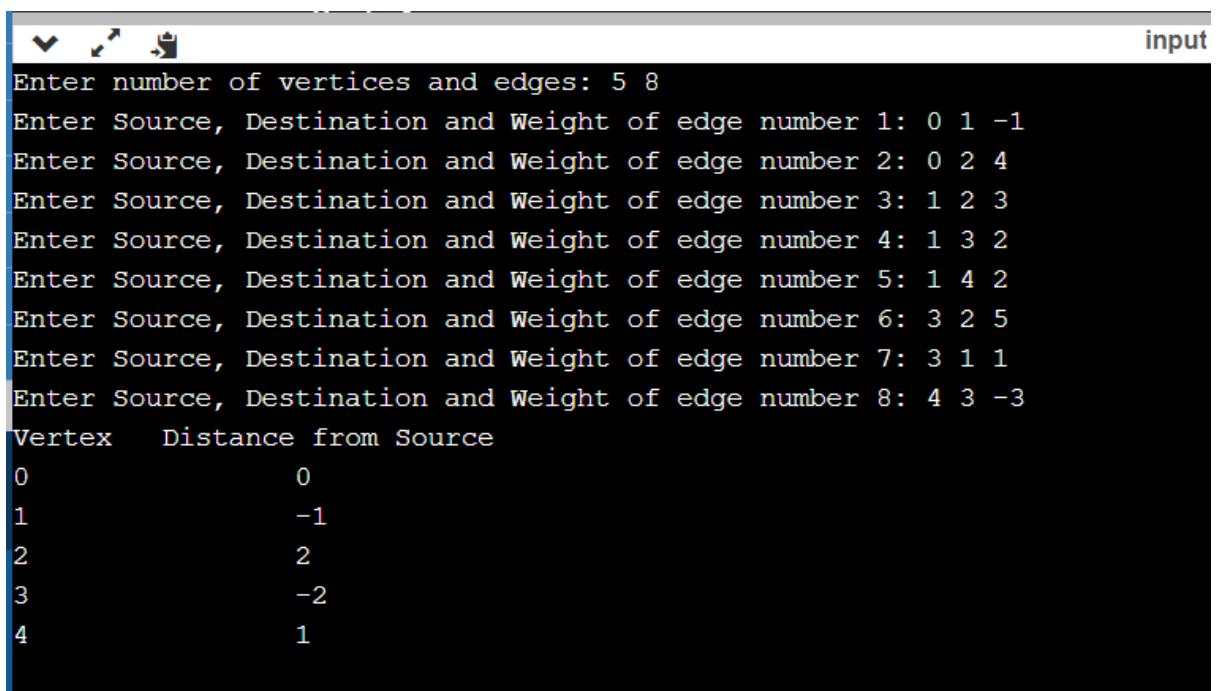
    int i;
    for(i=0;i<E;i++){
        printf("Enter Source, Destination and Weight of edge number %d:
",i+1);
        scanf("%d",&graph->edge[i].src);
        scanf("%d",&graph->edge[i].dest);
        scanf("%d",&graph->edge[i].weight);
    }

    BellmanFord(graph, 0);

    return 0;
}

```

OUTPUT:



```

input
Enter number of vertices and edges: 5 8
Enter Source, Destination and Weight of edge number 1: 0 1 -1
Enter Source, Destination and Weight of edge number 2: 0 2 4
Enter Source, Destination and Weight of edge number 3: 1 2 3
Enter Source, Destination and Weight of edge number 4: 1 3 2
Enter Source, Destination and Weight of edge number 5: 1 4 2
Enter Source, Destination and Weight of edge number 6: 3 2 5
Enter Source, Destination and Weight of edge number 7: 3 1 1
Enter Source, Destination and Weight of edge number 8: 4 3 -3
Vertex    Distance from Source
0          0
1         -1
2          2
3         -2
4          1

```

CONCLUSION: We learnt about Bellman-Ford algorithm and implemented that in a C Program.

EXPERIMENT - 2.2

AIM: All Pair Shortest Path [Floyd Warshall]

COMPLEXITY:

- Worst case time complexity: $\theta(V^3)$
- Average case time complexity: $\theta(V^3)$
- Best case time complexity: $\theta(V^3)$
- Space complexity: $\theta(V^2)$

THEORY:

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

ALGORITHM:

We initialize the solution matrix the same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in the shortest path from i to j . We keep the value of $\text{dist}[i][j]$ as it is.

2) k is an intermediate vertex in the shortest path from i to j . We update the value of $\text{dist}[i][j]$ as $\text{dist}[i][k] + \text{dist}[k][j]$ if $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$

PSEUDOCODE:

```
Floyd-Warshall()
    d[v][u] = infinity for each pair (v,u)
    d[v][v] = 0 for each vertex v
    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                d[i][j] = min(d[i][j], d[i][k] + d[k][j])
```

CODE:

```
#include<stdio.h>

#define V 4

#define INF 99999

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

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

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

```

    printSolution(dist);
}

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

    int graph[V][V]/* = { {0, 5, INF, 10},
                           {INF, 0, 3, INF},
                           {INF, INF, 0, 1},
                           {INF, INF, INF, 0}
                           }*/;

    int i,j,m,n,value;
    printf("Enter the matrix size : (m x n) :");
    scanf("%d x %d", &m, &n);

    printf("Enter 99999 for INF \n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {

```

```

        printf("Enter value for %d x %d: ", i+1 ,j+1);
        scanf("%d", &graph[i][j]);
    }
}

floydWarshall(graph);
return 0;
}

```

OUTPUT:

```

Enter the matrix size : (m x n) :4 x 4
Enter 99999 for INF
Enter value for 1 x 1: 0
Enter value for 1 x 2: 5
Enter value for 1 x 3: 99999
Enter value for 1 x 4: 10
Enter value for 2 x 1: 99999
Enter value for 2 x 2: 0
Enter value for 2 x 3: 3
Enter value for 2 x 4: 99999
Enter value for 3 x 1: 99999
Enter value for 3 x 2: 99999
Enter value for 3 x 3: 0
Enter value for 3 x 4: 1
Enter value for 4 x 1: 99999
Enter value for 4 x 2: 99999
Enter value for 4 x 3: 99999
Enter value for 4 x 4: 0
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

```

CONCLUSION: We learnt about Flyod Warshall and implemented that in a C Program.

MODULE – 3

MODULE NAME: Greedy Approach

EXPERIMENT 3.1

AIM: Single Source Shortest Path [Dijkstra]

COMPLEXITY:

- Worst case time complexity: $\Theta(E+V \log V)$
- Average case time complexity: $\Theta(E+V \log V)$
- Best case time complexity: $\Theta(E+V \log V)$
- Space complexity: $\Theta(V)$

THEORY:

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with a given source as root. We maintain two sets, one set contains vertices included in the shortest path tree, the other set includes vertices not yet included in the shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

ALGORITHM:

Set all vertices distances = infinity **except for** the source vertex, set the source distance = 0.

Push the source vertex **in** a min-priority queue **in** the form (distance , vertex), **as** the comparison **in** the min-priority queue will be according to vertices distances.

Pop the vertex **with** the minimum distance **from** the priority queue (at first the popped vertex = source).

Update the distances of the connected vertices to the popped vertex **in** case of "**current vertex distance + edge weight < next vertex distance**", then push the vertex **with** the new distance to the priority queue.

If the popped vertex **is** visited before, just **continue** without using it.

Apply the same algorithm again until the priority queue **is** empty.

PSEUDOCODE:

```
dijkstra(v) :
    d[i] = inf for each vertex i
    d[v] = 0
    s = new empty set
    while s.size() < n
        x = inf
        u = -1
        for each i in V-s //V is the set of vertices
            if x >= d[i]
                then x = d[i], u = i
        insert u into s
        // The process from now is called Relaxing
        for each i in adj[u]
            d[i] = min(d[i], d[u] + w(u,i))
```

CODE:

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10
```

```

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()
{
    int G[MAX][MAX],i,j,n,u;
    printf("Enter no. 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",&G[i][j]);

    printf("\nEnter the starting node:");
    scanf("%d",&u);
    dijkstra(G,n,u);

    return 0;
}

void dijkstra(int G[MAX][MAX],int n,int startnode)
{
    int cost[MAX][MAX],distance[MAX],pred[MAX];
    int visited[MAX],count,mindistance,nextnode,i,j;

    //pred[] stores the predecessor of each node
    //count gives the number of nodes seen so far
    //create the cost matrix
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(G[i][j]==0)
                cost[i][j]=INFINITY;
            else
                cost[i][j]=G[i][j];

    //initialize pred[],distance[] and visited[]
    for(i=0;i<n;i++)
    {
        distance[i]=cost[startnode][i];
        pred[i]=startnode;
        visited[i]=0;
    }

    distance[startnode]=0;

```



```

visited[startnode]=1;
count=1;

while(count<n-1)
{
    mindistance=INFINITY;

    //nextnode gives the node at minimum distance
    for(i=0;i<n;i++)
        if(distance[i]<mindistance&&!visited[i])
        {
            mindistance=distance[i];
            nextnode=i;
        }

    //check if a better path exists through nextnode
    visited[nextnode]=1;
    for(i=0;i<n;i++)
        if(!visited[i])
            if(mindistance+cost[nextnode][i]<distance[i])
            {
                distance[i]=mindistance+cost[nextnode][i];
                pred[i]=nextnode;
            }
    count++;
}

//print the path and distance of each node
for(i=0;i<n;i++)
    if(i!=startnode)
    {
        printf("\nDistance of node%d=%d",i,distance[i]);
        printf("\nPath=%d",i);

        j=i;
        do
        {
            j=pred[j];
            printf("<-%d",j);
        }while(j!=startnode);
    }
}

```

OUTPUT:

```
Enter no. of vertices:7

Enter the adjacency matrix:
0 0 1 2 0 0 0
0 0 2 0 0 3 0
1 2 0 1 3 0 0
2 0 1 0 0 0 1
0 0 3 0 0 2 0
0 3 0 0 2 0 1
0 0 0 1 0 1 0

Enter the starting node:0

Distance of node1=3
Path=1<-2<-0
Distance of node2=1
Path=2<-0
Distance of node3=2
Path=3<-0
Distance of node4=4
Path=4<-2<-0
Distance of node5=4
Path=5<-6<-3<-0
Distance of node6=3
Path=6<-3<-0
```

CONCLUSION: We learnt about Dijkstra's algorithm and implemented that in a C Program.

EXPERIMENT - 3.2

AIM: Knapsack Problem

COMPLEXITY:

- Time complexity: $\Theta(n*W)$
- Space complexity: $\Theta(n*W)$

THEORY:

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays $val[0..n-1]$ and $wt[0..n-1]$ which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of $val[]$ such that sum of the weights of this subset is smaller than or equal to W . You cannot break an item, either pick the complete item or don't pick it (0-1 property).

In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a $DP[][]$ table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.

The state $DP[i][j]$ will denote maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider ' w_i ' (weight in 'ith' row) we can fill it in all columns which have 'weight values $> w_i$ '. Now two possibilities can take place:

- Fill ' w_i ' in the given column.
- Do not fill ' w_i ' in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill 'ith' weight in 'jth' column then $DP[i][j]$ state will be same as $DP[i-1][j]$ but if we fill the weight, $DP[i][j]$ will be equal to the value of ' w_i ' + value of the column weighing ' $j-w_i$ ' in the previous row. So we take the maximum of these two possibilities to fill the current state.

ALGORITHM:

Step 1: Node root represents the initial state of the knapsack, where you have **not** selected any package.

- $TotalValue = 0$.
- The upper bound of the root node $UpperBound = M * \text{Maximum unit cost}$.

Step 2: Node root will have child nodes corresponding to the ability to select the package **with** the largest unit cost. For each node, you re-calculate the parameters:

- $TotalValue = TotalValue \text{ (old)} + \text{number of selected packages} * \text{value of each package}$.
- $M = M \text{ (old)} - \text{number of packages selected} * \text{weight of each package}$.
- $UpperBound = TotalValue + M \text{ (new)} * \text{The unit cost of the package to be considered next}$.

Step 3: In child nodes, you will prioritize branching **for** the node having the larger upper bound. The children of this node correspond to the ability of selecting the next package having large unit cost. For each node, you must re-calculate the parameters $TotalValue$, M , $UpperBound$ according to the formula mentioned **in** step 2.

Step 4: Repeat Step 3 **with** the note: **for** nodes **with** upper bound **is** lower **or** equal values to the temporary maximum cost of an option found, you do **not** need to branch **for** that node anymore.

Step 5: If all nodes are branched **or** cut off, the most expensive option **is** the one to look **for**.

PSEUDOCODE:

```
maxProfit = 0
for i = 0 to 2^N:
    bin = binary(i)           // Convert the number to binary
    profit = 0
    weight = 0
    for j = 0 to bin.length():
        if bin[j] is set:     // If the bit j is set, we have
to include that item.
            if weight + wt[j] > W: // When weight of the
combination exceeds Capacity, Break.
                profit = 0
                break
            profit = profit + val[j]
            weight = weight + wt[j]
    maxProfit = max(maxProfit, profit) // Update max profit.
```

CODE:

```
#include<stdio.h>
#include<time.h>
#include<conio.h>
void knapsack(float capacity, int n, float weight[], float profit[])
{
    float x[20], totalprofit,y;
    int i,j;
    y=capacity;
    totalprofit=0;
    for(i=0;i < n;i++)
        x[i]=0.0;
    for(i=0;i < n;i++)
    {
        if(weight[i] > y)
            break;
        else
        {
            x[i]=1.0;
            totalprofit=totalprofit+profit[i];
            y=y-weight[i];
        }
    }
    if(i < n)
        x[i]=y/weight[i];
    totalprofit=totalprofit+(x[i]*profit[i]);
```

```

printf("The selected elements are:-\n ");
for(i=0;i < n;i++)
    if(x[i]==1.0)
        printf("\nProfit is %f with weight %f ", profit[i],
weight[i]);
    else if(x[i] > 0.0)
        printf("\n%f part of Profit %f with weight %f", x[i],
profit[i], weight[i]);
    printf("\nTotal profit for %d objects with capacity %f = %f\n\n", n,
capacity,totalprofit);
}
void main()
{
    float weight[20],profit[20],ratio[20], t1,t2,t3;
    int n;
    time_t start,stop;
    float capacity;
    int i,j;
    printf("Enter number of objects: ");
    scanf("%d", &n);
    printf("\nEnter the capacity of knapsack: ");
    scanf("%f", &capacity);
    for(i=0;i < n;i++)
    {
        printf("\nEnter %d(th) profit: ", (i+1));
        scanf("%f", &profit[i]);
        printf("Enter %d(th) weight: ", (i+1));
        scanf("%f", &weight[i]);
        ratio[i]=profit[i]/weight[i];
    }
    start=time(NULL);
    for(i=0;i < n;i++)
        for(j=0;j < n;j++)
        {
            if(ratio[i] > ratio[j])
            {
                t1=ratio[i];
                ratio[i]=ratio[j];
                ratio[j]=t1;
                t2=weight[i];
                weight[i]=weight[j];
                weight[j]=t2;
                t3=profit[i];
                profit[i]=profit[j];
                profit[j]=t3;
            }
        }
}

```

```

    }
    knapsack(capacity,n,weight,profit);
    stop=time(NULL);
    printf("\nKnapsack = %f\n", difftime(stop,start));
    getch();
}

```

OUTPUT:

```

Enter number of objects: 5

Enter the capacity of knapsack: 10

Enter 1(th)  profit: 9
Enter 1(th)  weight: 6

Enter 2(th)  profit: 15
Enter 2(th)  weight: 3

Enter 3(th)  profit: 20
Enter 3(th)  weight: 2

Enter 4(th)  profit: 8
Enter 4(th)  weight: 4

Enter 5(th)  profit: 10
Enter 5(th)  weight: 3
The selected elements are:-

Profit is 20.000000 with weight 2.000000
Profit is 15.000000 with weight 3.000000
Profit is 10.000000 with weight 3.000000
0.500000 part of Profit 8.000000 with weight 4.000000
Total profit for 5 objects with capacity 10.000000 = 49.000000

Knapsack = 0.000000

```

CONCLUSION: We learnt about the Knapsack problem and implemented that in a C Program.

MODULE - 4

MODULE NAME: Backtracking and String Matching

EXPERIMENT - 4.1

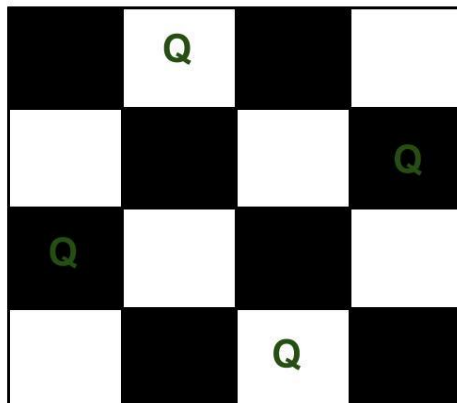
AIM: N Queen Problem

COMPLEXITY:

Time Complexity: $O(N!)$.

THEORY:

The N Queen is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other. For example, the following is a solution for the 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

```
{ 0,  1,  0,  0}  
{ 0,  0,  0,  1}  
{ 1,  0,  0,  0}  
{ 0,  0,  1,  0}
```


ALGORITHM:

- 1) Start **in** the leftmost column
- 2) If all queens are placed
 return true
- 3) Try all rows **in** the current column.
 Do following **for** every tried row.
 - a) If the queen can be placed safely **in** this row then mark this [row, column] **as** part of the solution **and** recursively check **if** placing queen here leads to a solution.
 - b) If placing the queen **in** [row, column] leads to a solution then **return** true.
 - c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
- 3) If all rows have been tried and nothing worked, **return** false to trigger backtracking.

PSEUDOCODE:

```
START
1. begin from the leftmost column
2. if all the queens are placed,
    return true/ print configuration
3. check for all rows in the current column
    a) if queen placed safely, mark row and column; and
       recursively check if we approach in the current
       configuration, do we obtain a solution or not
    b) if placing yields a solution, return true
    c) if placing does not yield a solution, unmark and
       try other rows
4. if all rows tried and solution not obtained, return
   false and backtrack
END
```

CODE:

```
#include <stdio.h>

int N;

int board[100][100];

int is_attackable(int i,int j)
```

```

{
    int k,l;
    //checking if there is a queen in row or column
    for(k=0;k<N;k++)
    {
        if((board[i][k] == 1) || (board[k][j] == 1))
            return 1;
    }
    //checking for diagonals
    for(k=0;k<N;k++)
    {
        for(l=0;l<N;l++)
        {
            if(((k+l) == (i+j)) || ((k-l) == (i-j)))
            {
                if(board[k][l] == 1)
                    return 1;
            }
        }
    }
    return 0;
}

int N_queen(int n)
{
    int i,j;
    if(n==0)
        return 1;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            if((!is_attackable(i,j)) && (board[i][j]!=1))
            {
                board[i][j] = 1;

                if(N_queen(n-1)==1)
                {
                    return 1;
                }
                board[i][j] = 0;
            }
        }
    }
    return 0;
}

```

```

}

int main()
{

    printf("Enter the value of N for NxN chessboard\n");
    scanf("%d",&N);

    int i,j;
    //setting all elements to 0
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            board[i][j]=0;
        }
    }
    //calling the function
    N_queen(N);
    //printing the matix
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
            printf("%d\t",board[i][j]);
        printf("\n");
    }

}

```

OUTPUT:

```

Enter the value of N for NxN chessboard
4
0      1      0      0
0      0      0      1
1      0      0      0
0      0      1      0

...Program finished with exit code 0
Press ENTER to exit console.

```

CONCLUSION: We learnt about the N Queen Problem and implemented that in a C Program.

EXPERIMENT - 4.2

AIM: String Matching Algorithm [KMP Algorithm]

COMPLEXITY:

Complexity of the overall algorithm is $O(n + k)$.

THEORY:

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm declared as "this is the method to find a place where one or several strings are found within the larger string."

Given a text array, $T[1.....n]$, of n character and a pattern array, $P[1.....m]$, of m characters. The problems are to find an integer s , called a valid shift where $0 \leq s < n-m$ and $T[s+1.....s+m] = P[1.....m]$. In other words, to find even if P in T , i.e., where P is a substring of T . The item of P and T are character drawn from some finite alphabet such as $\{0, 1\}$ or $\{A, B, ..., Z, a, b, ..., z\}$. Given a string $T[1.....n]$, the substrings are represented as $T[i.....j]$ for some $0 \leq i \leq j \leq n-1$, the string formed by the characters in T from index i to index j , inclusive. This process that a string is a substring of itself (take $i = 0$ and $j = m$).

The proper substring of string $T[1.....n]$ is $T[1.....j]$ for some $0 < i \leq j \leq n-1$. That is, we must have either $i > 0$ or $j < m-1$.

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of $O(n)$ is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. **The Prefix Function (Π):** The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. **The KMP Matcher:** With string 'S,' pattern 'p' and prefix function ' Π ' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

ALGORITHM:

```
Begin
  n := size of text
  m := size of pattern
  call findPrefix(pattern, m, prefArray)

  while i < n, do
    if text[i] = pattern[j], then
      increase i and j by 1
    if j = m, then
      print the location (i-j) as there is the pattern
      j := prefArray[j-1]
    else if i < n AND pattern[j] ≠ text[i] then
      if j ≠ 0 then
        j := prefArray[j - 1]
      else
        increase i by 1
  done
End
```

PSEUDOCODE:

```
COMPUTE- PREFIX- FUNCTION (P)
1. m ← length [P]           //'p' pattern to be matched
2.  $\Pi[1] \leftarrow 0$ 
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P[k + 1] ≠ P[q]
6. do k ←  $\Pi[k]$ 
7. If P[k + 1] = P[q]
8. then k ← k + 1
9.  $\Pi[q] \leftarrow k$ 
10. Return  $\Pi$ 
```

```
KMP-MATCHER (T, P)
1. n ← length [T]
2. m ← length [P]
3.  $\Pi \leftarrow$  COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0           // numbers of characters matched
5. for i ← 1 to n   // scan S from left to right
6. do while q > 0 and P[q + 1] ≠ T[i]
7. do q ←  $\Pi[q]$      // next character does not match
8. If P[q + 1] = T[i]
9. then q ← q + 1    // next character matches
10. If q = m         // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ←  $\Pi[q]$        // look for the next match
```

CODE:

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

int main()
{
    char string[100], matchcase[20], c;
    int i = 0, j = 0, index;

    /*Scanning string*/
    printf("Enter string: ");
    do
    {
        fflush(stdin);
        c = getchar();
        string[i++] = tolower(c);
```

```

    } while (c != '\n');
    string[i - 1] = '\0';
    /*Scanning substring*/
    printf("Enter substring: ");
    i = 0;
    do
    {
        fflush(stdin);
        c = getchar();
        matchcase[i++] = tolower(c);
    } while (c != '\n');
    matchcase[i - 1] = '\0';
    for (i = 0; i < strlen(string) - strlen(matchcase) + 1; i++)
    {
        index = i;
        if (string[i] == matchcase[j])
        {
            do
            {
                i++;
                j++;
            } while(j != strlen(matchcase) && string[i] == matchcase[j]);
            if (j == strlen(matchcase))
            {
                printf("Match found from position %d to %d.\n", index + 1,
i);

                return 0;
            }
            else
            {
                i = index + 1;
                j = 0;
            }
        }
    }
    printf("No substring match found in the string.\n");

    return 0;
}

```


OUTPUT:

```
Enter string: programming
Enter substring: program
Match found from position 1 to 7.

...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION: We learnt about the KMP Algorithm and implemented that in a C Program.