Name: -Ayush Jain
SAP ID: 60004200132
Batch: B2
Analysis Of Algorithm

Experiment – 1

Aim: - Write a program to implement and analyze the time complexity of Insertion Sort and Selection Sort.

Theory: -

Insertion Sort:
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.
Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithms with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e., it is appropriate for data sets that are already partially sorted.

Complexity Analysis of Insertion Sort: o Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is O(n). o Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is O(n^2). o Worst-Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is O(n^2).

Selection Sort:
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.

- The subarray which is already sorted.
- Remaining subarray which is unsorted.

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

Complexity Analysis of Selection Sort: o Best Case Complexity: The selection sort algorithm has a best-case time complexity of $O(n^2)$ for the already sorted array.

- o Average Case Complexity: The average-case time complexity for the selection sort algorithm is $O(n^2)$, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- o Worst Case Complexity: The worst-case time complexity is also $O(n^2)$, which occurs when we sort the descending order of an array into the ascending order.

Algorithm: -

Insertion Sort:

```
Algorithm 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
```

Selection Sort:

```
Algoritm selectionsort
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
```

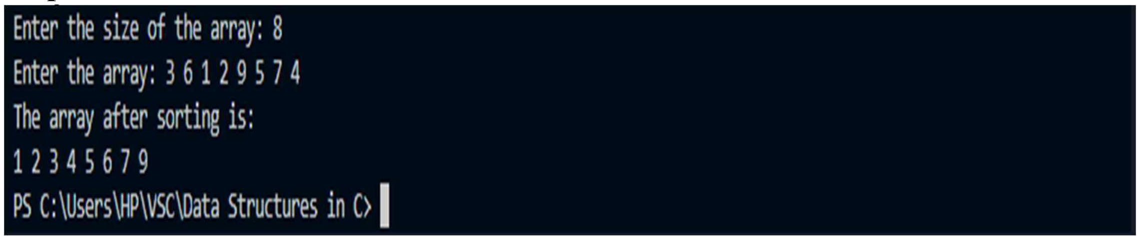
<u>Code</u>: -

Insertion Sort:

```
#include <stdio.h>

void insertion_sort(int a[], int n)
{     int i, j, key;     for(i
= 1; i < n; i ++)
    {        key = a[i];         for(j = i - 1; j >= 0
&& a[j] > key; j --)
        {
            a[j + 1] = a[j];
        }
        a[j + 1] = key;
    }
}

int main()
{     int n,
i;
    printf("Enter the size of the array: ");
scanf("%d", &n);
     int a[n];    printf("Enter the
array: ");      for(i = 0; i < n; i
++)
    {
        scanf("%d", &a[i]);
    }

    insertion_sort(a, n);     printf("The array
after sorting is: \n");     for(i = 0; i < n; i
++)
```

```
    {        printf("%d ",
a[i]);
    }
}
```

Output: -

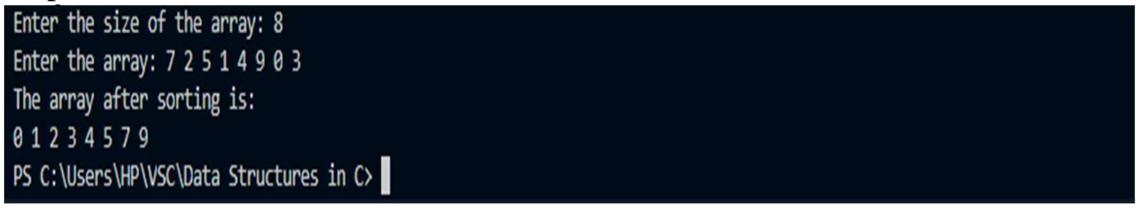


```
Enter the size of the array: 8
Enter the array: 3 6 1 2 9 5 7 4
The array after sorting is:
1 2 3 4 5 6 7 9
PS C:\Users\HP\VSC\Data Structures in C>
```

Selection Sort:

```
#include <stdio.h>

void selection_sort(int a[], int n)
{    int i, j, pos, temp;
for(i = 0; i < n - 1; i ++)
    {        pos
= i;
        for(j = i + 1; j < n; j ++)
        {            if(a[pos]
> a[j])
            {
                pos = j;
            }
}
        if(pos != i)
        {            temp =
a[i];            a[i] =
a[pos];            a[pos] =
temp;
        }
    }
}

int main()
{    int n, i;    printf("Enter the size of
the array: ");    scanf("%d", &n);
     int a[n];    printf("Enter the
array: ");     for(i = 0; i < n; i
++)
    {
        scanf("%d", &a[i]);
    }
    selection_sort(a, n);     printf("The array
after sorting is: \n");     for(i = 0; i < n; i
++)
    {
        printf("%d ", a[i]);
    }
}
```

Output: -



```
Enter the size of the array: 8
Enter the array: 7 2 5 1 4 9 0 3
The array after sorting is:
0 1 2 3 4 5 7 9
PS C:\Users\HP\VSC\Data Structures in C>
```

<u>Conclusion</u>: -

Learned about the time complexities of insertion sort and selection sort and implemented them.

Experiment – 2

Aim: - Write a program to implement and analyze the time complexity of Merge Sort and Quick Sort.

Theory: -

Merge Sort:
Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. It divides the given list into two equal halves, calls itself for the two halves, and then merges the two sorted halves. We have to define the merge() function to perform the merging.
The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs are merged into the four-element lists, and so on until we get the sorted list.

Complexity Analysis of Merge Sort: o Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is O(n*logn). o Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is O(n*logn). o Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is O(n*logn).

Quick Sort:
Quicksort is the widely used sorting algorithm that makes n log n comparisons in the average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquers approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.
Conquer: Recursively, sort two subarrays with Quicksort.
Combine: Combine the already sorted array.
Quicksort picks an element as a pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Complexity Analysis of Quick Sort: o Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is O(n*logn). o Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is O(n*logn). o Worst-Case Complexity - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is O(n^2).

Algorithm: -

Merge Sort:

```
Algorithm mergesort( var a as array )
  if ( n == 1 ) return a

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]

  l1 = mergesort( l1 )
l2 = mergesort( l2 )

  return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

  var c as array
  while ( a and b have elements )
if ( a[0] > b[0] )        add b[0] to
the end of c
       remove b[0] from b
else
       add a[0] to the end of c
remove a[0] from a      end if
  end while

  while ( a has elements )
add a[0] to the end of c
remove a[0] from a
  end while
  while ( b has elements )
add b[0] to the end of c
    remove b[0] from b
  end while

  return c

end
```

Quick Sort:

```
function partitionFunc(left, right, pivot)
  leftPointer = left
  rightPointer = right - 1

  while True do       while
A[++leftPointer] < pivot do
      //do-nothing
end while

    while rightPointer > 0 && A[--rightPointer] > pivot do
      //do-nothing
    end while

    if leftPointer >= rightPointer
      break
    else
      swap leftPointer,rightPointer
    end if

end while

  swap leftPointer,right
  return leftPointer

end function
```

```
Algorithm quickSort(left, right)

  if right-left <= 0
return
  else           pivot = A[right]       partition =
partitionFunc(left, right, pivot)
quickSort(left,partition-1)
```

```
      quickSort(partition+1,right)
    end if

end
```

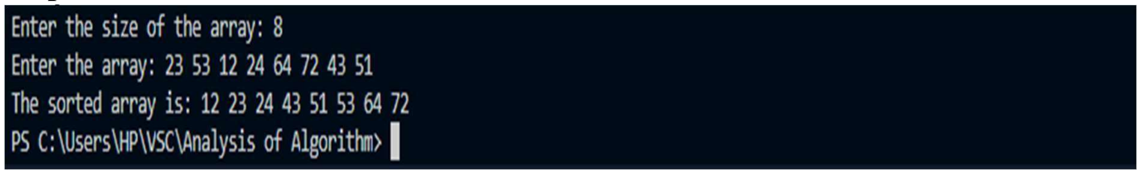
<u>Code</u>: -

Merge Sort:

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

void merge(int arr[], int l, int m, int r)
{     int i, j,
k;
    int n1 = m - l + 1;     int
n2 = r - m;

    int L[20], R[20];

    for (i = 0; i < n1; i++)
    {
        L[i] = arr[l + i];
    }
    for (j = 0; j < n2; j++)
    {
        R[j] = arr[m + 1+ j];
    }      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++;
    }

    while (i < n1)
    {         arr[k] =
L[i];         i++;
k++;
```

```
    }

    while (j < n2)
    {         arr[k] =
R[j];         j++;
k++;
    }
}

void mergeSort(int arr[], int l, int r)
{     if (l <
r)
    {
        int m = l+(r-l)/2;

        mergeSort(arr, l, m);
mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

int main()
{     int i, n, a[20];     printf("Enter the size
of the array: ");     scanf("%d", &n);

    printf("Enter the array: ");
for(i = 0; i < n; i ++)
    {
        scanf("%d", &a[i]);
    }

    mergeSort(a, 0, n - 1);

    printf("The sorted array is: ");
for(i = 0; i < n; i ++)
    {         printf("%d ",
a[i]);
    }
}
```

Output: -



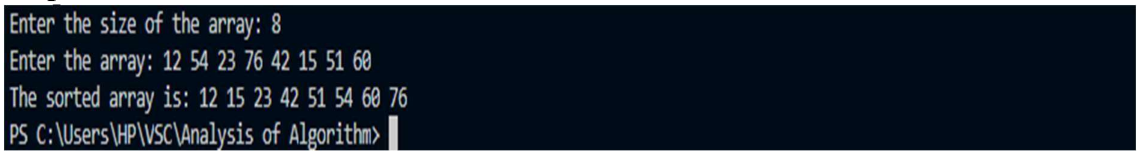
```
Enter the size of the array: 8
Enter the array: 23 53 12 24 64 72 43 51
The sorted array is: 12 23 24 43 51 53 64 72
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

Quick Sort:

```
#include<stdio.h>

void swap(int *a, int *b)
{    int t;    t
= *a;     *a
= *b;
   *b = t;
}

int partition(int a[], int low, int high)
{     int i, j, pivot;     pivot =
a[high];     i = low - 1;     for(j =
low; j <= high - 1; j ++)
   {        if(a[j] <
pivot)
       {            i ++;
swap(&a[i], &a[j]);
       }
   }
   swap(&a[i + 1], &a[high]);
return (i + 1);
}

void quicksort(int a[], int low, int high)
{
   if(low < high)
   {
      int par = partition(a, low, high);

      quicksort(a, low, par - 1);
quicksort(a, par + 1, high);
   }
}

int main()
{
   int i, n, a[20];
   printf("Enter the size of the array: ");
scanf("%d", &n);

   printf("Enter the array: ");
for(i = 0; i < n; i ++)
   {
      scanf("%d", &a[i]);
   }
```

```
    quicksort(a, 0, n - 1);

    printf("The sorted array is: ");
for(i = 0; i < n; i ++)
    {        printf("%d ",
a[i]);
    }
}
```

Output: -



```
Enter the size of the array: 8
Enter the array: 12 54 23 76 42 15 51 60
The sorted array is: 12 15 23 42 51 54 60 76
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

<u>Conclusion</u>: -

Learned about the time complexities of merge sort and quick sort and implemented them.

Experiment – 3

Aim: - Write a program to implement Single Source Shortest Path using Dynamic Programming.

Theory: -

Floyd-Warshall's Algorithm:
Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph. At first, the output matrix is the same as the given cost matrix of the graph. After that, the output matrix will be updated with all vertices k as the intermediate vertex.
The time complexity of this algorithm is O(V^3), where V is the number of vertices in the graph.

Bellman-Ford Algorithm:
Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

Algorithm: -

Floyd-Warshall's Algorithm:
Algorithm floydWarshal(cost)

```
  for k := 0 to n, do
for i := 0 to n, do
for j := 0 to n, do
         if cost[i,k] + cost[k,j] < cost[i,j], then
cost[i,j] := cost[i,k] + cost[k,j]           end if
end      end    end    display the current cost
matrix

end
```

Bellman-Ford Algorithm:
Algorithm bellmanFord(dist, pred, source)

```
  iCount := 1    maxEdge := n * (n - 1) / 2     //n is
number of vertices

  for all vertices v of the graph, do
    dist[v]  := ∞
pred[v]    :=    ϕ
end

  dist[source] := 0    eCount := number of edges
present in the graph    create edge list named
edgeList

  while iCount < n, do
for i := 0 to eCount, do
      if   dist[edgeList[i].v]   >   dist[edgeList[i].u]   +   (cost[u,v]   for   edge   i)
dist[edgeList[i].v] > dist[edgeList[i].u] + (cost[u,v] for edge i) pred[edgeList[i].v] :=
edgeList[i].u      end
  end

  iCount := iCount + 1    for all vertices i in the graph, do      if
dist[edgeList[i].v] > dist[edgeList[i].u] + (cost[u,v] for edge i),
then return true
    end if
  end

  return false

end
```

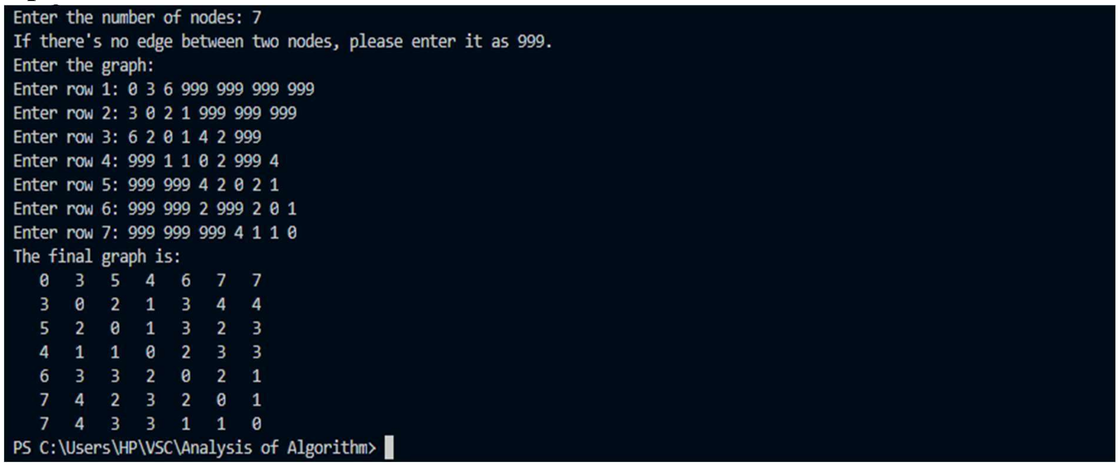
<u>Code</u>: -

Floyd-Warshall's Algorithm:

```
#include <stdio.h>

int V;

#define INF 999

void printMatrix(int matrix[][V])
{
  printf("The final graph is: \n");
for (int i = 0; i < V; i++)
  {
    for (int j = 0; j < V; j++)
    {          if (matrix[i][j]
== INF)
```

```
            {
               printf("%4s", "INF");
            }
else
            {                printf("%4d",
matrix[i][j]);
            }        }
printf("\n");
   }
}

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

   for (i = 0; i < V; i++)
for (j = 0; j < V; j++)
       {            matrix[i][j] =
graph[i][j];
       }

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

int main()
{     int i,
j;
   printf("Enter  the  number  of  nodes:  ");
scanf("%d", &V);     int graph[V][V];

   printf("If there's no edge between two nodes, please enter it as 999.\n");
printf("Enter the graph: \n");     for(i = 0; i < V; i ++)
   {
      printf("Enter row %d: ", (i + 1));
for(j = 0; j < V; j ++)
       {
           scanf("%d", &graph[i][j]);
       }
   }

   floydWarshall(graph);
}
```

Output: -



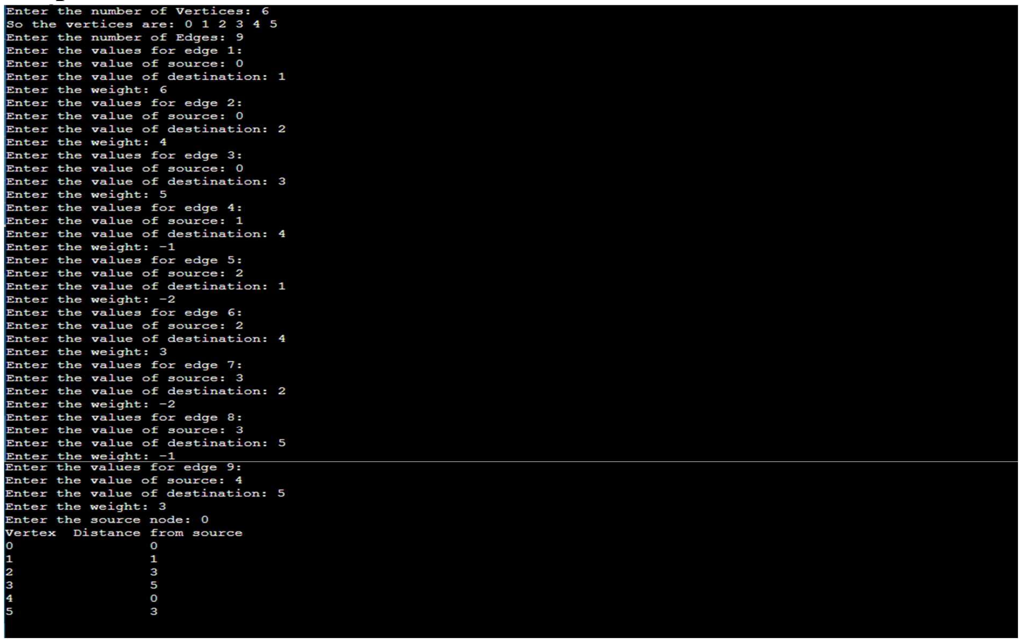
```
Enter the number of nodes: 7
If there's no edge between two nodes, please enter it as 999.
Enter the graph:
Enter row 1: 0 3 6 999 999 999 999
Enter row 2: 3 0 2 1 999 999 999
Enter row 3: 6 2 0 1 4 2 999
Enter row 4: 999 1 1 0 2 999 4
Enter row 5: 999 999 4 2 0 2 1
Enter row 6: 999 999 2 999 2 0 1
Enter row 7: 999 999 999 4 1 1 0
The final graph is:
   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
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

Bellman-Ford Algorithm:

```
#include <stdio.h>
#include <malloc.h>
#define INFINITY 99999

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

struct Graph
{
    int V, E;     struct
Edge *edge;
};

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 bellman_ford(struct Graph *graph, int src)
```

```
{
   int  V  =  graph->V;
int   E   =   graph->E;
int dist[V];

   for (int i = 0; i < V; i++)
   {
      dist[i] = INFINITY;
   }

   dist[src] = 0;

   for (int i = 0; i < V - 1; i++)
   {        for (int 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] != INFINITY && dist[u] + weight < dist[v])
         {
            dist[v] = dist[u] + weight;
         }
      }
   }

   for (int 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] != INFINITY && dist[u] + weight < dist[v])        {
         // dist[v] = dist[u] + weight;             printf("The graph
contains negative weight cycle ");             return;
      }
   }

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

   return;
}

int main()
{     int i, V, E, source;     printf("Enter the
number of Vertices: ");     scanf("%d", &V);
printf("Sothe vertices are: ");     for (i = 0; i
< V; i++)
    {
        printf("%d ", i);
    }     printf("\n");     printf("Enter the
number of Edges: ");     scanf("%d", &E);

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

    for (i = 0; i < E; i++)
    {
        printf("Enter the values for edge %d: \n", (i + 1));
printf("Enter the value of source: ");         scanf("%d",
&graph->edge[i].src);         printf("Enter the value of
destination: ");         scanf("%d", &graph->edge[i].dest);
printf("Enter the weight: ");         scanf("%d", &graph-
>edge[i].weight);
    }
    printf("Enter the source node: ");     scanf("%d",
&source);

    bellman_ford(graph, source);     return
0;
}
```

Output: -



```
Enter the number of Vertices: 6
So the vertices are: 0 1 2 3 4 5
Enter the number of Edges: 9
Enter the values for edge 1:
Enter the value of source: 0
Enter the value of destination: 1
Enter the weight: 6
Enter the values for edge 2:
Enter the value of source: 0
Enter the value of destination: 2
Enter the weight: 4
Enter the values for edge 3:
Enter the value of source: 0
Enter the value of destination: 3
Enter the weight: 5
Enter the values for edge 4:
Enter the value of source: 1
Enter the value of destination: 4
Enter the weight: -1
Enter the values for edge 5:
Enter the value of source: 2
Enter the value of destination: 1
Enter the weight: -2
Enter the values for edge 6:
Enter the value of source: 2
Enter the value of destination: 4
Enter the weight: 3
Enter the values for edge 7:
Enter the value of source: 3
Enter the value of destination: 2
Enter the weight: -2
Enter the values for edge 8:
Enter the value of source: 3
Enter the value of destination: 5
Enter the weight: -1
Enter the values for edge 9:
Enter the value of source: 4
Enter the value of destination: 5
Enter the weight: 3
Enter the source node: 0
Vertex  Distance from source
0                0
1                1
2                3
3                5
4                0
5                3
```

Conclusion: -

Learned about different dynamic programs for finding single source shortest path and implemented them.

Experiment – 4

Aim: - Write a program to implement Longest Common Subsequence.

Theory: -

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences. If S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2.
In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.

Algorithm: -

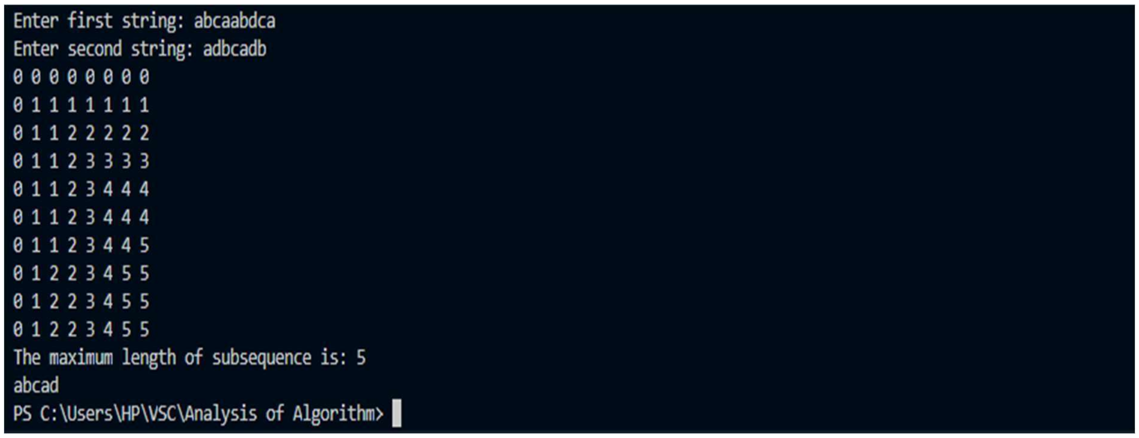
```
Algorithm: LCS-Length-Table-Formulation (X, Y)
m := length(X) n := length(Y) for i = 1 to m do C[i,
0] := 0 for j = 1 to n do C[0, j] := 0 for i = 1 to m do
for j = 1 to n do if xi = yj
C[i, j] := C[i - 1, j - 1] + 1
B[i, j] := 'D' else if
C[i -1, j] ≥ C[i, j -1]
C[i, j] := C[i - 1, j] + 1
B[i, j] := 'U' else
C[i, j] := C[i, j - 1]
B[i, j] := 'L' return
C and B
Algorithm: Print-LCS (B, X, i, j)
if i = 0 and j = 0
return if B[i, j]
= 'D'
Print-LCS(B, X, i-1, j-1)
Print(xi)
else  if  B[i,  j]  =  'U'
Print-LCS(B, X, i-1, j)
else
Print-LCS(B, X, i, j-1)
```

<u>Code</u>: -

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

void subsequence(char b[10][10], char c1[], int i, int j)
{
   if(i == 0 || j ==0)
   {
return;    }
   else if(b[i][j] == 'z')
   {
      subsequence(b, c1, i - 1, j - 1);
printf("%c", c1[i - 1]);
   }
   else if(b[i][j] == 'y')
   {
      subsequence(b, c1, i - 1, j);
   }
else    {
      subsequence(b, c1, i, j - 1);
   }
}
void lcs(char c1[], char c2[], int r, int c)
{    int i, j, a[10][10];
char b[10][10];    for(i
= 0; i <= r; i ++)
   {
      for(j = 0; j <= c; j ++)
      {
         if(i == 0 || j == 0)
         {
a[i][j] = 0;
b[i][j] = 0;
         }
         else if(c1[i - 1] == c2[j - 1])
         {
            a[i][j] = 1 + a[i - 1][j - 1];
            b[i][j] = 'z';
         }
else          {
            if(a[i - 1][j] > a[i][j - 1])
            {                a[i][j]
= a[i - 1][j];                b[i][j]
= 'y';
            }
else             {
```

```
                    a[i][j] = a[i][j - 1];
b[i][j] = 'x';
                }
            }
        }
    }
    int max = 0;      for(i =
0; i <= r; i ++)
    {
        for(j = 0; j <=c; j ++)
        {
            printf("%d ", a[i][j]);
            if(max < a[i][j])
            {
                max = a[i][j];
            }        }
printf("\n");
    }
    printf("The maximum length of subsequence is: %d", max);
printf("\n");

    subsequence(b, c1, r, c);
}
int main() {      char c1[10],
c2[10];     printf("Enter first
string: ");     scanf("%s", c1);
printf("Enter second string: ");
scanf("%s", c2);
    lcs(c1, c2, strlen(c1), strlen(c2));
}
```

<u>Output</u>: -



```
Enter first string: abcaabdca
Enter second string: adbcadb
0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1
0 1 1 2 2 2 2 2
0 1 1 2 3 3 3 3
0 1 1 2 3 4 4 4
0 1 1 2 3 4 4 4
0 1 1 2 3 4 4 5
0 1 2 2 3 4 5 5
0 1 2 2 3 4 5 5
0 1 2 2 3 4 5 5
The maximum length of subsequence is: 5
abcad
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

<u>Conclusion</u>: -

Learned about finding the longest common subsequence between two strings and implemented it.

Experiment – 5

Aim: - Write a program to implement Minimum Spanning Tree using Prim's and Kruskal's Algorithm.

Theory: -

Prim's Algorithm:
Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized. Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.
Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows - o First, we have to initialize an MST with the randomly chosen vertex.

- o Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree. o Repeat step 2 until the minimum spanning tree is formed.

Kruskal's Algorithm:
Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum. In Kruskal's algorithm, we start with edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows - o First, sort all the edges from low weight to high.

- o Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge. o Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

Algorithm: -

Prim’s Algorithm:
Algorithm Prims

```
  T = ∅;    U =
{ 1 };    while
(U ≠ V)
     let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
     T = T ∪ {(u, v)}
U = U ∪ {v}    end

end
```

Kruskal’s Algorithm:
Algorithm Kruskal(Graph)

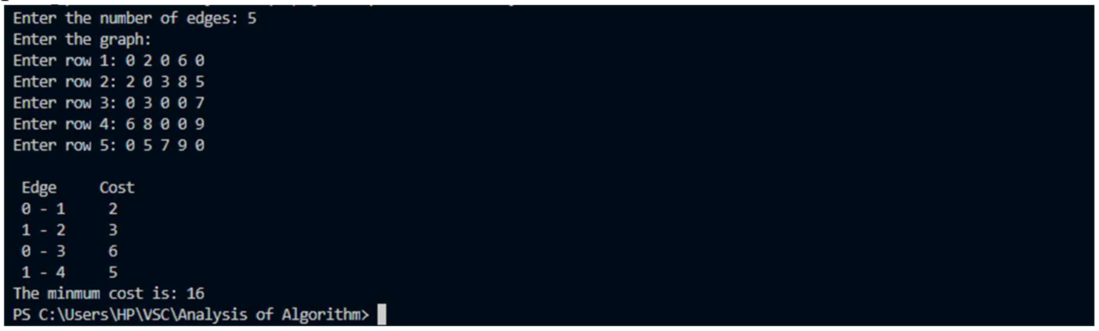
```
  create a new empty tree F    for
each vertex v in the Graph:
     make_set(v)    end    for each edge (u, v) in the Graph
ordered by weight (ascend):
     u_set  =  find_set(u)
v_set    =    find_set(v)
if u_set ≠ v_set:
        add the edge (u, v) in the tree F
union of two sets(u_set and v_set)
     end if
end
  return F

end
```

Code: -

Prim’s Algorithm:

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


int n = 5;


int minimum_key(int k[], int mst[])
{
   int minimum = INT_MAX, min,i;


   for (i = 0; i < n; i ++)
   {
        if (mst[i] == 0 && k[i] < minimum )
        {
            minimum = k[i];
min = i;
        }
    }
    return min;
}

void prim(int g[30][30])
{
    int  parent[n];              int  k[n];
int mst[n];          int i, count, edge,
v, cost = 0;     for (i = 0; i < n; i++)
    {
        k[i] = INT_MAX;
mst[i] = 0;
    }
    k[0] = 0;     parent[0] = -1;        for (count
= 0; count < n - 1; count ++)
    {
        edge = minimum_key(k, mst);
mst[edge] = 1;             for (v = 0; v <
n; v ++)
        {
            if (g[edge][v] && mst[v] == 0 && g[edge][v] < k[v])
            {
                parent[v] = edge;
k[v] = g[edge][v];
            }
        }
     }
     printf("\n Edge \t  Cost\n");
for (i = 1; i < n; i++)
     {
         printf(" %d - %d \t   %d \n", parent[i], i, g[i][parent[i]]);
cost += g[i][parent[i]];
     }
     printf("The minmum cost is: %d", cost);
}
int main()
{        int i, j, a[30][30];     printf("Enter
the number of edges: ");     scanf("%d",
&n);

    printf("Enter the graph: \n");
for(i = 0; i < n; i ++)
    {
     printf("Enter row %d: ", i + 1);
for(j = 0; j < n; j ++)
        {
            scanf("%d", &a[i][j]);
        }
    }
prim(a);
return 0;
}
```

Output: -



```
Enter the number of edges: 5
Enter the graph:
Enter row 1: 0 2 0 6 0
Enter row 2: 2 0 3 8 5
Enter row 3: 0 3 0 0 7
Enter row 4: 6 8 0 0 9
Enter row 5: 0 5 7 9 0

 Edge     Cost
 0 - 1     2
 1 - 2     3
 0 - 3     6
 1 - 4     5
The minmum cost is: 16
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

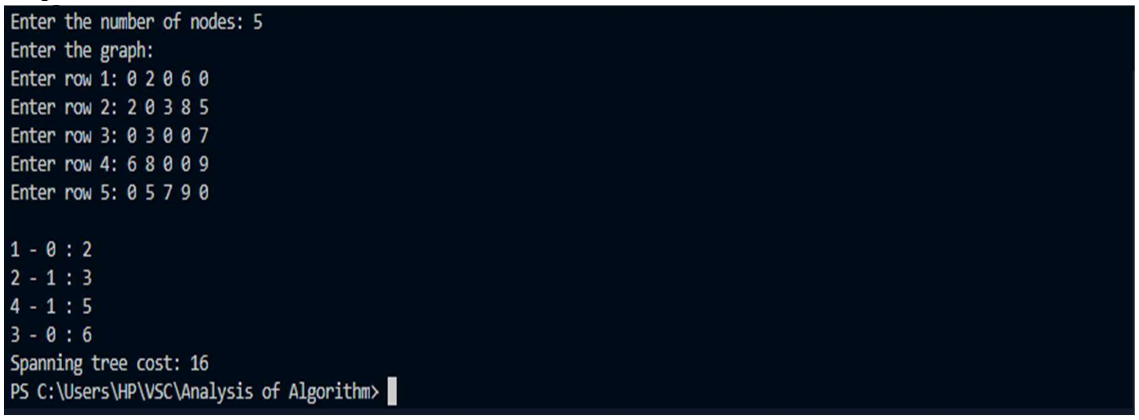
Kruskal's Algorithm:

```
#include <stdio.h>
#define MAX 30
typedef struct edge
{
    int u, v, w;
} edge;

typedef struct edge_list
{
    edge data[MAX];
    int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void applyUnion(int belongs[], int c1, int c2)
{     int i;      for (i = 0; i
< n; i++)
    {
        if (belongs[i] == c2)
        {
            belongs[i] = c1;
        }
    } }
```

```
void sort()
{    int i,
j;
   edge temp;

   for (i = 1; i < elist.n; i++)
   {        for (j = 0; j < elist.n - 1;
j++)
       {
          if (elist.data[j].w > elist.data[j + 1].w)
          {               temp = elist.data[j];
elist.data[j] = elist.data[j + 1];
elist.data[j + 1] = temp;
          }
       }
   }
}

void kruskalAlgo()
{
   int belongs[MAX], i, j, cno1, cno2;
elist.n = 0;

   for (i = 1; i < n; i++)
   {
      for (j = 0; j < i; j++)
      {
         if (Graph[i][j] != 0)
         {               elist.data[elist.n].u = i;
elist.data[elist.n].v = j;
elist.data[elist.n].w = Graph[i][j];
elist.n++;
         }
      }
   }
sort();

   for (i = 0; i < n; i++)
   {
      belongs[i] = i;
   }

   spanlist.n = 0;

   for (i = 0; i < elist.n; i++)
   {
      cno1 = belongs[elist.data[i].u];
cno2 = belongs[elist.data[i].v];
```

```
        if (cno1 != cno2)
        {
            spanlist.data[spanlist.n] = elist.data[i];
spanlist.n = spanlist.n + 1;             applyUnion(belongs,
cno1, cno2);
        }
    }
}
int main()
{     int i, j, cost =
0;

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

    printf("Enter the graph: \n");
for (i = 0; i < n; i++)
    {
        printf("Enter row %d: ", (i + 1));
for (j = 0; j < n; j++)
        {
            scanf("%d", &Graph[i][j]);
        }
    }
    kruskalAlgo();     for (i = 0; i
< spanlist.n; i++)
    {
        printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
cost = cost + spanlist.data[i].w;
    }
    printf("\nSpanning tree cost: %d", cost);
}
```

Output: -



```
Enter the number of nodes: 5
Enter the graph:
Enter row 1: 0 2 0 6 0
Enter row 2: 2 0 3 8 5
Enter row 3: 0 3 0 0 7
Enter row 4: 6 8 0 0 9
Enter row 5: 0 5 7 9 0

1 - 0 : 2
2 - 1 : 3
4 - 1 : 5
3 - 0 : 6
Spanning tree cost: 16
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

<u>Conclusion</u>: -

Learned about Prim’s and Kruskal’s algorithm to find a minimum spanning tree and implemented it.

Experiment – 6

Aim: - Write a program to implement Single Source Shortest Path using Greedy Approach.

Theory: -

Dijkstra's Algorithm:
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 a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

Algorithm: -

Dijkstra's Algorithm:
Algorithm dijkstraShortestPath(n, dist, next, start)

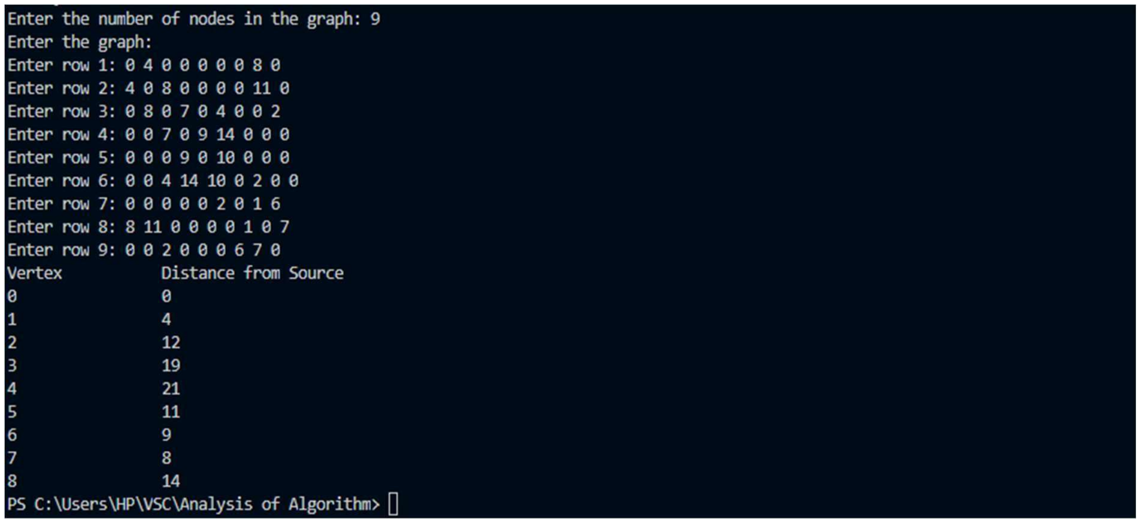
```
   create a status list to hold the current status of the selected node
   for all vertices u in V do
      status[u] := unconsidered
      dist[u] := distance from source using cost matrix
      next[u] := start
   end

   status[start] := considered, dist[start] := 0 and next[start] := φ
   while take unconsidered vertex u as distance is minimum do
      status[u] := considered
      for all vertex v in V do
         if status[v] = unconsidered then
            if dist[v] > dist[u] + cost[u,v] then
               dist[v] := dist[u] + cost[u,v]
               next[v] := u
            end if
         end if
      end
   end

end
```

Code: -

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

int V;

int minDistance(int dist[], int sptSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)         if
(sptSet[v] == 0 && 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
%d\n", i, dist[i]);
}

void dijkstra(int graph[30][30], int src)
{        int  dist[30];         int
sptSet[30];      for (int i = 0;
i < V; i++)
    {
        dist[i] = INT_MAX;
sptSet[i] = 0;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
int u = minDistance(dist, sptSet);        sptSet[u] =
1;

        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);
}

int main()
{     int i, j, graph[30][30];     printf("Enter the number of
nodes in the graph: ");     scanf("%d", &V);
```

```
   printf("Enter the graph: \n");
for(i = 0; i < V; i ++)
   {
       printf("Enter row %d: ", i + 1);
for(j = 0; j< V; j ++)
       {
           scanf("%d", &graph[i][j]);
       }
   }

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

<u>Output</u>: -



```
Enter the number of nodes in the graph: 9
Enter the graph:
Enter row 1: 0 4 0 0 0 0 0 8 0
Enter row 2: 4 0 8 0 0 0 0 11 0
Enter row 3: 0 8 0 7 0 4 0 0 2
Enter row 4: 0 0 7 0 9 14 0 0 0
Enter row 5: 0 0 0 9 0 10 0 0 0
Enter row 6: 0 0 4 14 10 0 2 0 0
Enter row 7: 0 0 0 0 0 2 0 1 6
Enter row 8: 8 11 0 0 0 0 1 0 7
Enter row 9: 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
PS C:\Users\HP\VSC\Analysis of Algorithm> 
```

<u>Conclusion</u>: -

Learned about greedy method for finding single source shortest path and implemented them.

Experiment – 7

Aim: - Write a program to implement the N Queens problem.

Theory: -

N – Queens' problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column, or diagonal. It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens' problem and then generate it to n - queens problem.
Backtracking Algorithm: The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

Algorithm: -

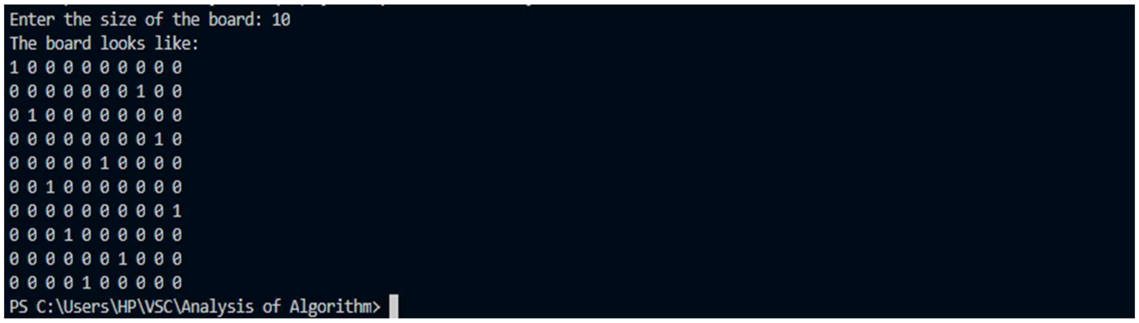
```
Algorithm isValid(board, row, col)

  if there is a queen at the left of current col, then
    return false
  end if
  if there is a queen at the left upper diagonal, then
    return false
  end if
  if there is a queen at the left lower diagonal, then
    return false;
  end if
  return true //otherwise it is valid place

end

Algorithm solveNQueen(board, col)

  if all columns are filled, then
    return true
  end if
  for each row of the board, do
    if isValid(board, i, col), then
      set queen at place (i, col) in the board
      if solveNQueen(board, col+1) = true, then
        return true
      otherwise remove queen from place (i, col) from board.
      end if
    end if
  done
  return false
end
```

Code: -

```
#include<stdio.h>
int n;
int canPlace(int board[n][n], int r, int c)
{    int i,
j;
   for (i = 0; i < c; i++)
   {
      if (board[r][i] == 1)
      {
return 0;
      }
   }
   for (i = r, j = c; i >= 0 && j >= 0; i--, j--)
   {
      if (board[i][j] == 1)
      {
return 0;
      }
   }

   for (i = r, j = c; i < n && j >= 0; i++, j--)
   {
      if (board[i][j] == 1)
      {
return 0;
      }
   }
   return 1;
}

int solveNQueen(int board[n][n], int c)
{
   int i;     if
(c >= n)
   {
return 1;
   }
else
   {
      for (i = 0; i < n; i++)
      {
         if (canPlace(board, i, c) == 1)
         {
board[i][c] = 1;
            if (solveNQueen(board, c + 1) == 1)
            {
return 1;
```

```
  }                 board[i][c] =
0;
          }
        }
    }
    return 0;
}
int main()
{    int i,
j;
    printf("Enter the size of the board: ");
scanf("%d", &n);     int board[n][n];     for
(i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
board[i][j] = 0;
        }
    }
    if (solveNQueen(board, 0) == 0)
    {
        printf("No solution exists");
    }
else
    {
        printf("The board looks like: \n");
for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
            {
                printf("%d ", board[i][j]);
            }
            printf("\n");
        }
    }
```

<u>Output</u>: -



```
Enter the size of the board: 10
The board looks like:
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

<u>Conclusion</u>: -

Learned about N-Queens’ problem and implemented it.

Experiment – 8

Aim: - Write a program to implement the Sum of Subsets.

Theory: -

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains nonnegative values. It is assumed that the input set is unique (no duplicates are presented).
Backtracking Algorithm for Subset Sum
Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.
Assume given set of 4 elements, say w[1] … w[4]. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.

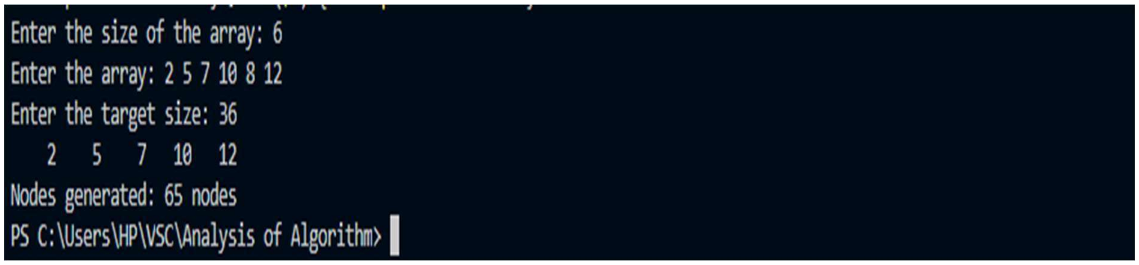
Algorithm: -

```
Algorithm subset_sum(int list[], int sum, int starting_index, int target_sum)

    if( target_sum == sum )          subset_count++;          if(starting_index <
list.length)             subset_sum(list, sum - list[starting_index-1], starting_index,
target_sum);          end if     else
        for( int i = starting_index; i < list.length; i++ )
subset_sum(list, sum + list[i], i + 1, target_sum);          end
    end if

end
```

Code: -

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

static int total_nodes;

void printSubset(int A[], int size)
{
   for (int i = 0; i < size; i++) {
printf("%*d", 5, A[i]);
   }
   printf("\n");
}
void subset_sum(int s[], int t[], int s_size, int t_size, int sum, int ite, int const target_sum)
{
   total_nodes++;      if (target_sum == sum) {           printSubset(t, t_size);
subset_sum(s, t, s_size, t_size - 1, sum - s[ite], ite + 1, target_sum);
return;
   }     else {         for (int i = ite; i < s_size; i++) {             t[t_size] = s[i];
subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
      }
   }
}
void generateSubsets(int s[], int size, int target_sum)
{
   int* tuplet_vector = (int*)malloc(size * sizeof(int));

   subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);

   free(tuplet_vector);
}
int main()
{     int i, n, target_size;      printf("Enter
the size of the array: ");       scanf("%d",
&n);    int weights[n];     printf("Enter the
array: ");     for(i = 0; i < n; i ++)
   {
      scanf("%d", &weights[i]);
   }
   printf("Enter the target size: ");
scanf("%d", &target_size);

   generateSubsets(weights, n, target_size);     printf("Nodes
generated: %d nodes", total_nodes);
   return 0;
}
```

Output: -



```
Enter the size of the array: 6
Enter the array: 2 5 7 10 8 12
Enter the target size: 36
   2   5   7   10   12
Nodes generated: 65 nodes
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

Conclusion: -

Learned about Sum of Subset and implemented it.

Experiment – 9

Aim: - Write a program to implement the Graph Coloring.

Theory: -

Graph coloring is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color. The objective is to minimize the number of colors while coloring a graph. The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NPComplete problem.
Method to Color a Graph:
The steps required to color a graph G with n number of vertices are as follows − Step 1 − Arrange the vertices of the graph in some order.
Step 2 − Choose the first vertex and color it with the first color.
Step 3 − Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

Algorithm: -

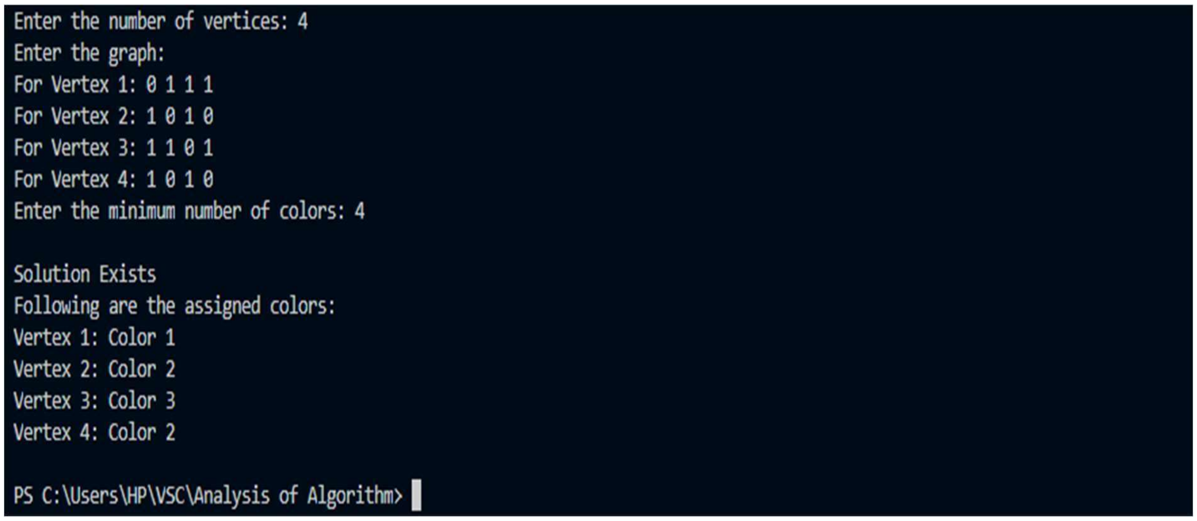
```
Algorithm mColoring

  create the processors P(i0,i1,...in-1) where 0_iv < m, 0 _ v < n    status[i0,..in-
1] = 1

  for j varies from 0 to n-1 do
    begin

      for k varies from 0 to n-1 do
begin
        if aj,k=1 and ij=ikthen
status[i0,..in-1] =0
      end

    end      ok =
ΣStatus

  if ok > 0, then display valid coloring exists
else
    display invalid coloring

end
```

<u>Code</u>: -

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

int V;

void printSolution(int color[])
{     int
i;
    printf( "Solution Exists \nFollowing are the assigned colors: \n");     for
(i = 0; i < V; i++)
    {
        printf("Vertex %d: Color %d\n", (i + 1), color[i]);
    }
printf("\n");
}

int isSafe(int v, int graph[V][V], int color[], int c)
{     for (int i = 0; i < V;
i++)
    {
        if (graph[v][i] && c == color[i])
        {
return 0;
        }
    }
    return 1;
}

int graphColoringUtil(int graph[V][V], int m, int color[], int v)
{
    if (v == V)
    {
return 1;
    }
    for (int c = 1; c <= m; c++)
    {
        if (isSafe(v, graph, color, c))
        {           color[v] = c;           if (graphColoringUtil(graph,
m, color, v + 1) == 1)
            {
return 1;           }
color[v] = 0;
        }
    }
    return 0;
}
```

```
int graphColoring(int graph[V][V], int m)
{     int color[V];     for (int
i = 0; i < V; i++)
   {        color[i]
= 0;
   }

   if (graphColoringUtil(graph, m, color, 0) == 0)
   {
      printf("Solution does not exist");
return 0;
   }
   printSolution(color);
return 1;
}

int main()
{    int m, i,
j;
   printf("Enter  the  number  of  vertices:  ");
scanf("%d", &V);     int graph[V][V];

   printf("Enter the graph: \n");
for(i = 0; i < V; i ++)
   {
      printf("For Vertex %d: ", (i + 1));
for(j = 0; j < V; j ++)
      {
         scanf("%d", &graph[i][j]);
      }
   }
   printf("Enter  the  minimum  number  of  colors:  ");
scanf("%d", &m);     printf("\n");

   graphColoring(graph, m);     return
0;
}
```

Output: -



```
Enter the number of vertices: 4
Enter the graph:
For Vertex 1: 0 1 1 1
For Vertex 2: 1 0 1 0
For Vertex 3: 1 1 0 1
For Vertex 4: 1 0 1 0
Enter the minimum number of colors: 4

Solution Exists
Following are the assigned colors:
Vertex 1: Color 1
Vertex 2: Color 2
Vertex 3: Color 3
Vertex 4: Color 2

PS C:\Users\HP\VSC\Analysis of Algorithm>
```

Conclusion: -
Learned about Graph Coloring and implemented it.

Experiment – 10

Aim: - Write a program to implement Rabin Karp String Matching Algorithm and KMP Algorithm.

Theory: -

Rabin Karp Algorithm:
The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character sub sequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

KMP Algorithm:
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: -

Rabin Karp Algorithm:

```
Algorithm Rabin-Karp-Matcher(T, P, d, q)
n ← length [T]  m ← length [P]  h ← dm-
1 mod q
 p ← 0  t0 ← 0  for i ← 1 to
m    do p ← (dp + P[i]) mod
q     t0 ← (dt0+T [i]) mod q
for s ← 0 to n-m       do if p
= ts
```

```
      then if P [1.....m] = T [s+1.....s + m]        then
"Pattern occurs with shift" s     if s < n-m        then
ts+1 ←  (d (ts-T [s+1]h)+T [s+m+1])mod q
```

KMP Algorithm:

```
Algorithm Prefix-Function(P)
 m ←length [P]                    //'p' pattern to be matched
 Π [1] ← 0  k ← 0  for q ← 2 to m
    do while k > 0 and P [k + 1] ≠ P [q]
do k ← Π [k]     if P [k + 1] = P [q]
then k← k + 1
    Π [q] ← k
 Return Π
```

```
Algorithm KMP-Matcher(T, P)  n
← length [T]
 m ← length [P]
 Π← COMPUTE-PREFIX-FUNCTION (P)  q
← 0     // numbers of characters matched  for i
← 1 to n          // scan S from left to right
    do while q > 0 and P [q + 1] ≠ T [i]     do q ← Π [q]
            // next character does not match
    if P [q + 1] = T [i]
       then q ← q + 1     // next character matches        if q
= m                              // is all of p matched?
          then print "Pattern occurs with shift" i - m
q ← Π [q]
```

<u>Code</u>: -

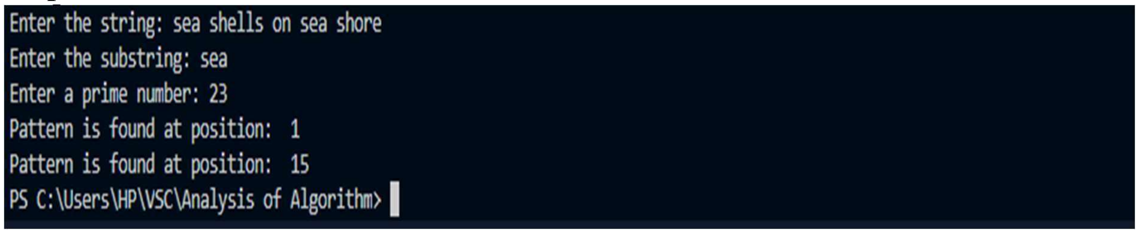
Rabin Karp Algorithm:

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

#define d 10

void rabinKarp(char pattern[], char text[], int q)
{
   int m = strlen(pattern);
   int n = strlen(text);     int i,
j, p = 0, t = 0, h = 1;

   for (i = 0; i < m - 1; i++)
```

```
    {
        h = (h * d) % q;
    }

    // Calculate hash value for pattern and text      for
(i = 0; i < m; i++)
    {
        p = (d * p + pattern[i]) % q;
t = (d * t + text[i]) % q;
    }

    // Find the match      for (i =
0; i <= n - m; i++)
    {         if (p
== t)
        {
            for (j = 0; j < m; j++)
            {                   if (text[i + j] !=
pattern[j])
                {
break;
                }
            }
            if (j == m)
            {
                printf("Pattern is found at position:  %d \n", i + 1);
            }
        }

        if (i < n - m)
        {
            t = (d * (t - text[i] * h) + text[i + m]) % q;
if (t < 0)              {                   t = (t + q);
            }
        }
    }
}

int main()
{
    char text[50], pattern[20];      int q;
printf("Enter the string: ");      scanf("
%[^\n]s", text);       printf("Enter the
substring: ");          scanf(" %[^\n]s",
pattern);          printf("Enter a prime
number: ");     scanf("%d", &q);

    rabinKarp(pattern, text, q);     return
0;
```

```
}
```

Output: -

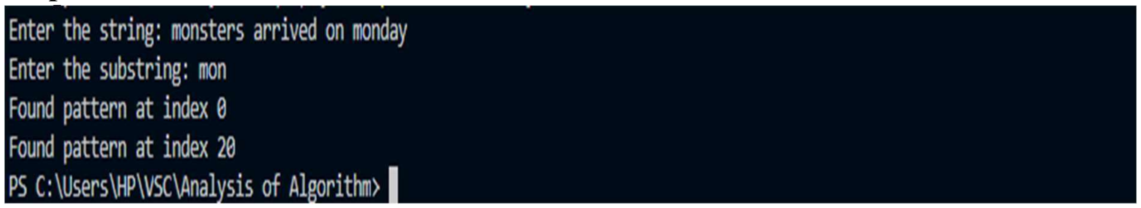


```
Enter the string: sea shells on sea shore
Enter the substring: sea
Enter a prime number: 23
Pattern is found at position:  1
Pattern is found at position:  15
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

KMP Algorithm:

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

void computeLPSArray(char *pat, int M, int *lps)
{     int len = 0;
lps[0] = 0;
int i = 1;
while (i < M)
   {
       if (pat[i] == pat[len])
        {
len++;             lps[i]
= len;            i++;
}        else        {
          if (len != 0)
          {                len =
lps[len - 1];
          }
else             {
              lps[i] = 0;
i++;
          }
        }
    }
}

void KMPSearch(char *pat, char *txt)
{
   int  M  =  strlen(pat);
int   N   =   strlen(txt);
int lps[M];
   computeLPSArray(pat, M, lps);
    int   i   =   0;
int    j    =    0;
while (i < N)
   {
        if (pat[j] == txt[i])
        {
j++;
i++;
        }

        if (j == M)
        {
            printf("Found pattern at index %d\n", i - j);
j = lps[j - 1];
        }
        else if (i < N && pat[j] != txt[i])
        {           if (j != 0)
{               j = lps[j -
1];
            }
else            {
i = i + 1;
            }
        }
    }
}

int main()
{
    char text[50], pattern[20];
printf("Enter the string: ");
    scanf(" %[^\n]s", text);
printf("Enter the substring: ");     scanf("
%[^\n]s", pattern);

    KMPSearch(pattern, text);
return 0;
}
```

Output: -



```
Enter the string: monsters arrived on monday
Enter the substring: mon
Found pattern at index 0
Found pattern at index 20
PS C:\Users\HP\VSC\Analysis of Algorithm>
```

<u>Conclusion</u>: -

Learned about Rabin Karp and KMP string matching and implemented it.