

Department of Computer Engineering

Analysis of Algorithm Laboratory

Academic year: 2021 – 22

Semester - IV

Kartik Jolapara

60004200107

B1

Experiment 1

Aim: Write a program to implement and analyze 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 which are already partially sorted.

Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from 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 selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Algorithm:

```
1) Insertion Sort function
insertionSortR(array A, int n)
if n > 0
    insertionSortR(A, n-1)
    x ← A[n]
    j ← n-1
    while j >= 0 and A[j] > x
        A[j+1] ← A[j]
        j ← j-1
    end while
    A[j+1] ← x    end
if
end function
```

```
2) Selection Sort procedure
selection sort    list : array of
items
    n    : size of list

    for i = 1 to n - 1    min =
i        for j = i+1 to n
```

```

        if list[j] < list[min]
    then
        min = j;
    end if    end for
    if indexMin != i
    then      swap
    list[min] and
    list[i]
        end if
    end for
end procedure

```

Code:

```

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

int InsertionSort(int arr[],int size)
{
    for(int j=1;j<size;j++)
    {
        int k=j-1,key=arr[j];
        while (arr[k]>key && j>=0)
        {
            arr[k+1]=arr[k];
            k--;
        }
        arr[k+1]=key;
    }
}

int SelectionSort(int arr[],int size)
{
    int min,index;
    for (int i = 0; i < size; i++)
    {
        min=arr[i];
        index=i;
        for (int j = i; j < size-1 ; j++)
        {
            if(min>arr[j+1])
            {
                min=arr[j+1];
                index=j+1;
            }
        }
        arr[index]=arr[i];
        arr[i]=min;
    }
}

```

```

    }
}

int main()
{
    int a[20],n,ch;
    printf("ENTER NUMBER OF TERMS IN ARRAY:");
    scanf("%d",&n);
    printf("ENTER %d TERMS:\n",n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\nStart at : %ld\n",time(NULL));
do
    {
        printf("*****ARRAY OPERATIONS*****\n");
        printf("ENTER THE CHOICE OF OPERATION YOU WANT TO PERFORM\n");
        printf("      1.SORT USING INSERTION SORT\n      2.SORT USING SELECTION SORT\n      3. EXIT");
        printf("\n*****\n");
        scanf("%d",&ch);
        switch (ch)
        {
case 1:
            InsertionSort(a,n);
            printf("\nEnd time : %ld\n",time(NULL));
            break;

            case 2:
                SelectionSort(a,n);
                printf("\nEnd time : %ld\n",time(NULL));
            break;

            case 3:
            break;

            default:
                printf("INVALID INPUT");
            break;
        }
    }while (ch!=3);
    return 0;
}

```

Output:

```

ENTER NUMBER OF TERMS IN ARRAY:10
ENTER 10 TERMS:
2 3 4 1 6 5 8 9 7 10

Start at : 1654500524
*****ARRAY OPERATIONS*****
ENTER THE CHOICE OF OPERATION YOU WANT TO PERFORM
1.SORT USING INSERTION SORT
2.SORT USING SELECTION SORT
3. EXIT
*****
1

End time : 1654500527
*****ARRAY OPERATIONS*****
ENTER THE CHOICE OF OPERATION YOU WANT TO PERFORM
1.SORT USING INSERTION SORT
2.SORT USING SELECTION SORT
3. EXIT
*****
2

End time : 1654500530
*****ARRAY OPERATIONS*****
ENTER THE CHOICE OF OPERATION YOU WANT TO PERFORM
1.SORT USING INSERTION SORT
2.SORT USING SELECTION SORT
3. EXIT
*****
3

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

```

Conclusion:

Insertion Sort and Selection Sort algorithms were implemented and their time complexities were analyzed which were found out to be $O(n^2)$ and $O(n^2)$ respectively.

Experiment 2

Aim: Write a program to implement and analyze time complexity of Merge sort and Quick sort.

Theory:

Merge Sort:

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then it merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Quick Sort:

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Algorithm: Merge

Sort:

```
void TopDownMergeSort(A[], B[], n)
```

```
{
```

```
    CopyArray(A, 0, n, B);    // one time copy of A[] to B[]
```

```
    TopDownSplitMerge(B, 0, n, A); // sort data from B[] into A[]
```

```
}
```

```
// Split A[] into 2 runs, sort both runs into B[], merge both runs from B[] to A[]
```

```
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
```

```
void TopDownSplitMerge(B[], iBegin, iEnd, A[])
```

```
{
```

```
    if (iEnd - iBegin <= 1)    // if run size == 1
```

```
return;    // consider it sorted
```

```
    // split the run longer than 1 item into halves    iMiddle =
```

```
(iEnd + iBegin) / 2;    // iMiddle = mid point
```

```
    // recursively sort both runs from array A[] into B[]
```

```

TopDownSplitMerge(A, iBegin, iMiddle, B); // sort the left run

TopDownSplitMerge(A, iMiddle, iEnd, B); // sort the right run
// merge the resulting runs from array B[] into A[]

TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}

// Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1 ].
// Result is      B[ iBegin:iEnd-1 ].

void TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i = iBegin, j = iMiddle;

    // While there are elements in the left or right runs...
    for (k = iBegin; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iMiddle && (j >= iEnd || A[i] <= A[j]))
        {
            B[k] = A[i];      i = i + 1;
        } else {
            B[k] = A[j];      j = j + 1;
        }
    }
}

void CopyArray(A[], iBegin, iEnd, B[])
{
    for (k = iBegin; k < iEnd; k++)
        B[k] = A[k];
}

```

Quick Sort:

```

/* low -> Starting index, high -> Ending index
*/ quickSort(arr[], low, high) {    if (low < high) {

    /* pi is partitioning index, arr[pi] is now at right place
    */    pi = partition(arr, low, high);    quickSort(arr, low,
pi - 1); // Before pi    quickSort(arr, pi + 1, high); // After
pi

}

```

```

}
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position) pivot =
    arr[high];  i = (low - 1) // Index of smaller element and
    indicates the
    // right position of pivot found so far
    for (j = low; j <= high- 1; j++){
        // If current element is smaller than the pivot
        if (arr[j] < pivot){ i++; // increment index
        of smaller element  swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}

```

Code:

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include<stdlib.h>

```

```

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

```

```

void merge(int a[], int low, int mid, int high)
{
    int i=low,j=mid+1,k=low;
    int B[10000];
    while(i<=mid && j<=high)
    {
        if(a[i]<a[j])
        {
            B[k]=a[i];

```



```

i++;
k++;
    }
else
    {
        B[k] =
a[j];      j++;
k++;
    }
    }
    while(j<=high)
    {
        B[k] =
a[j];      k++;
j++;
    }
    while(i<=mid)
    {
        B[k] = a[i];
i++;
        k++;
    }
    for(i=low;i<k;i++)
    {      a[i] =
B[i];
    }
}

```

```

void mergeSort(int a[],int low,int high)
{
    if(low < high)
    {
        int mid = (low + high)/2;
mergeSort(a,low,mid);
mergeSort(a,mid+1,high);
        merge(a,low, mid, high);
    }
}

```

```

void quickSort(int a[],int low,int high)
{
    if(low < high)
    {
        int pivot = a[high],pi,j;
int 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]);
pi = i+1;
quickSort(a,low,pi-1);
quickSort(a,pi+1,high);
}
}

int main()
{
    int a[10000],n,i,s;    printf("Enter
the size of array : ");
    scanf("%d",&n);    printf("\nEnter
array elements : ");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    do{
        printf("\n1)Merge Sort\n2)Quick Sort\n3)Exit\nYour choice = ");
        scanf("%d",&s);
        printf("\nStart time = %ld",time(NULL));
        switch(s)
        {
            case 1:
mergeSort(a,0,n-1);
                break;
            case 2:
                quickSort(a,0,n-1);
            break;
            case 3:
            break;
            default:
                printf("\n Invalid Choice");
        }
        printf("\nSorted Array : ");
        for(i=0;i<n;i++)
        {
            printf(" %d ",a[i]);
        }
        printf("\nEnd Time = %ld",time(NULL));
    }while(s!=3);
    getch();
}

```

Output:

```
Enter the size of array : 10
Enter array elements : 2 1 3 6 4 5 7 8 9 0

1)Merge Sort
2)Quick Sort
3)Exit
Your choice = 1

Start time = 1654501250
Sorted Array : 0 1 2 3 4 5 6 7 8 9
End Time = 1654501250
1)Merge Sort
2)Quick Sort
3)Exit
Your choice = 2

Start time = 1654501253
Sorted Array : 0 1 2 3 4 5 6 7 8 9
End Time = 1654501253
1)Merge Sort
2)Quick Sort
3)Exit
Your choice = 3

Start time = 1654501257
Sorted Array : 0 1 2 3 4 5 6 7 8 9
End Time = 1654501257

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

Conclusion:

Merge sort and Quick sort algorithms were implemented and their time complexities were analyzed which were found out to be $O(n \log(n))$ and $O(n^2)$ respectively.

Experiment 3

Aim: Write a program to implement Single source shortest path using dynamic programming.

Theory:

Any recursive formula can be directly translated into recursive algorithms. However, sometimes the compiler will not implement the recursive algorithm very efficiently. When this is the case, we must do something to help the compiler by rewriting the program to systematically record the answers to subproblems in a table. This is the basic approach behind dynamic programming – all problems must have “optimal substructure.

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.s Algorithm: function bellmanFord(G, S) for each vertex V in G distance[V] <- infinite

```
previous[V] <- NULL distance[S] <- 0
for each vertex V in G
    for each edge (U,V) in G tempDistance <-
distance[U] + edge_weight(U, V) if
tempDistance < distance[V] distance[V] <-
tempDistance previous[V] <- U for each edge
(U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V]
        Error: Negative Cycle Exists

return distance[], previous[]
```

Code:

```
#include <stdio.h>

int main()
{
    int c[5][5]= {
        {0,2,6,7,100},
        {100,0,100,3,6},
        {100,100,0,100,1},
        {100,100,100,0,5},
        {100,100,100,100,0},
    };
};
```

```

    int d[100],pi[100],n,i,j,k,val,m;
printf("Enter the number of nodes : ");
scanf("%d",&n);  for(i=0;i<n;i++){
d[i]=c[0][i];    if (i!=0)    {
pi[i]=1;
    }
}
for(i=0;i<n-1;i++)
{
    for(j=0;j<n;j++)
    {
        for(k=0;k<n;k++)
        {
            val = d[j]
+ c[j][k];
            if(d[k] > val)
            {
d[k] = val;
pi[k] = j+1;
            }
        }
    }
printf("Iteration %d:\n",i+1);
for(m=0;m<n;m++)
{
    printf("\nd%d = %d",m+1,d[m]);
    printf("\tp%d = %d",m+1,pi[m]);
}
printf("\n\n");
}
printf("Final table:\n");
for(i=0;i<n;i++){
printf("\nd%d = %d",i+1,d[i]);
printf("\tp%d = %d",i+1,pi[i]);
}
return 0;
}

```

Output:

```
Enter the number of nodes : 5
```

```
Iteration 1:
```

```
d1 = 0  p1 = 0
```

```
d2 = 2  p2 = 1
```

```
d3 = 6  p3 = 1
```

```
d4 = 5  p4 = 2
```

```
d5 = 7  p5 = 3
```

```
Iteration 2:
```

```
d1 = 0  p1 = 0
```

```
d2 = 2  p2 = 1
```

```
d3 = 6  p3 = 1
```

```
d4 = 5  p4 = 2
```

```
d5 = 7  p5 = 3
```

```
Iteration 3:
```

```
d1 = 0  p1 = 0
```

```
d2 = 2  p2 = 1
```

```
d3 = 6  p3 = 1
```

```
d4 = 5  p4 = 2
```

```
d5 = 7  p5 = 3
```

```
Iteration 4:
```

```
d1 = 0  p1 = 0
```

```
d2 = 2  p2 = 1
```

```
d3 = 6  p3 = 1
```

```
d4 = 5  p4 = 2
```

```
d5 = 7  p5 = 3
```

```
Final table:
```

```
d1 = 0  p1 = 0
```

```
d2 = 2  p2 = 1
```

```
d3 = 6  p3 = 1
```

```
d4 = 5  p4 = 2
```

```
d5 = 7  p5 = 3
```

```
...Program finished with exit code 0
```

```
Press ENTER to exit console.
```

Conclusion: Dynamic approach to find single source shortest path was implemented using the Bellman Ford algorithm with Time Complexity $O(E*V)$.

Experiment 4

Aim: Write a program to implement Longest Common Subsequence.

Theory:

The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from the longest common substring problem: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff utility, and has applications in computational linguistics and bioinformatics.

For the case of two sequences of n and m elements, the running time of the dynamic programming approach is $O(n \times m)$ Algorithm:

```
function LCSLength(X[1..m],
Y[1..n])  C = array(0..m, 0..n)
for i := 0..m    C[i,0] = 0   for j :=
0..n    C[0,j] = 0   for i := 1..m
for j := 1..n
    if X[i] = Y[j]
        C[i,j] := C[i-1,j-1] + 1
    else
        C[i,j] := max(C[i,j-1], C[i-
1,j])  return C[m,n] Code:
```

```
#include <stdio.h>
#include<string.h>      int
b[25][25], c[25 + 1][25 + 1];

void LCS_Length (char x[], char y[])
{
    int m = strlen (x);   int n
= strlen (y);   for (int i = 0; i
< m + 1; i++)
    {
        c[0][i] = 0;
    }
    for (int i = 0; i < m + 1; i++)
```

```

    {      c[i][0]
= 0;
    }
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {      if (x[i] ==
y[j])
            {      c[i + 1][j + 1] =
c[i][j] + 1;      b[i][j] = 0;
            }
            else if (c[i][j + 1] >= c[i + 1][j])
            {      c[i + 1][j + 1] =
c[i][j + 1];      b[i][j] = 1;
            }
            else
            {      c[i + 1][j + 1] = c[i
+ 1][j];      b[i][j] = 2;
            }
        }
    }
    return;
}

```

```

void Print_LCS (char x[], int i, int j)
{
    if (i == -1 || j == -
1)      return;  if
(b[i][j] == 0)
    {
        Print_LCS (x, i - 1, j - 1);
printf ("%c ", x[i]);
    }
    else if (b[i][j] == 1)
    {

```

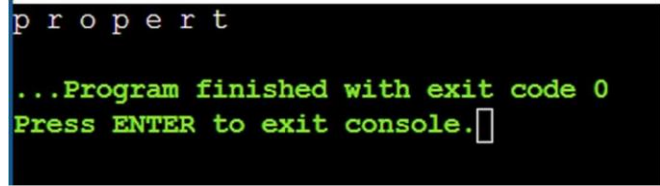


```

        Print_LCS (x, i - 1, j);
    }
    else
    {
        Print_LCS (x, i, j - 1);
    }
}
int main ()
{
    char str1[] =
    "properties";  char str2[]
    = "prosperity";  //
    printf("%s",str1);
    LCS_Length (str1, str2);
    int m = strlen (str1);  int n
    = strlen (str2);  Print_LCS
    (str1, m, n);  return 0;
}

```

Output:



```

p r o p e r t
...Program finished with exit code 0
Press ENTER to exit console.

```

Conclusion: A program to implement Longest Common Subsequence was performed.

Experiment 5

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

Theory:

The minimum spanning tree is a spanning tree whose sum of the edges is minimum.

General properties of minimum spanning tree:

- If we remove any edge from the spanning tree, then it becomes disconnected. Therefore, we cannot remove any edge from the spanning tree.
- If we add an edge to the spanning tree then it creates a loop. Therefore, we cannot add any edge to the spanning tree.
- In a graph, each edge has a distinct weight, then there exists only a single and unique minimum spanning tree. If the edge weight is not distinct, then there can be more than one minimum spanning tree.
- A complete undirected graph can have an $n(n-1)/2$ number of spanning trees.
- Every connected and undirected graph contains at least one spanning tree.
- The disconnected graph does not have any spanning tree.
- In a complete graph, we can remove maximum $(n-1)$ edges to construct a spanning tree.

Prim's:

Prim's algorithm is a greedy algorithm i.e., it picks an optimal way at each point and at last tracks down the briefest way by making a spanning tree. Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which structure a tree that incorporates each vertex has the base amount of weights among every one of the trees that can be formed from the graph

Kruskal:

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph.

Algorithm:

Prim's:

$T = \emptyset$;

$U = \{1\}$; while $(U \neq V)$ let (u, v) be the lowest cost edge such that $u \in U$ and $v \in V$

- U ;

$T = T \cup \{(u, v)\}$

$U = U \cup \{v\}$

Kruskal:

KRUSKAL(G):

$A = \emptyset$

For each vertex $v \in G.V$:

MAKE-SET(v)

For each edge $(u, v) \in G.E$ ordered by increasing order by weight(u, v):

if FIND-SET(u) \neq FIND-SET(v):

A = A \cup $\{(u, v)\}$

UNION(u , v)

return A

Code:

Prim:

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

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

```
#define infinity 9999
```

```
#define MAX 20
```

```
int G[MAX][MAX],spanning[MAX][MAX],n;
```

```
int prims();
```

```
int main()
```

```
{
```

```
    int i,j,total_cost;
```

```
    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]);
```

```
    total_cost=prims();
```

```
    printf("\nspanning tree matrix:\n");
```

```
    for(i=0;i<n;i++)
```

```
    {
```

```
        printf("\n");
```

```
    for(j=0;j<n;j++)
```

```
    printf("%d\t",spanning[i][j]);
```

```
    }
```

```
    printf("\n\nTotal cost of spanning tree=%d",total_cost);
```

```
    return 0;
```

```
}
```

```
int prims()
```

```
{
```

```
    int cost[MAX][MAX];
```

```
    int u,v,min_distance,distance[MAX],from[MAX];
```

```
int visited[MAX],no_of_edges,i,min_cost,j;
```

```
    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];
            spanning[i][j]=0;
        }
    }
    distance[0]=0;
visited[0]=1;
for(i=1;i<n;i++)
{
    distance[i]=cost[0][i];
from[i]=0;
    visited[i]=0;
}
    min_cost=0;
no_of_edges=n-1;
    while(no_of_edges>0)
    {
        min_distance=infinity;
        for(i=1;i<n;i++)
        {
            if(visited[i]==0&&distance[i]<min_distance)
            {
                v=i;
                min_distance=distance[i];
            }
        }
        u=from[v];
        spanning[u][v]=distance[v];
        spanning[v][u]=distance[v];
        no_of_edges--;
        visited[v]=1;
        for(i=1;i<n;i++)
            if(visited[i]==0&&cost[i][v]<distance[i])
            {
                distance[i]=cost[i][v];
                from[i]=v;
            }
        min_cost=min_cost+cost[u][v];
    }
    return(min_cost);
}

```

```

Enter no. of vertices:4

Enter the adjacency matrix:
0 1 2 3
1 0 4 5
2 4 0 6
3 5 6 0

spanning tree matrix:

0      1      2      3
1      0      0      0
2      0      0      0
3      0      0      0

Total cost of spanning tree=6

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

```

Kruskal:

```

#include<stdio.h>
#define MAX 30

typedef struct edge
{
    int u,v,w;
}edge;

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

edgelist elist;

int G[MAX][MAX],n;
edgelist spanlist;

void kruskal();
int find(int belongs[],int vertexno);
void union1(int belongs[],int c1,int c2);
void sort();
void print();

void main()
{
    int i,j,total_cost;
    printf("\nEnter number of vertices:");
    scanf("%d",&n);
    printf("\nEnter the adjacency matrix:\n");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)

```

```
scanf("%d",&G[i][j]);
kruskal(); print();
}
```

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

    for(i=1;i<n;i++)
        for(j=0;j<i;j++)
        {
            if(G[i][j]!=0)
            {
                elist.data[elist.n].u=i;
                elist.data[elist.n].v=j;
                elist.data[elist.n].w=G[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=find(belongs,elist.data[i].u);
        cno2=find(belongs,elist.data[i].v);
        if(cno1!=cno2)
        {
            spanlist.data[spanlist.n]=elist.data[i];
            spanlist.n=spanlist.n+1;
            union1(belongs,cno1,cno2);
        }
    }
}
```

```
int find(int belongs[],int vertexno)
{
    return(belongs[vertexno]);
}
```

```
void union1(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 print()
{
    int i,cost=0;
    for(i=0;i<spanlist.n;i++)
    {
        printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);
        cost=cost+spanlist.data[i].w;
    }
    printf("\n\nCost of the spanning tree=%d",cost);
}

```

Output:

```

Enter number of vertices:4

Enter the adjacency matrix:
0 1 2 3
1 0 4 5
2 4 0 6
3 5 6 0

1      0      1
2      0      2
3      0      3

Cost of the spanning tree=6

...Program finished with exit code 29
Press ENTER to exit console.

```

Conclusion: Minimum Spanning Tree was implemented using Prim's and Kruskal's algorithms with time complexities $O(|V|^2)$ and $O(E \log E)$ respectively.

Experiment 6

Aim: Write a program to implement Single source shortest path using Greedy Approach.

Theory:

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:

```
function dijkstra(G, S)
for each vertex V in G
distance[V] <- infinite
previous[V] <- NULL

    If V != S, add V to Priority Queue Q
distance[S] <- 0

    while Q IS NOT EMPTY      U <- Extract MIN from Q
for each unvisited neighbour V of U
tempDistance <- distance[U] + edge_weight(U, V)
if tempDistance < distance[V]      distance[V] <-
tempDistance      previous[V] <- U    return
distance[], previous[]
```

Code:

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

int min(int a[],int n)
{
int min = 500;
int i;
for(i=0;i<n;i++)
{
    if(a[i]<=min)
    {
        min = i;
    }
}
return min;
```



```
}
```

```
int main()
```

```
{
```

```
int m,dv[100],i,j,mn,d[100],pi[100],mk,node;
```

```
int count = 0; int
```

```
v[]={1,0,0,0,0,0};
```

```
int a[6][6] = {
```

```
{100,2,5,100,100,100},
```

```
{100,100,2,3,100,100},
```

```
{100,100,100,100,2,100},
```

```
{100,100,100,100,100,2},
```

```
{100,100,100,7,100,1},
```

```
{100,100,100,100,100,100}
```

```
};
```

```
printf("Enter the number of nodes :
```

```
"); scanf("%d", &m); d[0] = 0;
```

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

```
{
```

```
    d[i] = 100;
```

```
}
```

```
node = 0;
```

```
while(count<6)
```

```
{
```

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

```
    {
```

```
if(v[i]==1)
```

```
    {
```

```
        continue;
```

```
    }
```

```
        if(d[i]<d[node])
```

```
        {
```

```
node = i;
```

```
        }
```

```
    }
```

```
    for(j=0;j<m;j++)
```

```
    {
```

```
        dv[j] = d[node] + a[node][j];
```

```
if(dv[j]<d[j])
```

```
    {      d[j]
```

```
= dv[j];
```

```
    }
```

```
    }
```

```
    mn = min(dv,m);
```

```
pi[node] = mn;
```

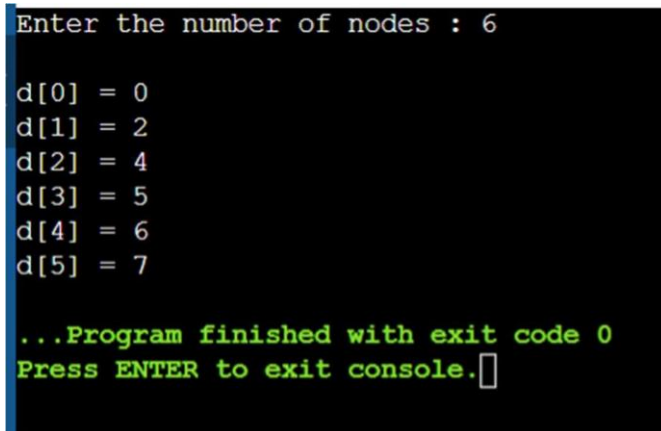
```
v[node] = 1;
```

```
count += 1;
```

```
node += 1;
```

```
}  
for(i=0;i<m;i++)  
{  
    printf("\nd[%d] = %d",i,d[i]);  
}  
getch();  
}
```

Output:



```
Enter the number of nodes : 6  
  
d[0] = 0  
d[1] = 2  
d[2] = 4  
d[3] = 5  
d[4] = 6  
d[5] = 7  
  
...Program finished with exit code 0  
Press ENTER to exit console.█
```

Conclusion: Greedy Approach was used to implement Single Source Shortest Path using Dijkstra's Algorithm with time complexity $O(E \log V)$.

Experiment 7

Aim: Write a program to implement n-Queens problem.

Theory:

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

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:

Place (k, i)

```
{  
    For j ← 1 to k - 1    do if  
(x [j] = i)    or (Abs x [j]) - i) =  
(Abs (j - k))    then return  
false;    return true;  
}
```

N - Queens (k, n)

```
{  
    For i ← 1 to n  
do if Place (k, i) then  
    {    x [k] ← i;  
if (k ==n) then  
write (x [1....n]);  
else  
    N - Queens (k + 1, n);  
    }  
}
```

Code:

```
#include<stdio.h>
#include<math.h>
int board[20],count;

int main()
{
    int n,i,j;
    void queen(int row,int n);

    printf(" - N Queens Problem
    Using Backtracking -");
    printf("\n\nEnter number of
    Queens:"); scanf("%d",&n);
    queen(1,n);
    return 0;
}

void print(int n)
{
    int i,j;
    printf("\n\nSolution
    %d:\n\n",++count);

    for(i=1;i<=n;++i)
        printf("\t%d",i);

    for(i=1;i<=n;++i)
    {
        printf("\n\n%d",i);
        for(j=1;j<=n;++j)
        {
            if(board[i]==j)
                printf("\tQ"); else
                printf("\t-");
        }
    }
}

int place(int row,int column)
{
    int i;
    for(i=1;i<=row-1;++i)
    {
        if(board[i]==column)
            return 0;
        else if(abs(board[i]-
            column)==abs(irow)) return
            0;
```

```
}
```

```
return 1;
```

```
}
```

```
void queen(int row,int n)
```

```
{
```

```
int column;
```

```
for(column=1;column<=n;++column
```

```
n)
```

```
{
```

```
if(place(row,column))
```

```
{
```

```
board[row]=column;
```

```
if(row==n)
```

```
print(n);
```

```
else
```

```
queen(row+1,n);
```

```
}
```

```
}
```

```
}
```

Output:

- N Queens Problem Using Backtracking -

Enter number of Queens:4

Solution 1:

	1	2	3	4
1	-	Q	-	-
2	-	-	-	Q
3	Q	-	-	-
4	-	-	Q	-

Solution 2:

	1	2	3	4
1	-	-	Q	-
2	Q	-	-	-
3	-	-	-	Q
4	-	Q	-	-

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

Conclusion: N-Queens algorithm was implemented with time complexity $O(n^2)$.

Experiment 8

Aim: Write a program to implement Sum of Subsets.

Theory:

Sum of subsets problem is analogous to the knapsack problem. The Knapsack Problem tries to fill the knapsack using a given set of items to maximize the profit. Items are selected in such a way that the total weight in the knapsack does not exceed the capacity of the knapsack. The inequality condition in the knapsack problem is replaced by equality in the sum of subsets problem. Given the set of n positive integers, $W = \{w_1, w_2, \dots, w_n\}$, and given a positive integer M , the sum of the subset problem can be formulated as follows (where w_i and M correspond to item weights and knapsack capacity in the knapsack problem):

$$\sum_{i=1}^n w_i x_i = M \text{ where } x_i \in \{0, 1\}$$

Numbers are sorted in ascending order, such that $w_1 < w_2 < w_3 < \dots < w_n$. The solution is often represented using the solution vector X . If the i th item is included, set x_i to 1 else set it to 0. In each iteration, one item is tested. If the inclusion of an item does not violate the constraint of the problem, add it. Otherwise, backtrack, remove the previously added item, and continue the same procedure for all remaining items. The solution is easily described by the state space tree. Each left edge denotes the inclusion of w_i and the right edge denotes the exclusion of w_i . Any path from the root to the leaf forms a subset.

Algorithm:

```
Algorithm SUB_SET_PROBLEM(i, sum, W,
remSum)      W: Number for which subset is
to be computed i: Item index sum : Sum of
integers selected so far remSum : Size of
remaining problem i.e. (W – sum)

// Output : Solution tuple X if
FEASIBLE_SUB_SET(i) == 1 then
if (sum == W) then    print
X[1...i]  end else
X[i + 1] ← 1
SUB_SET_PROBLEM(i + 1, sum + w[i] + 1, W, remSum – w[i] + 1 )
X[i + 1] ← 0    // Exclude the ith item
SUB_SET_PROBLEM(i + 1, sum, W, remSum – w[i] + 1 )
end

function FEASIBLE_SUB_SET(i) if (sum + remSum ≥ W) AND (sum ==
W) or (sum + w[i] + 1 ≤ W) then
```

```
    return 0
```

```
end return
```

```
1
```

Code:

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

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

```
#define ARRAYSIZE(a)(sizeof(a))/(sizeof(a[[]]))
```

```
static int total_nodes;
```

```
void printSubset(int A[], int size)
```

```
{
```

```
    for(int i = 0; i < size; i++)
```

```
    {
```

```
        printf("%d", 5, A[i]);
```

```
    }
```

```
    printf("\n");
```

```
}
```

```
int comparator(const void *pLhs, const void *pRhs)
```

```
{
```

```
    int *lhs = (int *)pLhs;
```

```
    int *rhs = (int *)pRhs;
```

```
    return *lhs > *rhs;
```

```
}
```

```
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);
```

```
        if( ite + 1 < s_size && sum - s[ite] + s[ite+1] <= target_sum )
```

```
        {
```

```
            subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum);
```

```
        }
```

```
        return;
```

```
    }
```

```
    else
```

```
    {
```



```

        if( ite < s_size && sum + s[ite] <= target_sum )
        {
            for( int i = ite; i < s_size; i++ )
            {
                t[t_size] = s[i];

                if( sum + s[i] <= target_sum )
                {
                    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 *tuple_vector = (int *)malloc(size * sizeof(int));
    int total = 0;
    qsort(s, size, sizeof(int), &comparator);
    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }
    if( s[0] <= target_sum && total >= target_sum )
    {
        subset_sum(s, tuple_vector, size, 0, 0, 0, target_sum);
    }
    free(tuple_vector);
}

```

```

int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53; int size =
    ARRAYSIZE(weights);
    generateSubsets(weights, size, target);
    printf("Nodes generated %d\n",
    total_nodes); return 0;
}

```

Output:

```
      8      9      14      22
      8      14      15      16
      15      16      22
Nodes generated 68

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

Conclusion: Sum of Subsets was implemented using backtracking with time complexity $O(2^n)$.

Experiment 9

Aim: Write a program to implement Graph Coloring.

Theory:

Graph coloring problem is to assign colors to certain elements of a graph subject to certain constraints.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like Edge Coloring (No vertex is incident to two edges of same color) and Face Coloring (Geographical Map Coloring) can be transformed into vertex coloring.

Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number.

Algorithm:

```
isValid(vertex, colorList, col) Begin  for all vertices v of the
graph, do    if there is an edge between v and i, and col =
colorList[i], then        return false
    done  return true End graphColoring(colors, colorList,
vertex) Begin  if all vertices are checked, then    return
true  for all colors col from available colors, do    if
isValid(vertex, color, col), then        add col to the colorList
for vertex    if graphColoring(colors, colorList, vertex+1)
= true, then        return true        remove color for vertex
done  return false
End
```

Code:

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

#define V 4

void printSolution(int color[]);

bool isSafe(bool graph[V][V], int color[])
{ for (int i = 0; i < V;
i++)
    for (int j = i + 1; j < V; j++)
        if (graph[i][j] && color[j] == color[i])
            return false;
return true;
}
```

```

bool graphColoring(bool graph[V][V], int m, int i,
                  int color[V])
{
    if (i == V) {
        if (isSafe(graph, color)) {
            printSolution(color);
            return true;
        }
        return false;
    }

    for (int j = 1; j <= m; j++) {        color[i] = j;
        if (graphColoring(graph, m, i + 1,
color))
            return true;

        color[i] = 0;
    }

    return false;
}

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

int main()
{
    bool graph[V][V] = {
        { 0, 1, 0, 1 },
        { 1, 0, 1, 0 },
        { 0, 1, 0, 1 },
        { 1, 0, 1, 0 },
    };
    int m = 3;
    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    if (!graphColoring(graph, m, 0, color))
        printf("Solution does not exist");

    return 0;
}

```

Output:

```
Solution Exists: Following are the assigned colors
1 2 1 2

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

Conclusion: A program for graph coloring was implemented with time complexity $O((m^V))$.

Experiment 10

Aim: Write a program to implement Rabin Karp String matching algorithm and KMP algorithm.

Theory:

Rabin-Karp:

Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. To find a single match of a single pattern, the expected time of the algorithm is linear in the combined length of the pattern and text, although its worst-case time complexity is the product of the two lengths. To find multiple matches, the expected time is linear in the input lengths, plus the combined length of all the matches, which could be greater than linear. In contrast, the Aho–Corasick algorithm can find all matches of multiple patterns in worst-case time and space linear in the input length and the number of matches (instead of the total length of the matches).

A practical application of the algorithm is detecting plagiarism. Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single-string searching algorithms are impractical.

KMP:

For a given string 'S', string matching algorithm determines whether a pattern 'p' occurs in the given string 'S'. The disadvantage of a naive string matching algorithm is that this algorithm runs very slow. That means the time complexity of this algorithm is very high. To solve this problem, the KMP string matching algorithm comes into existence. It improves the time complexity of a normal string matching algorithm to $O(n)$, linear time. The working idea behind this algorithm is that whenever a mismatch is detected after some matches we know some of the characters in the given string of the next shift. This information is useful in avoiding the matching characters.

Algorithm: Rabin-

Karp:

```
function RabinKarp(string s[1..n], string
pattern[1..m])  hpattern := hash(pattern[1..m]);
for i from 1 to n-m+1    hs := hash(s[i..i+m-1])
if hs = hpattern        if s[i..i+m-1] = pattern[1..m]
return i
return not found
```

KMP:

COMPUTE- PREFIX- FUNCTION (P)

1. $m \leftarrow \text{length}[P]$
2. $\pi[1] \leftarrow 0$
3. $k \leftarrow 0$

4. for $q \leftarrow 2$ to m
5. do while $k > 0$ and $P[k + 1] \neq P[q]$
6. do $k \leftarrow \Pi[k]$
7. If $P[k + 1] = P[q]$
8. then $k \leftarrow k + 1$
9. $\Pi[q] \leftarrow k$
10. Return Π

KMP-MATCHER (T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$
5. for $i \leftarrow 1$ to n
6. do while $q > 0$ and $P[q + 1] \neq T[i]$
7. do $q \leftarrow \Pi[q]$
8. If $P[q + 1] = T[i]$
9. then $q \leftarrow q + 1$
10. If $q = m$
11. then print "Pattern occurs with shift" $i - m$
12. $q \leftarrow \Pi[q]$

Code:

Rabin-Karp:

```
#include<stdio.h>
#include<string.h>
#define d 256
```

```
void search(char pat[], char txt[], int q, int M, int N)
{
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;

    for (i = 0; i < M-1; i++)
        h = (h*d)%q;
```

```

    for (i = 0; i < M; i++)
    {
        p = (d*p + pat[i])%q;
        t = (d*t + txt[i])%q;
    }

    for (i = 0; i <= N - M; i++)
    {
        if ( p == t )
        {
            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                    break;
            }
            if (j == M)
                printf("Pattern found at index %d \n", i);
        }
        if ( i < N-M )
        {
            t = (d*(t - txt[i]*h) + txt[i+M])%q;
            if (t < 0)
                t = (t +
q);
        }
    }
}

```

```

int main()
{
    char txt[100];
    char pat[100];
    int q = 101,
i,n,m;
    printf("Size of Text : ");
    scanf("%d",&n);
    printf("Size of Pattern : ");
    scanf("%d",&m);
    printf("Enter the Text : ");
    for(i=0;i<n;i++)
    {
        scanf(" %c",&txt[i]);
    }
    printf("Enter the Pattern : ");
    for(i=0;i<m;i++)
    {
        scanf(" %c",&pat[i]);
    }
}

```



```

        search(pat, txt, q, m, n);
        return 0;
}

```

```

Size of Text : 6
Size of Pattern : 2
Enter the Text : DEEVYA
Enter the Pattern : EV
Pattern found at index 2

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

```

KMP:

```

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

```

```

void KMP(const char* text, const char* pattern, int m, int n)

```

```

{
    if (*pattern == '\0' || n == 0) {
        printf("The pattern occurs with shift 0");
    }

```

```

    if (*text == '\0' || n > m) {
        printf("Pattern not found");
    }

```

```

    int next[n + 1];

```

```

    for (int i = 0; i < n + 1; i++) {
        next[i] = 0;
    }

```

```

    for (int i = 1; i < n; i++)
    {
        int j = next[i + 1];

```

```

        while (j > 0 && pattern[j] != pattern[i]) {
            j = next[j];
        }

```

```

        if (j > 0 || pattern[j] == pattern[i]) {
            next[i + 1] = j + 1;
        }
    }
}

```

```

    }
}

for (int i = 0, j = 0; i < m; i++)
{
    if (*(text + i) == *(pattern + j))
    {
        if (++j == n) {
            printf("The pattern occurs with shift %d\n", i - j + 1);
        }
    }
    else if (j > 0)
    {
        j = next[j];
        i--;
    }
}
}

```

```

int main(void)
{
    char* text = "ABCABAABACABAC";
    char* pattern = "ABA";

    int n = strlen(text);
    int m = strlen(pattern);

    KMP(text, pattern, n, m);

    return 0;
}

```

Output:

KMP:

Input – Text : ABCABAABACABAC & Pattern : ABA

```

The pattern occurs with shift 3
The pattern occurs with shift 6
The pattern occurs with shift 10

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

```

Conclusion: A program was written to implement String matching using Rabin Karp and KMP algorithms with time complexities $O(nm)$ and $O(n + m)$ respectively.