Analysis of Algorithms

List of Lab Experiment

Sr. No.	Name of the experiment
1.	**Implementation of Insertion Sort**
2.	**Implementation of Binary Search**
3.	**Implementation of Merge Sort**
4.	**Implementation of Quick Sort**
5.	**Implementation of Prim's Algorithm**
6.	**Implementation of Kruskal's Algorithm**
7.	**Implementation of 0/1 Knapsack**
8.	**Implementation of Longest common subsequence algorithm**
9.	**Implementation of Graph Coloring using Backtracking.**
10.	**Implementation of Rabin Karp**
11.	**Implementation of Sequential search**

EXPERIMENT NO.1

Aim: Write a Program to implement Insertion sort.

Theory:Insertion Sort

An insertion sort is one that sorts a set of records by inserting records into an existing sorted file. The simplest way to insert next element into the sorted part is to shift it down, until it occupies correct position. Initially the element stays right after the sorted part. At each step algorithm compares the element with one before it and, if they stay in reversed order, swap them.

Algorithm:

Function *insertionSort*(arrayA)

For I **from**1**to** length[A]-1 **do**

y=A[i
] j =
i-1
whilej>=0 **and** A[j] >value **do**

A[j+1]=A[j]

j = j-1
done
A[j+1]=y

Done

Analysis of Insertion Sort:

The best case input is an array that is already sorted. In this case insertion sort has aline a running time (i.e., Θ(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)). The average case is also quadratic, which makes insertions impractical for sorting large arrays.

Conclusion: Insertion sort is very similar in that after the *k*th iteration, the first *k* elements in the array are in sorted order.Insertion sort's advantage is that it only scans as many elements as it needs in order to place the *k*+ 1st element, while selection sort must scan all remaining elements to find the *k*+ 1st element.Insertion sort is one of the fastest algorithms for sorting very small arrays. Insertion sort typically makes fewer comparisons than selection sort.

.***Code: Insertion Sort***

AIM:-TO WRITE A PROGRAM FOR INSERTION_SORT.

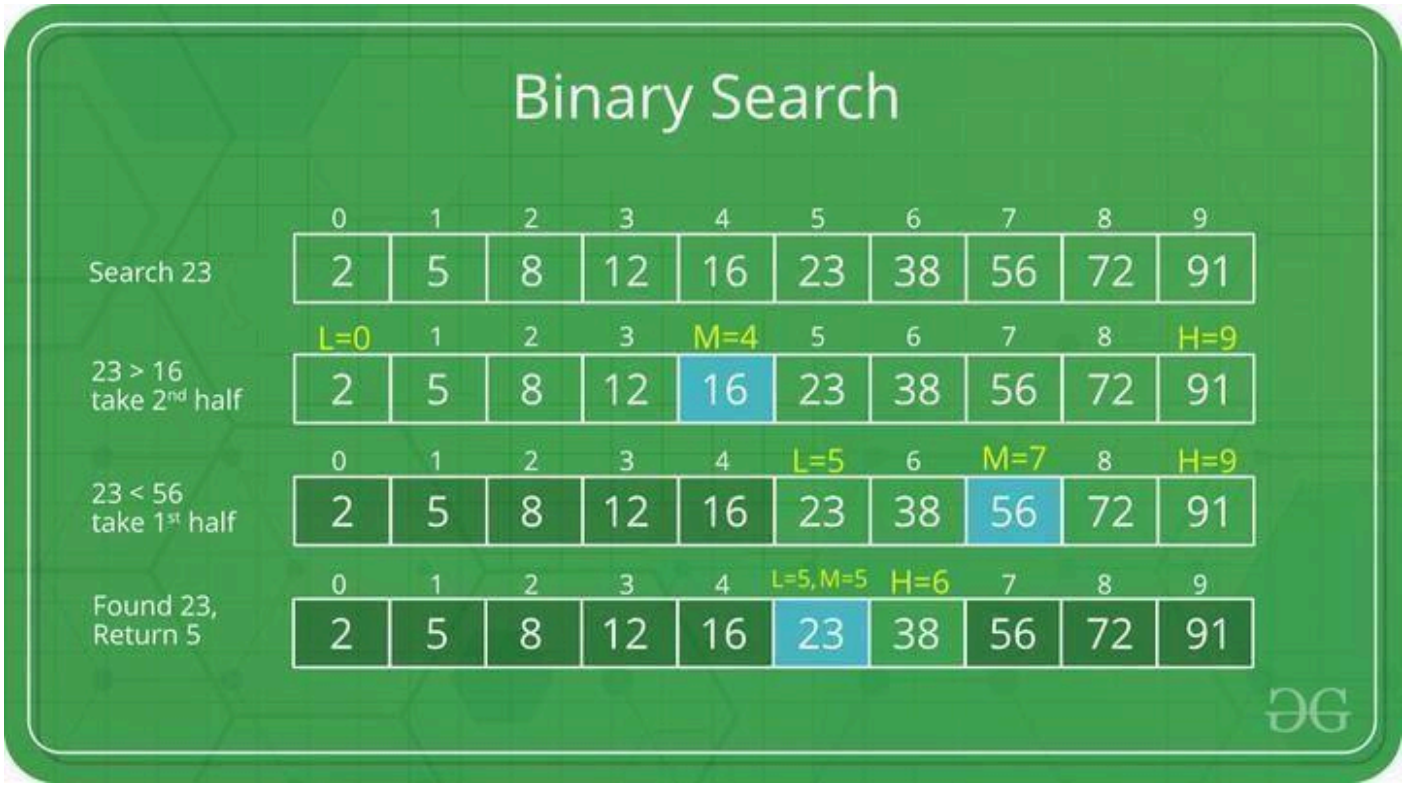
```
#include<stdio.h
>
#include<conio.h
>  #define  MAX
100
void insertion_sort(int arr[],int n);
void main()
{
int n,i,a[MAX];
clrscr();
printf("\n ENTER THE NUMBER OF ELEMENTS TO BE SORT ");
scanf("%d",&n);
printf("\n ENTER %d ELEMENTS IN ARRAY TO SORT ",n);
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("\n ELEMENTS BEFORE SORTING\n ");
for(i=0;i<n;i++)
{
prrintf("%d\t",a[i]);
}
insertion_sort(a,n);
printf("\n ELEMENTS AFTER SORTING\n ");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
getch();
}
void insertion_sort(int arr[],int n)
{
int i,key,j;
for(i=1;i<n;i++)
{
key=arr[i];
j=i-1;                          //Move elements of arr[0..i-1],that are greater than key,to one position ahead of
their current position
while(j>=0 && arr[j]>key)
{
arr[j+1]=arr[j]
; j=j-1;
}
arr[j+1]=key;
}
}
```

EXPERIMENT NO.2

Aim: Write a Program to implement Binary Search.

Theory: Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).



Algorithm:

Procedure binary_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

```
    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
    if A[midPoint] < x
        set lowerBound = midPoint + 1
    if A[midPoint] > x
        set upperBound = midPoint - 1
    if A[midPoint] = x
        EXIT: x found at location midPoint
end while
        end procedure
```

Analysis

Since the binary search algorithm performs searching iteratively, calculating the time complexity is not as easy as the linear search algorithm.The input array is searched iteratively by dividing into multiple sub-arrays after every unsuccessful iteration. Therefore, the recurrence relation formed would be of a dividing function.

Conclusion: Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in a sorted form.

Code: Binary Search

AIM: PROGRAM TO IMPLEMENT BINARY SEARCH

```
#include<stdio.h
> void main()
{
        int arr[100],i, b,e,mid,n,t;
        clrscr();
        printf("Enter Number of elements in array in sorted order.");
        scanf("%d",&n);
        printf("\nEnter %d Elements in Array: ",n);
        for(i=0;i<n;i++)
        {
                scanf("%d",&arr[i]);
        }
        printf("\nEnter Element to be search:");
        scanf("%d",&t);
        b=0;
        e=n-1;
        while(b<=e)
        {
                mid=(b+e)/2;
```

```
                if(t<arr[mid])
                        e=mid-1;
                else if(t>arr[mid])
                        b=mid+1;
                                                else        b=e+1;

                }


        if(t==arr[mid])
                printf("\nElement %d found at %d Location",t,mid);
        else getch();                                           printf("\n Element Not Found...");


}
```

*********OUTPUT**********

EXPERIMENTNO.3

Aim: Program to implement Merge sort analysis

Theory: Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most used and approached algorithms.Merge sort first divides the array into equal halves and then combines them in a sorted manner.

```
procedure 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 procedure

Analysis-Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

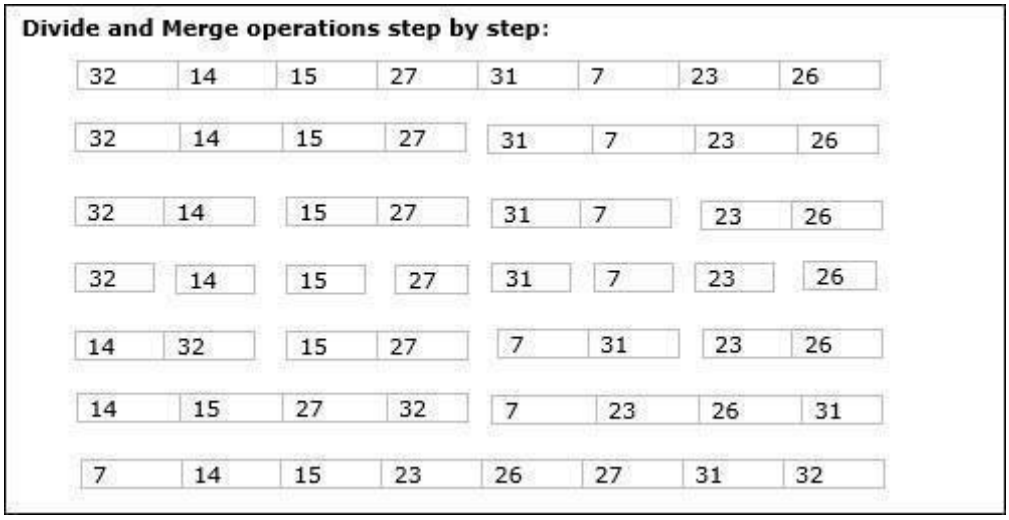
T(n)={c2xT(n2)+dxn if n≤1 otherwise where c and d are constantsTherefore, using this recurrence relation,

T(n)=2iT(n/2i)+i·d·n.

As,i=logn,T(n)=2lognT(n/2logn)+logn·d·n

=c·n+d·n·logn Therefore,T(n)=O(nlogn).

Example
In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.



Conclusion- Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most used and approached algorithms.

Code-Merge Sort

Aim: Program to implement Merge sort.

```
#include<stdio.h
>
#include<conio.h
>  #define  MAX
100
void mergeSort(int a[], int lb,int ub);
void merge(int a[], int lb,int mid,int ub);
void main()
```

{

```
        int n,i,a[MAX],lb=0,ub;
        clrscr();
        printf("Enter Number of Element to be Sort:
        "); scanf("%d",&n);
        printf("\n Enter %d Elements in Array to Sort: ",n);
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("\n Elements Before Sorting:\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",a[i]);
        }
        ub=n-1;
        mergeSort(a,lb,ub);
        printf("\n Elements after Sorting:\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",a[i]);
        }
        getch();
}
void mergeSort(int a[], int lb,int ub)
{
        int mid; if(lb<ub)
        {
                                                        mid=(lb+ub)/2;mergeSort(a,
        }
}
```

```
l
b
,
m
i
d
)
;
m
e
r
g
e
S
o
r
t
(
a
,
m
i
d
+
1
,
u
b
)
;
m
e
r
g
e
(
a
,
l
b
,
m
i
d
,
u
b
)
;
```

```
void merge(int a[],int lb,int mid,int ub)
{
        int
        i=lb,j=mid+1,k=lb,b[MAX];
        while(i<=mid && j<=ub)
        {
                if(a[i]<=a[j])
                {
                                        }
                                        else
                                        {


                                        } k++;


        }

        if(i>mid)
```

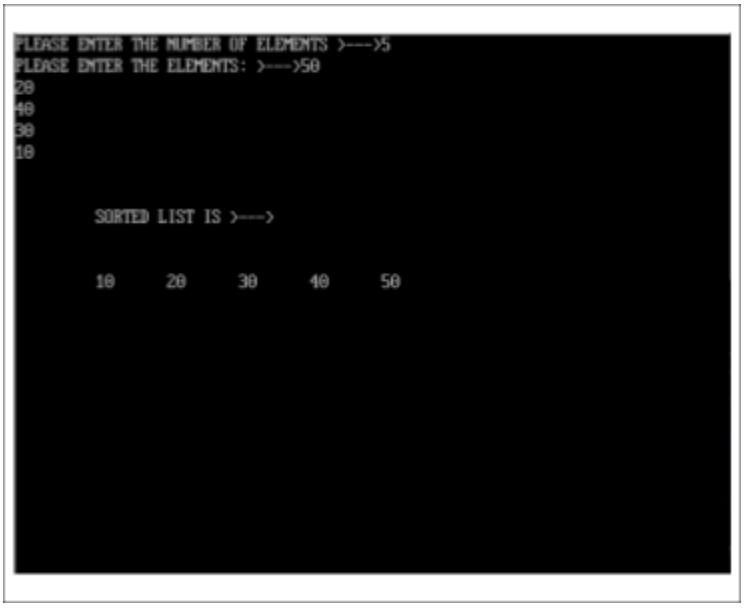
```
b
[
k
]
=
a
[
i
]
;
i
+
+
;
```

```
b
[
k
]
```

=
a
[
j
]
;
j
+
+
;

```
        {
                while(j<=ub)
                {
                                                        b
                                                        [
                                                        k
                }                                       ]
        }                                               =
        else                                            a
        {                                               [
                                                        j
                                                        ]
                                                        ;
                                                        k
                                                        +
                                                        +
                                                        ;
                                                        j
                                                        +
                                                        +
                                                        ;
                while(i<=mid)
                {
                        b[k]=a[i];
                        k++;i++;
                }
        }
        //copying sorted array elements from b[] into a[]
        for(i=lb;i<=ub;i++)
                a[i]=b[i];

}
```



OUTPUT:

EXPERIMENTNO.4

Aim:Program to implement Quick sort analysis.

Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Algorithm: QuickSort

Quick Sort Pivot Pseudocode

```
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

Quick Sort Pseudocode

```
procedure 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 procedure
```

Analysis-

The worst case complexity of Quick-Sort algorithm is $O(n^2)$. However, using this technique, in average cases generally we get the output in O (n log n) time.

Conclusion: Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

Code-Quick Sort

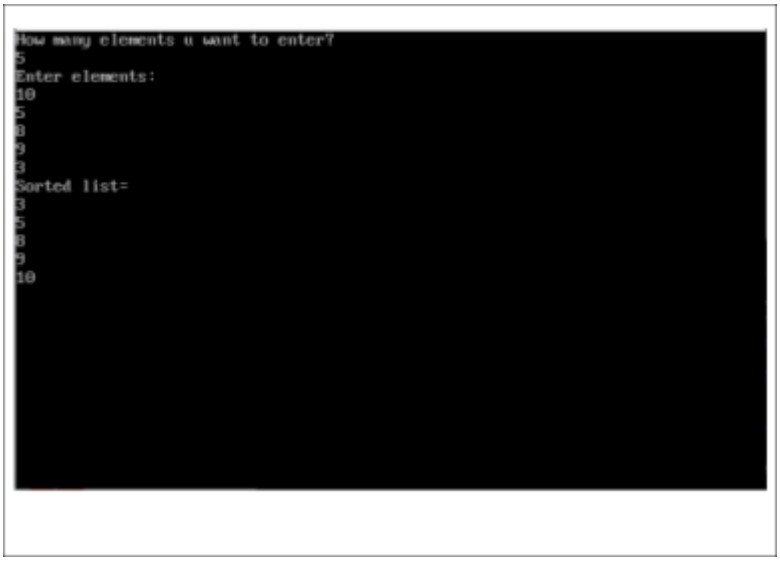
Aim: Program to implement Quick sort.

```
#include<stdio.h
>
#include<conio.h
>  #define  MAX
100
int partition(int a[],int lb,int ub);
void quickSort(int a[],int lb,int ub);
void main()
{
        int n,i,a[MAX],lb,ub;
        clrscr();
        printf("Enter Number of Element to be Sort:
        "); scanf("%d",&n);
        printf("\n Enter %d Elements in Array to Sort: ",n);
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("\n Array Before Sorting:\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        lb=0;
        ub=n-1
        ;
        quickSort(a,lb,ub);
        printf("\n Array After Sorting:\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        getch();
}
void quickSort(int a[],int lb,int ub)
{      int loc;
        if(lb<ub
        )
        {
                loc=partition(a,lb,ub)
                ;
                quickSort(a,lb,loc-1);
                quickSort(a,loc+1,ub
                );
        }
}
int partition(int a[],int lb, int ub)
{
        int
        pivot,start,end,temp;
        pivot=a[lb];
        start=lb;
```

```
end=ub;
while(start<end
)
{
        while(a[start]<=pivot)
                start++;
        while(a[end]>pivot)
                end--
        ; if(start<end)
```

```
                {
                        temp=a[start];
                        a[start]=a[end];
                        a[end]=temp;
                 }
        }
        temp=a[lb];
        a[lb]=a[end]
        ;
        a[end]=temp
        ; return end;
}
```

**********OUTPUT**********



EXPERIMENTNO.5

Aim:To implement Minimum Spanning Tree algorithm Prim's algorithm.
Theory:Prim'sAlgorithm

Prim's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a sub graph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).The algorithm, similar to any shortest path algorithm, begins from a vertex that is set as a root and walks through all the vertices in the graph by determining the least cost adjacent edges.To execute the prim's algorithm, the inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S. A minimum spanning tree of graph G is obtained as an output.

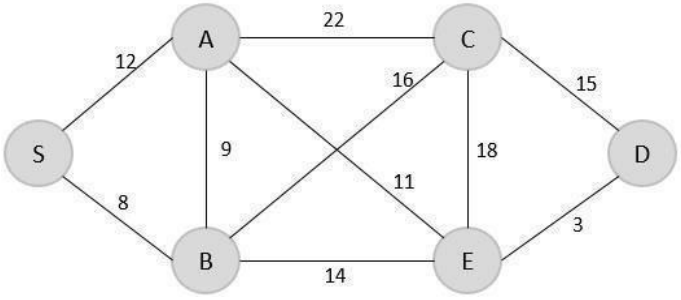
Algorithm

- Declare an array *visited*[] to store the visited vertices and firstly, add the arbitrary root, say S, to the visited array.
- Check whether the adjacent vertices of the last visited vertex are present in the *visited*[] array or not.
- If the vertices are not in the *visited*[] array, compare the cost of edges and add the least cost edge to the output spanning tree.
- The adjacent unvisited vertex with the least cost edge is added into the *visited*[] arrayand the least cost edge is added to the minimum spanning tree output.
- Steps 2 and 4 are repeated for all the unvisited vertices in the graph to obtain the full minimum spanning tree output for the given graph.
- Calculate the cost of the minimum spanning tree

obtained. Analysis

- The algorithm spends most of its time in finding the smallest edge. So, time of the algorithm basically depends on how do we search this edge.
- Straightforward method. Just find the smallest edge by searching the adjacency list of the vertices in V. In this case, each iteration costs O(m) time, yielding a total running time of O(mn).

Example-
Find the minimum spanning tree using prim's method (greedy approach) for the graph given below with S as the arbitrary root.



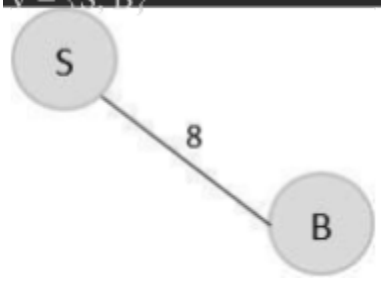
Solution

Step 1-Create a visited array to store all the visited vertices into it.

```
V = { }
```

The arbitrary root is mentioned to be S, so among all the edges that are connected to S we need to find the least cost edge.

```
S → B = 8
V = {S, B}
```

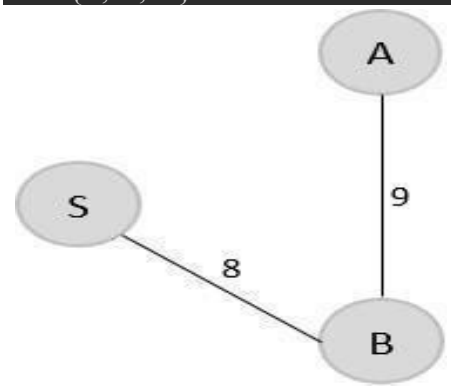


Step -2 Since B is the last visited, check for the least cost edge that is connected to the vertex B.

```
B → A = 9
B → C =
16 B → E
= 14
```

Hence, B → A is the edge added to the spanning tree.

```
V = {S, B, A}
```

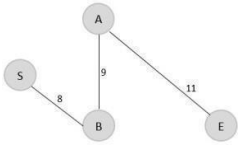


Step- 3 Since A is the last visited, check for the least cost edge that is connected to the vertex A.

```
A → C = 22
A → B = 9
A → E =
11
```

But A → B is already in the spanning tree, check for the next least cost edge. Hence, A → E is added to the spanning tree.

```
V = {S, B, A, E}
```

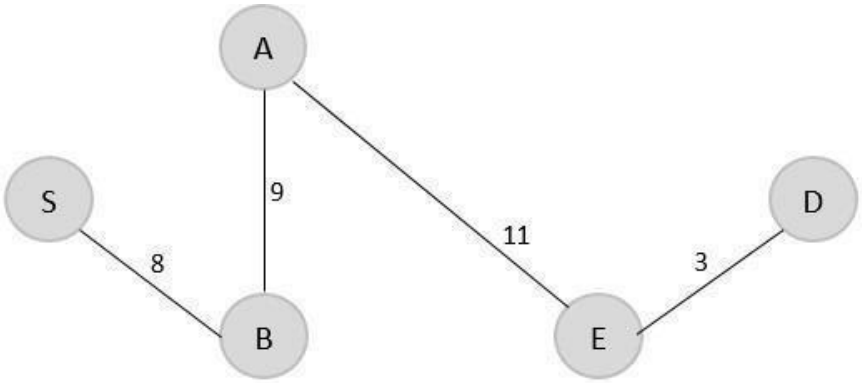


Step 4-Since E is the last visited, check for the least cost edge that is connected to the vertex E.

```
E → C =
18 E → D
= 3
```

Therefore, E → D is added to the spanning tree.

```
V = {S, B, A, E, D}
```

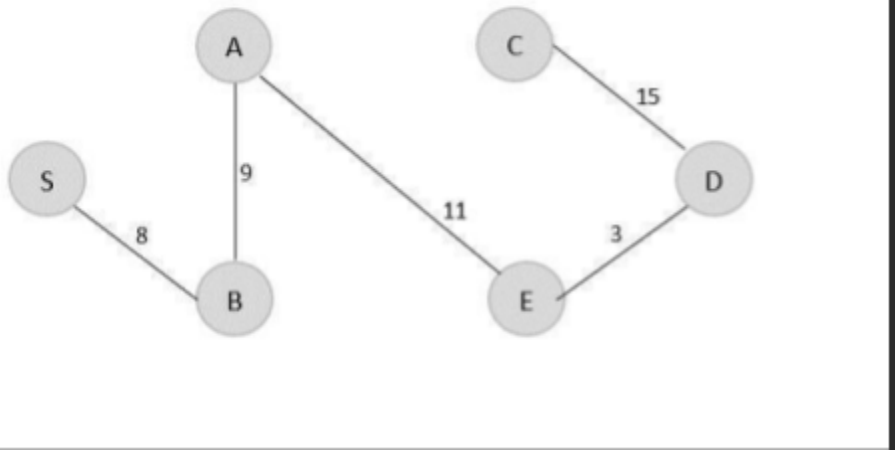


Step 5-Since D is the last visited, check for the least cost edge that is connected to the vertex D.

```
D → C = 15
E → D = 3
```

Therefore, D → C is added to the spanning tree.

```
V = {S, B, A, E, D, C}
```



The minimum spanning tree is obtained with the minimum cost = 46

Conclusion-Program implements Prim's minimum spanning tree problem that takes the cost adjacency matrix as the input and prints the spanning tree as the output along with the minimum cost.

Code-Prim's Algorithm for Minimum Spanning Tree.
Aim: Programto implement Prim's Algorithm for Minimum Spanning Tree.

```
#include<stdio.h>

#include<conio.h>
#define SIZE 20
#define INFINITY 32767
/*This function finds MST by prim's
algorithm*/ void Prim(int G[][SIZE],int nodes)
{
        int tree[SIZE],i,j,k;
        int min_dist,v1,v2,total=0;
//initialize the selected vertices
        list for(i=0;i<nodes;i++)
        tree[i] =0;
        printf("\n\n The Minimal Spanning Tree Is :\n");
        tree[0]=1;
        for(k=1;k<nodes;k++)
        {
                min_dist=INFINITY;
               //initially assign minimum distance as infinity
                for(i=0;i<=nodes;i++)
                {
                        for(j=0;j<=nodes;j++)
                        {
                                if(G[i][j]&&((tree[i]&&!tree[j])||(!tree[i]&&tree[j])))
                                {
                                        if(G[i][j]<min_dist)
                                        {
                                                min_dist=G[i][j];
                                                v1=i;
                                                v2=j;
```

```
                                        }
                                }
                        }
                }
                printf("\nEdge(%d%d)and weight=%d",v1,v2,min_dist);
                tree[v1]=tree[v2]=1;
                total=total+min_dist;
        }
        printf("\n\n\t Total Path Length Is=%d",total);
}
void main()
{
        int
        G[SIZE][SIZE],nodes;
        int v1,v2,length,i,j,n;
        clrscr();
        printf("\n\t Prim'S Algorithm\n");
        printf("\n Enter Number of Nodes in the Graph");
        scanf("%d",&nodes);
        printf("\n Enter number of edges in
        Graph"); scanf("%d",&n);
        for(i=0;i<nodes;i++) //initialize Graph
        for(j=0;j<nodes;j++)
        G[i][j]=0;
        printf("\n Enter edges and weights \n");
        for(i=1;i<=n;i++)
        {
                printf("\n Enter Edge by v1 and v2:");
                printf("[Read the graph from starting node 1]");
                scanf("%d%d",&v1,&v2);
                printf("\nEnter corresponding weight:");
                scanf("%d",&length);
```

```
                G[v1][v2]=G[v2][v1]=length;
        }
        printf("\n\t");
        Prim(G,nodes);
        getch();
}
```

Output

//Paste your own output after successful execution.

EXPERIMENT NO. 6

Aim: To implement Minimum Spanning Tree algorithm Kruskal's algorithm.

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component.

Below are the steps for finding MST using Kruskal's algorithm

1.Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3.Repeat step#2 until there are (V-1) edges in the spanning tree.

Algorithm

Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in the graph.

```
KRUSKAL (V, E, w)
A ← { } D Set A will ultimately contains the edges of the MST for
each vertex v in V
do MAKE-SET(v)
   sort E into nondecreasing order byweight w
for each (u,v)taken from the sorted list
 do if FIND-SET(u) = FIND-SET(v)
                    then A ← A∪{(u,v)}
UNION(u,v)
Return A
```

Analysis

The edge weight can be compared in constant time. Initialization of priority queue takes O(E l E) time by repeated insertion. At each iteration of while-loop, minimum edge can be removed in O(log E) time, which is O(log V), since graph is simple. The total running time is O((V + E) log V), which is O(E lg V) since graph is simple and connected.

Conclusion:

- Prim's algorithm initializes with a node, whereas Kruskal's algorithm initiates with an edge.
- Prim's algorithms span from one node to another while Kruskal's algorithm select the edges in a way that the position of the edge is not based on the last step.
- In prim's algorithm, graph must be a connected graph while the Kruskal's can function on disconnected graphs too.
- Prim's algorithm has a time complexity of $O(V^2)$, and Kruskal's time complexity is O(logV).

MINIMUM_COST_SPANNING_TREE KRUSKAL'S_ALGORITHM-Code

```
#include<stdio.h>
#define INFINITY
999 typedef struct
Graph
{
int   v1;
int   v2;
int
cost;
}GR;
GR G [20];
int tot_edges,
tot_nodes; void
create();
void
spanning_tree(); int
Minimum(int); void
main()
{
printf("\n\t Graph Creation by adjacency
matrix"); create();
spanning_tree();
}
void create()
{
int k;
printf("\n Enter Total number of
nodes:"); scanf("%d",&tot_nodes);
printf("\n Enter Total number of
edges:"); scanf("%d",&tot_edges);
for(k=0;k<tot_edges;k++)
{
printf("\n Enter Edge in (V1
V2)form:"); scanf("%d%d";,&G [k].v1,
&G[k].v2); printf("\n Enter
Corresponding Cost :");
scanf("%d",&G[k].cost);
}
}
void spanning_tree()
{
int count, k, v1, v2, i, j, tree [10] [10], pos, parent [10];
int sum;
int Find(int v2,int parent[]);
void Union(int i,int j,int parent[]);
count=0;
k=0;
sum=0;
for(i=0;i<tot_nodes;i++)
```

```
parent[i]=i;
while(count!=tot_nodes-
1)
{
pos=Minimum(tot_edges);//finding the minimum cost
edge if(pos==-1)//Perhaps no node in the graph
break;
```

```
v1=G[pos].v1;
v2=G[pos].v2;
i=Find(v1,parent)
;
j=Find(v2,parent)
; if(i!=j)
{
tree[k][0]=v1;//storing the minimum edge in array tree[]
tree[k][1]=v2;
k++;
count++;
sum+=G[pos].cost;//accumulating the total cost of MST

Union(i,j,parent);
}
G[pos].cost=INFINITY;
}
if(count==tot_nodes-1)
{
printf("\n Spanning tree
is..."); printf("\n          \n");
for(i=0;i<tot_nodes-1;i++)
{
Printf("%d",tree[i][0]);
printf(" – ");
printf("%d",
tree[i][1]); printf("]";);
}
printf("\n       ");
printf("\nCost of Spanning Tree is = %d",sum);
}
else
{
printf("There is no Spanning Tree");
}
}
int Minimum(int n)
{
int i,small,pos;
small=INFINIT
Y; pos=-1;
for(i=0;i<n;i++)
{
if(G[i].cost<small)
{
small=G[i].cost;
pos=i;
}
}
return pos;
```

```
}
int Find(int v2,int parent[])
{
```

```
while(parent[v2]!=v2)
{
v2=parent[v2];
}
return v2;
}
void Union(int i,int j,int parent[])
{ if(i<j)
parent[j]=i;
else
parent[i]=j;
}
```

OUTPUT:

Paste your Own output.

EXPERIMENT NO. 7

Aim: a. To implement 0/1 Knapsack using dynamic approach.

Theory:

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

Algorithm:

for w = 0 to W B[0,w]
= 0
fori = 1 to n B[i,0] = 0
fori = 1 to n for w = 0
to W
if wi<= w // item i can be part of the
solution if bi + B[i-1,w-wi] > B[i-1,w]

B[i,w] = bi + B[i-1,w- wi]

else B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // wi>
w

Example: Selection of n=4 items, capacity of knapsack M=8

Item i	Value v_i	Weight w_i
1	15	1
2	10	5
3	9	3
4	5	4

$f(0,g) = 0,\ f(k,0) = 0$

Recursion formula:

$$f(k,g) = \begin{cases} f(k-1,g) & \text{if } w_k > g \\ \max\{v_k + f(k-1,g-w_k),\ f(k-1,g)\} & \text{if } w_k \le g \text{ and } k>0 \end{cases}$$

Solution tabulated:

		Capacity remaining g=0	g=1	g=2	g=3	g=4	g=5	g=6	g=7	g=8
k=0	f(0,g) =	0	0	0	0	0	0	0	0	0
k=1	f(1,g) =	0	**15**	15	15	15	15	15	15	15
k=2	f(2,g) =	0	**15**	15	15	15	15	25	25	25
k=3	f(3,g) =	0	15	15	15	**24**	24	25	25	25
k=4	f(4, g) =	0	15	15	15	24	24	25	25	**29**

Last value: k=n, g=M f = f(n,M) = f(4,8) = 29

Conclusion: Time complexity O(n*W) where n is the number of items and W is the capacity of knapsack.

Code: Implementation of 0/1 Knapsack using Dynamic Programming.

```
#include<stdio.h>
#include<conio.h
> void main()
 {
  int i,w,n,W,profit[20], wt[20];
 void knapSack(int [],int [],int ,int);
 clrscr();
 printf("Enter number of items:\n");
 scanf("%d", &n);
 printf("Enter size of knapsack:");
scanf("%d", &W);
for(i=1;i<=n;i++)
{
 printf("Enter profit and weight of items %d=",i);
scanf("%d", &profit[i]);
scanf("%d",&wt[i]);
}
 knapSack(profit,
wt,n,W); getch();
}
int max(int a, int b)
{
if(a>b)
return a;
else
return b;
}
 void knapSack(int profit[],int wt[], int n,int W)
 {
  int i,w,K[20][20];
  for (i=0; i<=n; i++)
  {
for (w=0; w<=W; w++)
 {
  if
 (i==0||w==0)
 K[i][w] = 0;
  else
  if (wt[i]<=w)
 K[i][w] = max(profit[i]+K[i-1][w-wt[i]], K[i-1][w]);
else
 K[i][w] = K[i-1][w];
 }
}
  for (i=0; i<=n; i++)
  {
   for (w=0; w<=W; w++)
   {
    printf(" %d ",K[i][w]);
```

```
 }
 printf("\n");
}
printf("..............................."
);
printf("\nThe profit of Knacksack is=%d",K[n][W]);
```

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

Output:

Paste your Own output.

EXPERIMENT NO. 8

Aim: To implement LCS problem using dynamic programming approach.

Theory:

Definition 1: Given a sequence X=x1x2...xm, another sequence Z=z1z2...zk is a subsequence of X, if there exists a strictly increasing sequence i1i2...ik of indices of X such that for all j=1,2,...k, we have xij=zj.

Example 1:

If X=abcdefg, Z=abdg is a subsequence of
X. X=abcdefg,
Z=abdg

Definition 2:

Given two sequences X and Y. A sequence Z is a common subsequence of X
and Y if Z is a subsequence of both X and Y.

Example 2: X=abcdefg and Y=aaadgfd. Z=adf is a common subsequence of X and Y.
X=abcdefg
Y=aaaadgfd
Z=adf

Definition 3:

A longest common subsequence of X and Y is a common subsequence of X and
Y with the longest length. (The length of a sequence is the number of letters in the sequence.)
Longest common subsequence may not be unique.

Theorem (Optimal substructure of an LCS)

Let X=x1x2...xm, and Y=y1y2...yn be two sequences, and
Z=z1z2...zk be any LCS of X and Y.

1. If xm=yn, then zk=xm=yn and Z[1..k-1] is an LCS of X[1..m-1] and Y[1..n-1].
2. If xm≠yn, then zk≠xm implies that Z is an LCS of X[1..m-1] and Y.
3. If xm≠yn, then zk≠yn implies that Z is an LCS of X and Y[1..n-1].

The recursive equation

Let c[i,j] be the length of an LCS of X[1...i] and X[1...j].

c[i,j] can be computed as follows:

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } xi=yj, \\ \max\{c[i,j-1],c[i-1,j]\} & \text{if } i,j>0 \text{ and } xi\neq yj. \end{cases}$$

Computing the length of an LCS

• There are n×mc[i,j]'s. So we can compute them in a specific order.

The algorithm to compute an LCS

1. fori=1 to m do

2. c[i,0]=0;
3. for j=0 to n do
4. c[0,j]=0;
5. fori=1 to m do
6. for j=1 to n do
7. {
8. if xi ==yj then
9. c[i,j]=c[i-1,j-1]=1;
10. b[i,j]=1;
11. else if c[i-1,j]>=c[i,j-1] then
12. c[i,j]=c[i-1,j]
13. b[i,j]=2;
14. else c[i,j]=c[i,j-1]
15. b[i,j]=3;
16. }

Constructing an LCS (back-tracking)

- We can find an LCS using b[i,j]'s.
- We start with b[n,m] and track back to some cell b[0,i] or b[i,0].

The algorithm to construct an LCS

1. i=m
2. j=n;
3. ifi==0 or j==0 then exit;
4. if b[i,j]=1 then
{
i=i-1;
j=j-1;
print
"xi";
}
5. if b[i,j]==2 i=i-1
6. if b[i,j]==3 j=j-1
7. Goto Step 3.

Conclusion:
Time Complexity of the above implementation is O(mn).

Code:

```
#include <stdio.h>
#include <string.h>
#include<conio.h>
int i, j, m, n, LCS[20][20],z=0;
```

```
char b[20][20],d[20],S1[20],S2[20];
void print(int i,int j)
  {
```

```
  if(i==0 || j==0)
  return ;
  if(b[i][j]=='c')
  {
  print(i-1,j-1);
  d[z]=S1[i-1];
  z++;
  printf("%c",S1[i-1]);
  }
  else
  if(b[i][j]=='u'
  ) print(i-1,j);
  else
  print(i,j-1);
 }
void lcs()
{
 m = strlen(S1);
 n = strlen(S2);
  for (i = 0; i <= m; i++)
    LCS[i][0]=0;
   for (j = 0; j <= n; j++)
    LCS[0][j]=0;
    for(i=1;i<=m;i++)
  for(j=1;j<=n;j++)
  {
    if (S1[i - 1] == S2[j - 1])
   {
   LCS[i][j] = LCS[i - 1][j - 1] + 1;
   b[i][j]='c';
   }
 else if(LCS[i-1][j]>=LCS[i][j-1])
  {
```

```
   LCS[i][j]=LCS[i-1][j];
   b[i][j]='u';
  } else {
        LCS[i][j]=LCS[i][j-1];
        b[i][j]='l';
        }
       }
    }
void main()
{
int
p,q,rlen=0;
clrscr();
printf("Dynamic Progrmming:Longest Common Subseqeunce>>");
printf("\nEnter First String: \n");
scanf("%s",S1);
printf("\nEnter Second String: \n");
scanf("%s",S2);
lcs();
printf("\n\n");
for(p=0;p<n;p++)
 printf("%c",S2[p]);
 printf("\n");
 for(p=0;p<n;p++)
 printf(" --- ");
 printf("\n");
 for(p=0;p<m;p++)
 {
  printf("%c |\t ",S1[p]);
 for(q=0;q<n;q++)
 {
 printf("%d\t",LCS[p][q]);
 }
```

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

```
 }
printf("\n The Longest Common Subsequence is\"");
print(m,n);
printf("\"");
for(p=0;p<20;p++)
if(d[p]!='\0')
rlen++;
 printf("\n\n Length of Common Subsequence is:%d",rlen);
 getch();
}
```

OUTPUT:

Paste your Own output.

EXPERIMENT NO. 9

Aim:To implement Graph Coloring.

Theory:

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

For the graph-coloring problem we are interested in assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color. The optimization version calls for coloring a graph using the minimum number of colors. The decision version, known as k-coloring, asks whether a graph is colorable using at most k colors. The 3-coloring problem is a special case of the k-coloring problem where k=3 (i.e., we are allowed to use at most 3 colors).

Input:

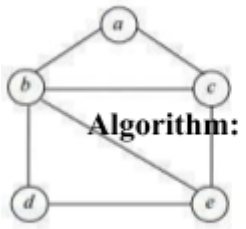
1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

2) An integer mwhich is maximum number of colors that can be used.

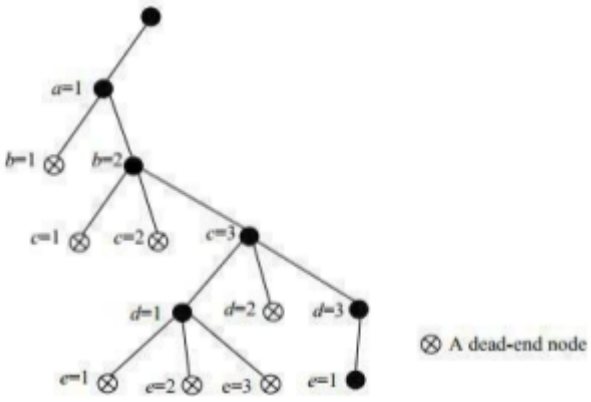
Output:

An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Example



input graph



Algorithm:

If all colors are assigned,

print vertex assigned colors

Else

a. Trying all possible colors, assign a color to the vertex

b. If color assignment is possible, recursively assign colors to next vertices

c. Ifcolor assignment is not possible, de-assign color, return False

Conclusion-

Time Complexity: O(mV). There is a total of O(mV) combinations of colors. The upper bound time complexity remains the same but the average time taken will be less.

Auxiliary Space: O(V). The recursive Stack of the graph coloring function will require O(V) space.

```
#include <stdio.h>
#include<conio.h>
#define MAX_VERTICES 100
int isSafe(int graph[MAX_VERTICES][MAX_VERTICES], int color[], int vertex, int c, int V)
{
int i;
   for (i = 0; i < V; i++) {
if (graph[vertex][i] == 1 && color[i] == c)
          {
   return 0;
}
   }
   return 1;
}
int graphColoringUtil(int graph[MAX_VERTICES][MAX_VERTICES], int color[], int vertex, int V, int m)
 {
int c;
if (vertex == V)
{
return 1;
   }
   for ( c = 1; c <= m; c++)
   {
if (isSafe(graph, color, vertex, c, V))
 {
   color[vertex] = c;
if (graphColoringUtil(graph, color, vertex + 1, V, m))
    {
          return 1;
   }
   color[vertex] = 0;
}
   }
   return 0;
 }
int graphColoring(int graph[MAX_VERTICES][MAX_VERTICES], int V, int m)
 {
   int color[MAX_VERTICES] = {0},i;
   if (graphColoringUtil(graph, color, 0, V, m) == 0)
 {
printf("Solution does not exist\n");
return 0;
   }
   printf("Solution exists: Following are the assigned colors\n");
   for ( i = 0; i < V; i++)
{
printf("Vertex %d -> Color %d\n", i + 1, color[i]);
   }
  return 1;
}

void main()
{
   int V, E, m,i;
```

```
    int graph[MAX_VERTICES][MAX_VERTICES] = {0};
    clrscr();
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);
    printf("Enter the edges (in the format u v):\n");
    for (i = 0; i < E; i++)
     {
int u, v;
scanf("%d %d", &u, &v);
graph[u - 1][v - 1] = 1;
graph[v - 1][u - 1] = 1;
      }
    printf("Enter the number of colors to use: ");
    scanf("%d", &m);
    graphColoring(graph, V, m);
    getch();
}
```

EXPERIMENT NO. 10

Aim: To implement Rabin Karp

Theory:

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that n > m.

Input : txt[] = "THIS IS A TEST TEXT" pat[] ="TEST"
Output : Pattern found at index 10
Input : txt[] = "AABAACAADAABAABA" pat[] =`"AABA"
Output :
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, 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. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.

2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.

Hash at the next shift must be efficiently computable fromthe current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])*and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

*hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod*

q hash(txt[s .. s+m-1]) : Hash value at shift *s*.

hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift *s*+1) *d*:

Number of characters in the alphabet

q:

Aprimenumber

h: d^(m-1)

Conclusion:

The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but itsworst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt [] match with hash value of pat[]. For example pat [] = “AAA” and txt[] = “AAAAAAA”

```
#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;
int p =
0; int t =
0; int h
= 1;
for (i= 0; i < m - 1; i++)
h = (h * d) % q;
// Calculate hash value for pattern and textfor
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[20],pattern[10];
    printf("Enter Text:\n");
    scanf("%s",text);
    printf("Enter
    Pattern:\n");
    scanf("%s",pattern);
int q = 13;
rabinKarp(pattern, text,
q);
}
```

OUTPUT:

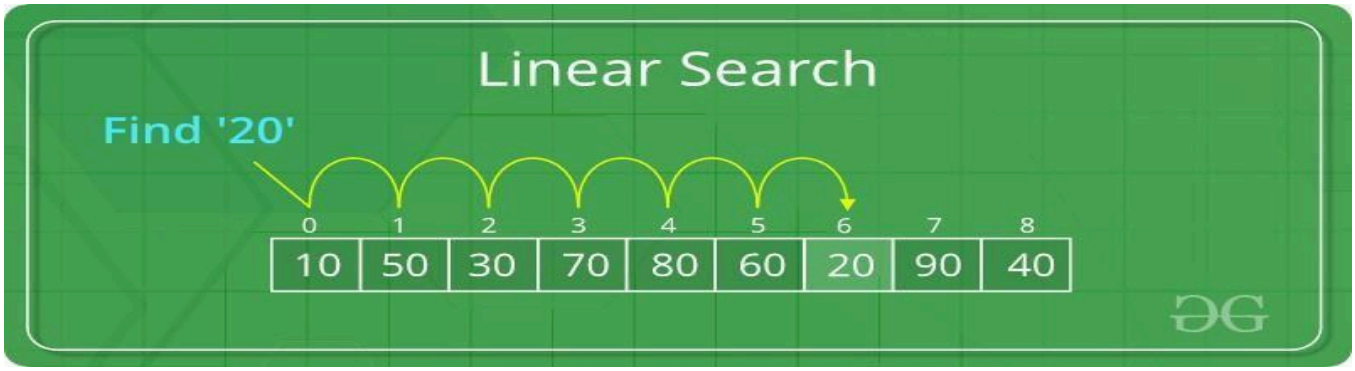
Paste your Own output.

EXPERIMENT NO

.11 AIM :- TO WRITE A PROGRAM FOR LINEAR_SEARCH.

Theory:

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.



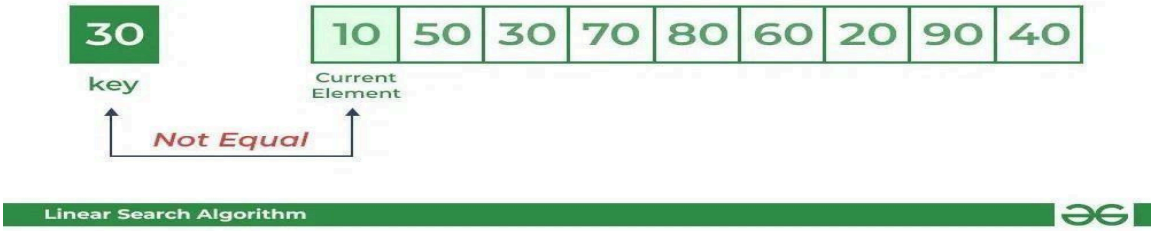
How Does Linear Search Algorithm Work?

In Linear Search Algorithm,

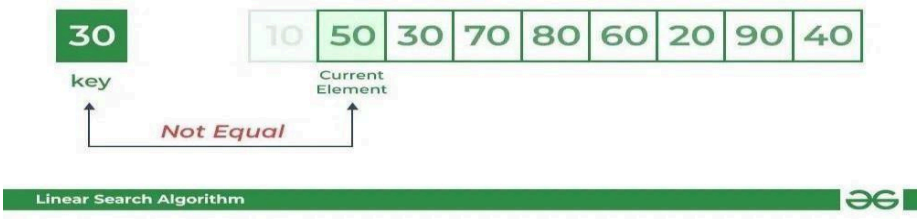
- Every element is considered as a potential match for the key and checked for the same.
- If any element is found equal to the key, the search is successful and the index of that element is returned.
- If no element is found equal to the key, the search yields "No match found".

For example: Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key** = 30

Step 1: *Start from the first element (index 0) and compare* ***key*** *with each element (arr[i]).*

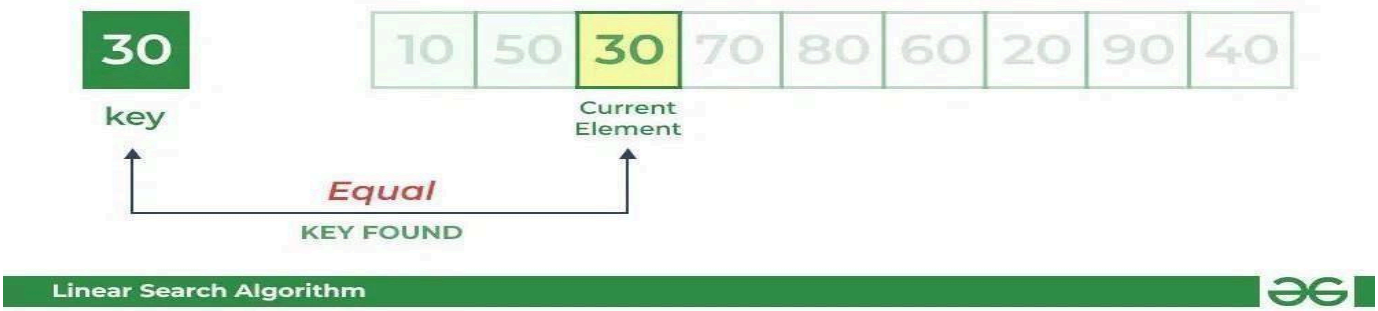


- *Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.*



- *Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.*

Step 2: *Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).*



Conclusion

Time Complexity:

- **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
- **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- **Average Case:** O(N)

Auxiliary Space: O(1) as except for the variable to iterate through the list, no other variable is used.

Advantages of Linear Search:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

When to use Linear Search?

- When we are dealing with a small dataset.
- When you are searching for a dataset stored in contiguous memory.

Code

```
#include <stdio.h>
int linearSearch(int a[], int n, int key) {
    for (int i = 0; i < n; i++)
    {
        if (a[i] == key)
        return i+1;
    }
  return -1;
}
int main()
{
  int a[20],key,n,res,i;
  printf(" Enter the Size of array:\n
  "); scanf("%d",&n);
  printf("Enter elements of the arrayare:\n");
  for (i = 0; i < n; i++)
  scanf("%d", &a[i]);
  printf("The elements of the arrayare:\n");
  for (i = 0; i < n; i++)
  printf("%d\t", a[i]);
  printf("\n Enter Element to be searched:\n");
  scanf("%d", &key);
  res = linearSearch(a, n, key);
   if (res == -1)
  printf("\nElement is not present in the
  array"); else
  printf("\nElement is present at %d position of array",
res); return 0;
}
```

OUTPUT:

Paste your Own output.