Name Shraddha Sanjay Gini
Subject Design and Analysis of Algorithm
Semester V

Year 3
Class
Roll No. 06

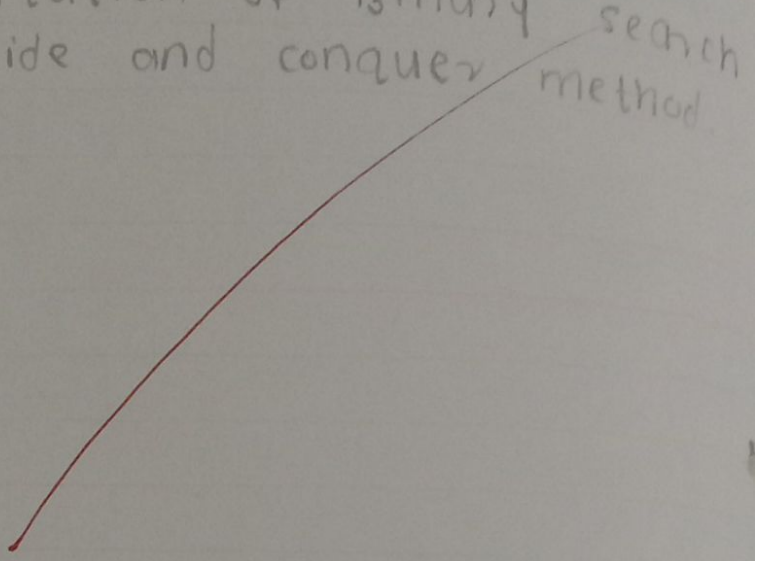
INDEX

S. No.	Experiment Description	Experiment Date	Submission Date	Remark/Sign.
1.	Implementation of Binary search using Divide and conquer method.	2/8/23		
2.	Implementation of merge & quick sort	27/8/23		
5.	Implementation of knapsack problem.	27/8/23		
4.	Finding the cost of minimum spanning tree.	4/9/23		
5.	Write a program of minimum spanning tree using Prim's algorithm.	4/9/23		
6.	Implementation of Huffman code	27/9/23		
7.	Write a program to find shortest path in graph using Dijkstra's algorithm	27/9/23		
8.	Write a program to implement LCS using dynamic programming	4/10/23		
9.	Write a program to find shortest path using	4/10/23		

S. No.	Experiment Description	Experiment Date	Submission Date	Remark/Sig
	Bellman's-ford algorithm.			
10.	Write a program to find all pair shortest path using floyd-warshall algorithm.	11/10/23		
11.	Write a program to implement an application of BFS on an undirected graph.	11/10/23		
12.	Write a program to implement an application of DFS on an undirected graph.	11/10/23		

Practical No. -01

Aim- Implementation of Binary search using Divide and conquer method



Practical No. - 01

Aim - Implementation of Binary search using Divide and conquer method.

Theory - Binary search is an algorithm that works by repeatedly dividing the search interval in half until the target value is found or determined not tobe present in the array. This algorithm is based on the principle of divide and conquer and is highly efficient for large sorted arrays.

The time complexity of binary search is O(logn). where n is the size of input array. However, binary search requires an array tobe sorted, which can add an extra O(nlogn) complexity if the array is unsorted.

Binary search is typically used for large sorted arrays, where it is highly efficient and can provide significant performance gain over linear search.

```
Function binary-search (A, n, T)
is
    L := 0
    R := n - 1
while L ≤ R do
    m := floor ((L+R)/2)
    if A[m] < T then
        L := m+1
    else if A[m] > T then
        R := m-1
    else:
        return m
    return unsuccessful
```

BINARY SEARCH:

CODE:

```
#include<stdio.h>
int main()
{
    int a[100],i,flag=0,lb,ub,mid,pos,item,n;

    printf("Enter the range:");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        printf("Enter the element %d:",i+1);
        scanf("%d",&a[i]);
    }
    printf("Enter the item you want to search:");
    scanf("%d",&item);

    lb=0;

    ub=n-1;

    while(lb<=ub)
    {
        mid=(lb+ub)/2;
        if(item==a[mid])
        {
            flag=1;
            pos=mid;
            break;
        }
        else if(item>a[mid])
        lb=mid+1;
        else
        ub=mid-1;
    }
    if(flag==1)
    printf("Item found at position %d",mid+1);
    else
    printf("Not found");
return 0;
}
```

OUTPUT:

```
Enter the range:5
Enter the element 1:1
Enter the element 2:2
Enter the element 3:3
Enter the element 4:4
Enter the element 5:5
Enter the item you want to search:3
Item found at position 3
--------------------------------
Process exited after 11.54 seconds with return value 0
Press any key to continue . . .
```

Time complexity :

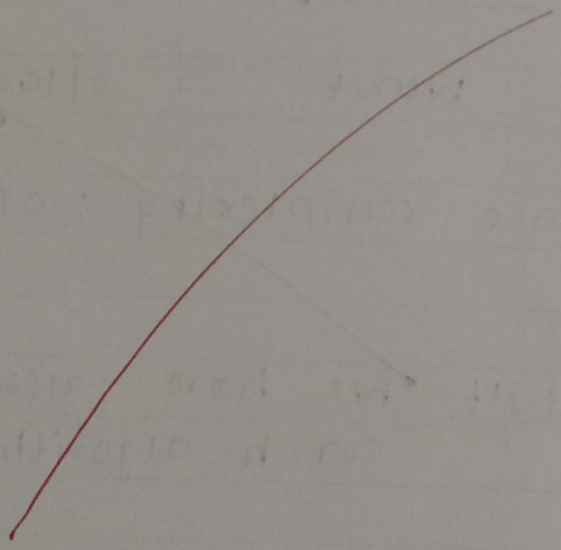
Best	O(1)
Average	O(log n)
Worst	O(log n)

space complexity : O(1)

Result- We have successfully implemented Binary search algorithm.

Practical - 02

Aim - Implementation of merge sort and quick sort using divide and conquer method. Determine time required to sort the elements.



Practical - 02

Aim - Implementation of merge sort and quick sort using Divide and conquer method. Determine the time required to sort the elements.

Theory - Merge sort is a popular comparison based sorting algorithm that follows the divide and conquer strategy to sort an array or a list. It breaks the input array into smaller subarrays, recursively sorts them, and then merges the sorted sub arrays to produce the final sorted array.

The key steps are as follows:

1) Divide: The input array is divided into two equal or nearly equal halves.
2) Conquer: The two halves are sorted recursively using merge sort algorithm.
3) Merge: The sorted halves are merged to produce the final sorted array.

Algorithm:

```
func merge (A, B)
  c = new Array ()
while (A has elem & B has elem)
IF  A[i] > B[i]
      (.add B[i]);
      B + +
else : (.add [A[i]])
```

```
return c

func merge_sort (A)
    m = middle of A.
  B = merge_sort (A[0:middle])
  c = merge_sort (A[middle+1:-1])
return merge (B, C)
```

Time complexity :
Best-case - $\Omega(n \log n)$
Average-case - $\Theta(n \log n)$
worst - $O(n \log n)$

space complexity: $O(n)$

Quick sort : Quick sort is a widely used sorting algorithm known for its efficency and the effectiveness in sorting large datasets.

Algorithm:

```
func quicksort (A, Lo, hi)
  if Lo ≥ 0 && hi ≥ 0 && Lo < hi then
      p := partition (A, Lo, hi)
      Quicksort (A, p + 1, hi)
  func partition (A, Lo, hi)
      pivot := A [floor (hi - Lo) / 2) + Lo]
      i := hi - 1
      j = hi + 1
```

```
/tmp/QI2CCQhAHb.o
Enter the size: 5
Enter the elements of array: 77 11 62 38 65
The sorted array is: 11 38 62 65 77
```

MERGE SORT :

```
#include <stdio.h>
#include <stdlib.h>
void Merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int size1 = mid - left + 1;
    int size2 = right - mid;
    int Left[size1], Right[size2];
    for (i = 0; i < size1; i++)
        Left[i] = arr[left + i];
    for (j = 0; j < size2; j++)
        Right[j] = arr[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < size1 && j < size2)
    {
        if (Left[i] <= Right[j])
        {
            arr[k] = Left[i];
            i++;
        }
        else
        {
            arr[k] = Right[j];
            j++;
        }
        k++;
    }
    while (i < size1)
    {
        arr[k] = Left[i];
        i++;
        k++;
    }
```

```
    while (j < size2)
    {
        arr[k] = Right[j];
        j++;
        k++;
    }
}
void Merge_Sort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        Merge_Sort(arr, left, mid);
        Merge_Sort(arr, mid + 1, right);

        Merge(arr, left, mid, right);
    }}
int main()
{
    int size;
    printf("Enter the size: ");
    scanf("%d", &size);
    int arr[size];
    printf("Enter the elements of array: ");
    for (int i = 0; i < size; i++)
    {
        scanf("%d", &arr[i]);
    }
    Merge_Sort(arr, 0, size - 1);
    printf("The sorted array is: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

output -

```
/tmp/QI2CCQhAHb.o
How many elements are u going to enter?: 6
Enter 6 elements: 8 1 6 3 9 2
Order of Sorted elements:  1 2 3 6 8 9
```

conclusion - we have successfully implemented merge sort and Quick sort

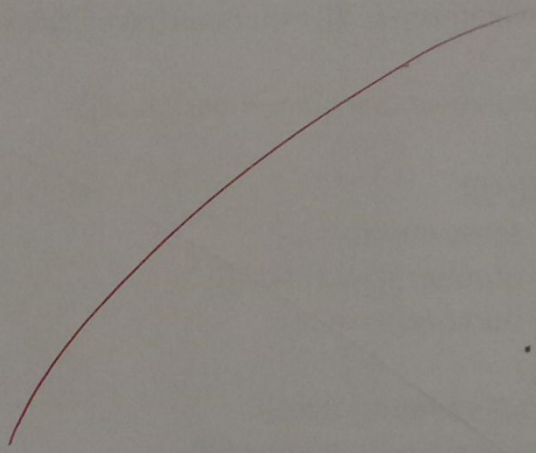
QUICK SORT :

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
  int i, j, pivot, temp;
  if(first<last){
    pivot=first;
    i=first;
    j=last;
    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
      i++;
      while(number[j]>number[pivot])
      j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }}
      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);
    }}
  int main(){
    int i, count, number[25];
    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);
    printf("Enter %d elements: ", count);
    for(i=0;i<count;i++)
    scanf("%d",&number[i]);
    quicksort(number,0,count-1);
    printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
    printf(" %d",number[i]);
    return 0;}
```

conclusion - we have successfully implemented merge sort and quick sort.

Practical-03

Aim- Implementation of knapsack problem using greedy strategy.



Practical - 03

Aim - Implementation of knapsack problem using greedy strategy.

Theory - The knapsack problem is a classic optimization challenge in computer science and mathematics. It involves selecting a subset of items from a given set, each with a specific weight and value in such a way that the total weight of selected item does not exceed a certain limit (the "knapsack" capacity), while maximizing the total value of selected items. In other words, it is about finding the most valuable combination of items that can fill into limited space.

The knapsack problem has variations and is often used to model real world scenarios like resource allocation, project scheduling and more. It is categorized as "combinational optimization" problem and has studied extensively due to its practical applications and its relevance in algorithmic analysis.

The greedy strategy for the knapsack problem choices at each step. In other words, you choose the item with the highest value to weight ratio first, adding as much as it is possible.

```
/tmp/QI2CCQhAHb.o
Enter the no. of objects:- 3
Enter the wts and profits of each object:-
22 7
47 8
67 9
Enter the capacityacity of knapsack:- 50 10
The result vector is:- 1.000000 0.595745    0.000000
Maximum profit is:- 11.765957
```

KNAPSACK PROBLEM CODE :

```
# include<stdio.h>
void knapsack(int n, float weight[], float profit[], float capacity) {
  float x[20], tp = 0;
  int i, j, u;
  u = capacity;
  for (i = 0; i < n; i++)
    x[i] = 0.0;
  for (i = 0; i < n; i++) {
    if (weight[i] > u)
      break;
    else {
      x[i] = 1.0;
      tp = tp + profit[i];
      u = u - weight[i];
    }
  }
  if (i < n)
    x[i] = u / weight[i];
  tp = tp + (x[i] * profit[i]);
  printf("\nThe result vector is:- ");
  for (i = 0; i < n; i++)
    printf("%f\t", x[i]);
  printf("\nMaximum profit is:- %f", tp);
}
int main() {
  float weight[20], profit[20], capacity;
  int num, i, j;
  float ratio[20], temp;
  printf("\nEnter the no. of objects:- ");
  scanf("%d", &num);
  printf("\nEnter the wts and profits of each object:- ");
  for (i = 0; i < num; i++) {
    scanf("%f %f", &weight[i], &profit[i]);
  }
```

conclusion - we have successfully implemented the knapsack problem using greedy strategy.

```
printf("\nEnter the capacityacity of knapsack:- ");
scanf("%f", &capacity);
for (i = 0; i < num; i++) {
  ratio[i] = profit[i] / weight[i];
}
for (i = 0; i < num; i++) {
  for (j = i + 1; j < num; j++) {
    if (ratio[i] < ratio[j]) {
      temp = ratio[j];
      ratio[j] = ratio[i];
      ratio[i] = temp;
      temp = weight[j];
      weight[j] = weight[i];
      weight[i] = temp;
      temp = profit[j];
      profit[j] = profit[i];
      profit[i] = temp;
    } } }
  knapsack(num, weight, profit, capacity);
  return(0);
}
```

conclusion: we have sucessfully implemented the knapsack problem using greedy strategy.

```
printf("\nEnter the capacityacity of knapsack:- ");
scanf("%f", &capacity);
for (i = 0; i < num; i++) {
  ratio[i] = profit[i] / weight[i];
}
for (i = 0; i < num; i++) {
  for (j = i + 1; j < num; j++) {
    if (ratio[i] < ratio[j]) {
      temp = ratio[j];
      ratio[j] = ratio[i];
      ratio[i] = temp;
      temp = weight[j];
      weight[j] = weight[i];
      weight[i] = temp;
      temp = profit[j];
      profit[j] = profit[i];
      profit[i] = temp;
    } } }
  knapsack(num, weight, profit, capacity);
  return(0);
}
```

conclurion: we have successfully implemented the knapsack problem using greedy strategy.

Practical - 04

Aim- Write a program of minimum cost spanning tree using Kuskal's algorithm

Practical-04

Aim - Write a program of minimum cost spanning tree using Kruskal's algorithm.

Theory - A minimum cost spanning tree is a subset of edges in a connected graph that connects all the vertices while minimi-zing the total cost. The cost associated with each edge can represent various measures such as a distance, weight or any relevant metric. The goal of finding MCST is to construct a tree that spans all the vertices with least possible sum of edge costs.

Kruskal's algorithm is a greedy algorithm used to find MCST of a connected path. Here is how it works:

(1) Initialization: Create a forest where each vertex is a seperate

(2) Sorting: Sort all edges in graph in ascending order based on their weights.

(3) Edge selection: Iterate through the sorted edges for each edge, check if adding it to the MCST will form a cycle.

(4) Termination: Repeat step 3 until the MCST contained V-1 edges.

output-

```
/tmp/WwzpJRgvbm.o
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

KRUSKAL ALGORITHM CODE :

```
#include <stdio.h>
#include <stdlib.h>
int comparator(const void* p1, const void* p2)
{
        const int(*x)[3] = p1;
        const int(*y)[3] = p2;
        return (*x)[2] - (*y)[2];
}
void makeSet(int parent[], int rank[], int n)
{
        for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 0;
        }
}
int findParent(int parent[], int component)
{
        if (parent[component] == component)
                return component;
        return parent[component]
                = findParent(parent, parent[component]);
}
void unionSet(int u, int v, int parent[], int rank[], int n)
{
        u = findParent(parent, u);
        v = findParent(parent, v);
        if (rank[u] < rank[v]) {
                parent[u] = v;
        }
        else if (rank[u] > rank[v]) {
                parent[v] = u;
        }
        else {
                parent[v] = u;
                rank[u]++;
```

conclusion- we have successfully written a program to find minimum cost spanning tree using kruskal's algorithm

```
        }
    }
void kruskalAlgo(int n, int edge[n][3])
{
        qsort(edge, n, sizeof(edge[0]), comparator);
        int parent[n];
        int rank[n];
        makeSet(parent, rank, n);
        int minCost = 0;
        printf("Following are the edges in the constructed MST\n");
        for (int i = 0; i < n; i++) {
                int v1 = findParent(parent, edge[i][0]);
                int v2 = findParent(parent, edge[i][1]);
                int wt = edge[i][2];
                if (v1 != v2) {
                        unionSet(v1, v2, parent, rank, n);
                        minCost += wt;
                        printf("%d -- %d == %d\n", edge[i][0],
                                edge[i][1], wt);
                }
        }
        printf("Minimum Cost Spanning Tree: %d\n", minCost);
}
int main()
{
        int edge[5][3] = { { 0, 1, 10 },
                           { 0, 2, 6 },
                           { 0, 3, 5 },
                           { 1, 3, 15 },
                           { 2, 3, 4 } };

        kruskalAlgo(5, edge);
        return 0;
}
```

conclusion - we have successfully written a program to find minimum cost spanning tree using kruskal's algorithm.

Practical-05

Aim- Write a program of minimum spanning tree using prim's algorithm.

Theory- The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

The working of Prim's algorithm can be described by using the following steps:

Step 1 - Determine an arbitrary vertex as the starting vertex of the MST.

Step 2 - Follow steps 3 to 5 till there are vertices that are not included in MST (known as fringe vertex).

Step 3 - Find edges connecting any tree vertex with fringe vertex.

Step 4 - Find the minimum among these edges.

output -

```
/tmp/p4pIDoBSAq.o
Edge    Weight
0 - 1     2
1 - 2     3
0   3     6
1   4     5
Total Path Weight: 16
```

step 5- Add the chosen edge to the MST if it does not form any cycle.

step 6- Return the MST and exit.

code-

Prim's Algorithm code :

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define V 5
int minKey(int key[], bool mstSet[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++) {
      if (!mstSet[v] && key[v] < min) {
         min = key[v];
         min_index = v;
      }
   }
   return min_index;
}
int calculateTotalWeight(int parent[], int graph[V][V]) {
   int totalWeight = 0;
   for (int i = 1; i < V; i++) {
      totalWeight += graph[i][parent[i]];
   }
   return totalWeight;
}
void primMST(int graph[V][V]) {
   int parent[V];
   int key[V];
   bool mstSet[V];
   for (int i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = false;
```

```
}
key[0] = 0;
parent[0] = -1;
for (int count = 0; count < V - 1; count++) {
   int u = minKey(key, mstSet);
   mstSet[u] = true;
   for (int v = 0; v < V; v++) {
      if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
         parent[v] = u;
         key[v] = graph[u][v];
      }
   }
}
```

```
  printf("Edge   Weight\n");
  for (int i = 1; i < V; i++) {
     printf("%d - %d    %d\n", parent[i], i, graph[i][parent[i]]);
  }
  int totalWeight = calculateTotalWeight(parent, graph);
  printf("Total Path Weight: %d\n", totalWeight);
}
int main() {
   int graph[V][V] = {
      {0, 2, 0, 6, 0},
      {2, 0, 3, 8, 5},
      {0, 3, 0, 0, 7},
      {6, 8, 0, 0, 9},
      {0, 5, 7, 9, 0}
   };
   primMST(graph);
   return 0;
}
```

conclusion - We have successfully written a program of minimum spanning tree using prim's algorithm.

Practical - 06

Aim - Implementation of Huffman code using greedy strategy.

Theory - Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters. lengths of the assigned codes are based on the frequencies of corresponding characters. The variable-length codes assigned to input characters are Prefix codes. means the codes are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman coding makes sure that there is no ambiguity when decoding the generated bitstream.
There are mainly two major parts in Huffman coding :

(i) Build a Huffman Tree from the input characters.

(ii) Traverse the Huffman Tree and assign codes to characters.

Huffman coding is done with the help of the following steps :

step 1 - calculate the frequency of each charact

in the string.

step 2 - Sort the characters in increasing order of the frequency. These are sorted in a priority queue Q.

step 3 - Make each unique character as a leaf node.

step 4 - Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of z as the sum of the above two minimum frequencies.

step 5 - Remove these two minimum frequencies from Q and add the sum into the list of frequencies.

step 6 - Insert node z into the tree.

step 7 - Repeat steps 3 to 5 for all the characters

step 8 - For each non-leaf node, assign 0 to the left edge and 1 to the right edge.

output -

```
/tmp/QlZlqtlqmQ.o
Huffman Codes:
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_TREE_NODES 256
struct Node {
   char data;
   unsigned freq;
   struct Node *left, *right;
};
struct MinHeap {
   unsigned size;
   unsigned capacity;
   struct Node** array;
};
struct Node* newNode(char data, unsigned freq) {
   struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
   temp->left = temp->right = NULL;
   temp->data = data;
   temp->freq = freq;
   return temp;
}
struct MinHeap* createMinHeap(unsigned capacity) {
   struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
   minHeap->size = 0;
   minHeap->capacity = capacity;
   minHeap->array = (struct Node**)malloc(capacity * sizeof(struct Node*));
   return minHeap;
}
void swapNode(struct Node** a, struct Node** b) {
   struct Node* t = *a;
   *a = *b;
   *b = t;
}
void minHeapify(struct MinHeap* minHeap, int idx) {
   int smallest = idx;
   int left = 2 * idx + 1;
   int right = 2 * idx + 2;

   if (left < minHeap->size &&
      minHeap->array[left]->freq < minHeap->array[smallest]->freq)
      smallest = left;
```

```
 if (right < minHeap->size &&
    minHeap->array[right]->freq < minHeap->array[smallest]->freq)
    smallest = right;
 if (smallest != idx) {
    swapNode(&minHeap->array[idx], &minHeap->array[smallest]);
    minHeapify(minHeap, smallest);
  }
}
int isSizeOne(struct MinHeap* minHeap) {
   return (minHeap->size == 1);
}
struct Node* extractMin(struct MinHeap* minHeap) {
   struct Node* temp = minHeap->array[0];
   minHeap->array[0] = minHeap->array[minHeap->size - 1];
   --minHeap->size;
   minHeapify(minHeap, 0);
   return temp;
 }
 void insertMinHeap(struct MinHeap* minHeap, struct Node* node) {
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
       minHeap->array[i] = minHeap->array[(i - 1) / 2];
       i = (i - 1) / 2;
     }
     minHeap->array[i] = node;
  }
    void buildMinHeap(struct MinHeap* minHeap) {
       int n = minHeap->size - 1;
       int i;
       for (i = (n - 1) / 2; i >= 0; --i)
          minHeapify(minHeap, i);
     }
     struct Node* buildHuffmanTree(char data[], int freq[], int size) {
        struct Node *left, *right, *top;
        struct MinHeap* minHeap = createMinHeap(size);
        for (int i = 0; i < size; ++i)
           insertMinHeap(minHeap, newNode(data[i], freq[i]));
        buildMinHeap(minHeap);
        while (!isSizeOne(minHeap)) {
           left = extractMin(minHeap);
           right = extractMin(minHeap);
           top = newNode('$', left->freq + right->freq);
           top->left = left;
```

```
    top->right = right;
    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}
void printCodes(struct Node* root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printCodes(root->right, arr, top + 1);
  }
  if (!root->left && !root->right) {
    printf("%c: ", root->data);
    for (int i = 0; i < top; ++i) {
      printf("%d", arr[i]);
    }
    printf("\n");
  }
}
void HuffmanCodes(char data[], int freq[], int size) {
  struct Node* root = buildHuffmanTree(data, freq, size);
  int arr[MAX_TREE_NODES], top = 0;
  printCodes(root, arr, top);
}
int main() {
  char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
  int freq[] = { 5, 9, 12, 13, 16, 45 };
  int size = sizeof(data) / sizeof(data[0]);
  printf("Huffman Codes:\n");
  HuffmanCodes(data, freq, size);
  return 0;
}
```

PAGE NO.:

conclusion - we have successfully implemented the Huffman code using greedy strategy.

Practical - 07

Aim - Write a program to find the shortest path in graph using Dijkstra's algorithm.

Theory - Dijkstra's algorithm is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex with the and iteratively selects the unvisited vertex, with the smallest tentative distance from the source.

It then visits the neighbors of this vertex and updates their tentative distance of a shorter path, found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

Algorithm -

Step 1 - Mark the source node with a current distance of 0 and rest with infinity.

Step 2 - Set the non-visited node with the smallest current distance as the

output-

```
/tmp/lll NxloyDK.o
Vertex          Distance from Source
0               0
1               4
2               12
3               19
4               21
5               11
6               9
7               8
8               14
```

current node.

step 3 - For each neighbor, v of the current nodes add the current distance of the adjacent node with the weight of the edge connecting 0 → 1. If it is smaller than the current distance of node, set it as the new current distance of v.

step 4 - mark the current node 1 as visited.

step 5 - Go to step 2 if there are any nodes unvisited.

CODE :

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 9
int minDistance(int dist[], bool sptSet[])
{
        int min = INT_MAX, min_index;
        for (int v = 0; v < V; v++)
                if (sptSet[v] == false && dist[v] <= min)
                        min = dist[v], min_index = v;
        return min_index;
    }
```

```
void printSolution(int dist[])
{
        printf("Vertex \t\t Distance from Source\n");
        for (int i = 0; i < V; i++)
                printf("%d \t\t\t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[V][V], int src)
{
        int dist[V];
        bool sptSet[V];
        for (int i = 0; i < V; i++)
                dist[i] = INT_MAX, sptSet[i] = false;
        dist[src] = 0;
        for (int count = 0; count < V - 1; count++) {
                int u = minDistance(dist, sptSet);
                sptSet[u] = true;
                for (int v = 0; v < V; v++)
                        if (!sptSet[v] && graph[u][v]
                                && dist[u] != INT_MAX
                                && dist[u] + graph[u][v] < dist[v])
                                dist[v] = dist[u] + graph[u][v];}
        printSolution(dist);
}
int main()
{
        int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                { 0, 0, 4, 14, 10, 0, 2, 0, 0 },

                                { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                                { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
        dijkstra(graph, 0);
        return 0;
}
```

PAGE NO.:

conclusion - we have successfully written a program to find the shortest path in graph using Dijkstra's algorithm.

PAGE NO.:

Practical - 08

Aim - Write a program to implement longest common subsequence (LCS) problem using dynamic programming.

Theory - The longest common subsequence (LCS) is defined as the longest subsequence that is common to all given subsequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If s_1 and s_2 are the two given sequences then, z is the common subsequence of s_1 and s_2, if the z is a subsequence of both s_1 and s_2. Furthermore, z must be a strictly increasing sequence of the indices of both s_1 and s_2.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in z.

If

$s_1 = \{B, C, D, A, A, C, D\}$

then, $\{A, D, B\}$ cannot be a subsequence of s_1 as the order of the element is not the same (i.e. not strictly increasing sequence).

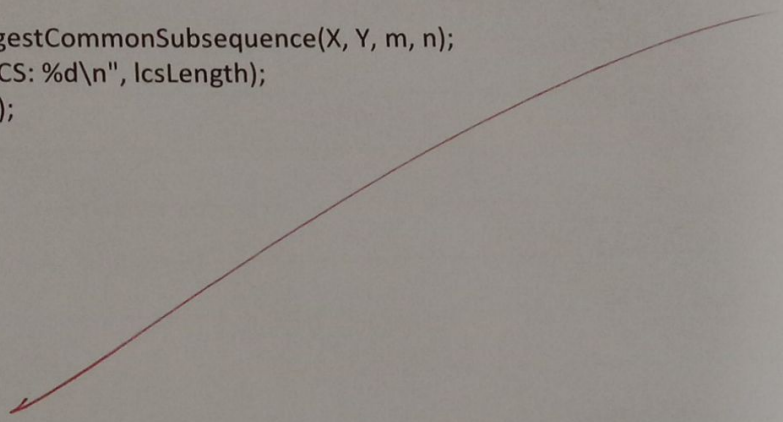
output :-

```
/tmp/IIII NxIayDK.o
Length of LCS. 4
Longest Common Subsequence: GTAD
```

CODE :

```
#include <stdio.h>
#include <string.h>
int longestCommonSubsequence(char *X, char *Y, int m, int n) {
    int dp[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = (dp[i - 1][j] > dp[i][j - 1]) ? dp[i - 1][j] : dp[i][j - 1];
            }
        }
    }
    return dp[m][n];
}
void printLCS(char *X, char *Y, int m, int n) {
    int dp[m + 1][n + 1];
    int len = longestCommonSubsequence(X, Y, m, n);
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = (dp[i - 1][j] > dp[i][j - 1]) ? dp[i - 1][j] : dp[i][j - 1];
            }
        }
    }
    char lcs[len + 1];
    lcs[len] = '\0';
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs[len - 1] = X[i - 1];
            i--;
            j--;
            len--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
```

```
            j--;
        }
    }
    printf("Longest Common Subsequence: %s\n", lcs);
}
int main() {
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    int m = strlen(X);
    int n = strlen(Y);
    int lcsLength = longestCommonSubsequence(X, Y, m, n);
    printf("Length of LCS: %d\n", lcsLength);
    printLCS(X, Y, m, n);
    return 0;
}
```



PAGE NO.:

conclusion- we have successfully written a program to implement longest common subsequence (LCS) problem using dynamic programing.

Practical-09

Aim - Write a program to find shortest path using Bellman's ford algorithm.

Theory - Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in graph. This algorithm can be used on both weighted and unweighted graphs.

The Bellman-Ford algorithm's primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on "Principle of Relaxation".

Algorithm:

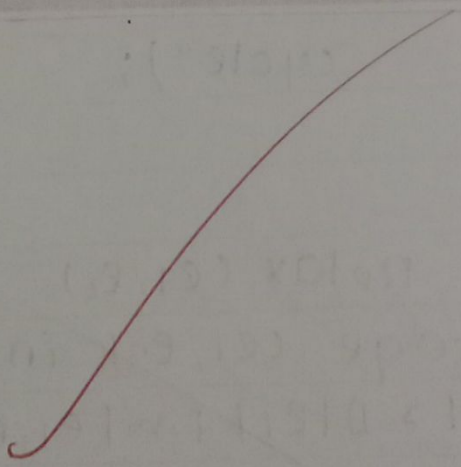
Data - Given a directed graph G(V,E), the starting vertex s, and weight w of each edge.

```
D[s] = 0;
R = V - S;
c = cardinality(V);
for each vertex k ∈ R do
```

```
    D[k] = ∞;
end
for each vertex i = 1 to (c-1) do
    for each edge (e1, e2) ∈ E do
        Relax (e1, e2);
    end
end
for each edge (e1, e2) ∈ E do
    if D[e2] > D[e1] + w[e1, e2] then
        print ("Graph contains negative weight
              cycle");
    end
end
Procedure Relax (e1, e2)
for each edge (e1, e2) in E do
    if D[e2] > D[e1] + w[e1, e2] then
        D[e2] = D[e1] + w[e1, e2];
    end
end
```

output -

```
/tmp/QiZiqtigmQ.o
Shortest paths from source vertex 0:
To vertex 1, Distance: -1, Path: 0 -> 0 -> 1 ->
To vertex 2, Distance: 2, Path: 0 -> 0 -> 1 -> 2 ->
To vertex 3, Distance: -2, Path: 0 -> 0 -> 1 -> 4 -> 3 ->
To vertex 4, Distance: 1, Path: 0 -> 0 -> 1 -> 4 ->
```



CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
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(E * sizeof(struct Edge));
   return graph;
}
void relax(int dist[], int parent[], struct Edge edge) {
   int u = edge.src;
   int v = edge.dest;
   int weight = edge.weight;
   if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
      dist[v] = dist[u] + weight;
      parent[v] = u;
   }}
void printPath(int parent[], int v) {
   if (v < 0)
      return;
   printPath(parent, parent[v]);
   printf("%d -> ", v);
}
void bellmanFord(struct Graph* graph, int src) {
   int V = graph->V;
   int E = graph->E;
   int dist[V];
   int parent[V];
   for (int i = 0; i < V; i++) {
      dist[i] = INT_MAX;
      parent[i] = -1;
   }
   dist[src] = 0;
   for (int i = 1; i < V; i++) {
      for (int j = 0; j < E; j++) {
```

```
      relax(dist, parent, graph->edge[j]);
    }
  }
  printf("Shortest paths from source vertex %d:\n", src);
  for (int i = 0; i < V; i++) {
     if (i != src) {
        printf("To vertex %d, Distance: %d, Path: %d -> ", i, dist[i], src);
        printPath(parent, i);
        printf("\n");
      } }}
int main() {
    int V = 5;
    int E = 8;
     struct Graph* graph = createGraph(V, E);
     graph->edge[0].src = 0;
     graph->edge[0].dest = 1;
     graph->edge[0].weight = -1;
     graph->edge[1].src = 0;
     graph->edge[1].dest = 2;
     graph->edge[1].weight = 4;
     graph->edge[2].src = 1;
     graph->edge[2].dest = 2;
     graph->edge[2].weight = 3;
     graph->edge[3].src = 1;
     graph->edge[3].dest = 3;
     graph->edge[3].weight = 2;
     graph->edge[4].src = 1;
     graph->edge[4].dest = 4;
     graph->edge[4].weight = 2;
     graph->edge[5].src = 3;
     graph->edge[5].dest = 2;
     graph->edge[5].weight = 5;
     graph->edge[6].src = 3;
     graph->edge[6].dest = 1;
     graph->edge[6].weight = 1;
     graph->edge[7].src = 4;
     graph->edge[7].dest = 3;
     graph->edge[7].weight = -3;
     int src = 0;
     bellmanFord(graph, src);
     return 0;
   }
```

PAGE NO.:

conclusion- we have successfully written a program to find shortest path using Bellman's ford algorithm.

PAGE NO.:

Practical -10

Aim- Write a program to find all pair shortest path using Floyd-warshall algorithm.

Theory- The Floyd Warshall algorithm is for solving all pairs of shortest-path problems. The problem is to find the shortest distance between every pair of vertices in a given edge-weighted directed graph. This algorithm follows the dynamic programming approach to find the shortest path.

Algorithm:

```
void floydwarshall()
{
int cost [N][N];
int i,j,k;
for(i=0;i<N;i++)
for(j=0;j<N;j++)
cost[i][j] = cost Mat[i][j];
for[k=0; k<N; k++)
{
for(i=0; i<N; i++)
for(j=0; j<N; j++)
if[cost[i][j] > cost[i][k] + cost[k][j];
cost[i][j] = cost [i][k] + cost[k][i];
} // display the matix cost[N][N]
}
```

```
/tmp/QlZlqtlgmQ.o
All Pair Shortest Paths:
0    5    8    9
INF  0    3    4
INF  INF  0    1
INF  INF  INF  0
```

CODE :

```
#include <stdio.h>
#include <limits.h>
#define V 4
void floydWarshall(int graph[V][V]) {
  int dist[V][V];
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      dist[i][j] = graph[i][j];
    } }
  for (int k = 0; k < V; k++) {
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX &&
          dist[i][k] + dist[k][j] < dist[i][j]) {
          dist[i][j] = dist[i][k] + dist[k][j];
        } } } }
  printf("All Pair Shortest Paths:\n");
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      if (dist[i][j] == INT_MAX) {
        printf("INF\t");
      } else {
        printf("%d\t", dist[i][j]);
      }
    }
    printf("\n");
  }}
int main() {
  int graph[V][V] = {
    {0, 5, INT_MAX, 10},
    {INT_MAX, 0, 3, INT_MAX},
    {INT_MAX, INT_MAX, 0, 1},
    {INT_MAX, INT_MAX, INT_MAX, 0}  };
  floydWarshall(graph);
  return 0;
}
```

CODE :

```
#include <stdio.h>
#include <limits.h>
#define V 4
void floydWarshall(int graph[V][V]) {
   int dist[V][V];
   for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
         dist[i][j] = graph[i][j];
      } }
   for (int k = 0; k < V; k++) {
      for (int i = 0; i < V; i++) {
         for (int j = 0; j < V; j++) {
            if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX &&
               dist[i][k] + dist[k][j] < dist[i][j]) {
               dist[i][j] = dist[i][k] + dist[k][j];
            } } } }
    printf("All Pair Shortest Paths:\n");
    for (int i = 0; i < V; i++) {
       for (int j = 0; j < V; j++) {
          if (dist[i][j] == INT_MAX) {
             printf("INF\t");
          } else {
             printf("%d\t", dist[i][j]);
          }
       }
       printf("\n");
    }}
 int main() {
    int graph[V][V] = {
       {0, 5, INT_MAX, 10},
       {INT_MAX, 0, 3, INT_MAX},
       {INT_MAX, INT_MAX, 0, 1},
       {INT_MAX, INT_MAX, INT_MAX, 0}   };
    floydWarshall(graph);
    return 0;
 }
```

PAGE NO.:

conclusion - we have successfully written a program to find all pair shortest path using Floyd-Warshall algorithm.

PAGE NO:

Practical - 11

Aim- Write a program to implement an application of BFS on an undirected graph.

Theory- The Breadth First search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at next depth level.

Breadth First Traversal for a graph is similar to the Breadth first Traversal of a tree. The only difference here is, that, unlike trees, graph may contain cycles. some may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- visited
- not visited

Pseudocode:

```
BFS (G, s)
    let Q be queue.
    Q.enqueue(s)
    mark s as visited.
    while (Q is not empty)
        v = Q.dequeue()
```

for all neighbours w of v in Graph G
if w is not visited
Q.enqueue (w)
mark w as visited

output-

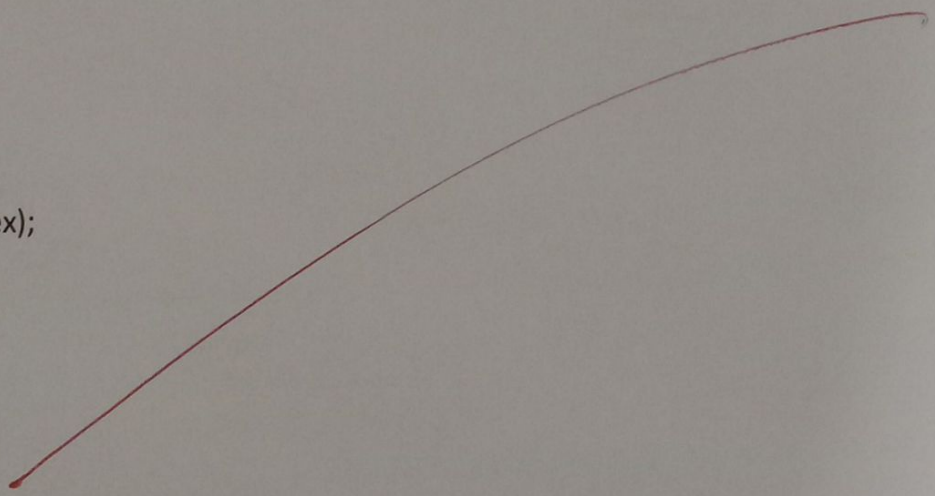
```
/tmp/QlZlqt1gmQ.o
Breadth-First Traversal starting from vertex 0: 0 2 1 4 3 5
```

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100
struct Queue {
    int front, rear, size;
    unsigned capacity;
    int* array;
};
struct Queue* createQueue(unsigned capacity) {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = (int*)malloc(queue->capacity * sizeof(int));
    return queue;
}
bool isEmpty(struct Queue* queue) {
    return (queue->size == 0);
}
bool isFull(struct Queue* queue) {
    return (queue->size == queue->capacity);
}
void enqueue(struct Queue* queue, int item) {
    if (isFull(queue)) return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size++;
}
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) return -1;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size--;
    return item;
}
struct AdjListNode {
    int dest;
    struct AdjListNode* next;
};
struct AdjList {
    struct AdjListNode* head;
};
```

```
struct Graph {
    int V;
    struct AdjList* array;
};
struct AdjListNode* newAdjListNode(int dest) {
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
    for (int i = 0; i < V; i++) {
        graph->array[i].head = NULL;
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}
void BFS(struct Graph* graph, int src) {
    bool* visited = (bool*)malloc(graph->V * sizeof(bool));
    for (int i = 0; i < graph->V; i++) {
        visited[i] = false;
    }
    struct Queue* queue = createQueue(graph->V);
    visited[src] = true;
    enqueue(queue, src);
    printf("Breadth-First Traversal starting from vertex %d: ", src);
    while (!isEmpty(queue)) {
        int vertex = dequeue(queue);
        printf("%d ", vertex);
        struct AdjListNode* currentNode = graph->array[vertex].head;
        while (currentNode != NULL) {
            int adjVertex = currentNode->dest;
            if (!visited[adjVertex]) {
                visited[adjVertex] = true;
```

```
            enqueue(queue, adjVertex);
        }
        currentNode = currentNode->next;
    }
  }
  printf("\n");
  free(visited);
  free(queue->array);
  free(queue);
}
int main() {
  int V = 6;
  struct Graph* graph = createGraph(V);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 3);
  addEdge(graph, 1, 4);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 5);
  int sourceVertex = 0;
  BFS(graph, sourceVertex);
  return 0;
}
```



PAGE NO.:

conclusion - we have successfully written a program to implement an application of BFS on an undirected graph.

Practical - 10a

Aim- Write a program to implement an application of DFS on an undirected graph.

Theory - Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

The DFS algorithm is a recursive algorithm that used the idea of backtracking. Here, the word backtrack means that when you are moving forward and there are no more node along the current path, you move backwards on the same path to find nodes to traverse.

Pseudocode:

```
DFS-iterative (G, s):
    let S be stack
    S.push(s)
    mark s as visited.
    while (S is not empty):
        v = S.top()
        S.pop()
        for all neighbours w of v in Graph G:
            if w is not visited:
                S.push(w)
```

mark w as visited

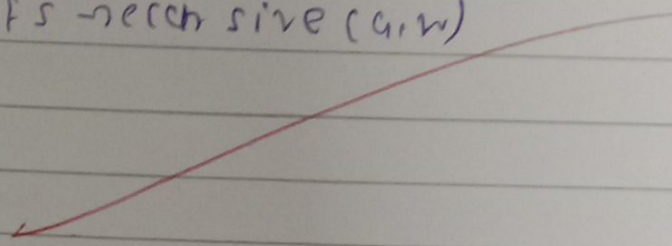
DFS-recursive (G, s):

mark s as visited

for all neighbours w of s in Graph G:

if w is not visited:

DFS-recursive (G, w)



CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
struct AdjListNode {
    int dest;
    struct AdjListNode* next;
};
struct AdjList {
    struct AdjListNode* head;
};
struct Graph {
    int V;
    struct AdjList* array;
};
struct AdjListNode* newAdjListNode(int dest) {
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
    for (int i = 0; i < V; i++) {
        graph->array[i].head = NULL;
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}
void DFSUtil(struct Graph* graph, int vertex, bool visited[]) {
    visited[vertex] = true;
    printf("%d ", vertex);
    struct AdjListNode* currentNode = graph->array[vertex].head;
    while (currentNode != NULL) {
        int adjVertex = currentNode->dest;
```

```
        if (!visited[adjVertex]) {
            DFSUtil(graph, adjVertex, visited);
        }
        currentNode = currentNode->next;
    }
}
void DFS(struct Graph* graph, int src) {
    bool* visited = (bool*)malloc(graph->V * sizeof(bool));
    for (int i = 0; i < graph->V; i++) {
        visited[i] = false;
    }
    printf("Depth-First Traversal starting from vertex %d: ", src);
    DFSUtil(graph, src, visited);
    printf("\n");
    free(visited);
}
int main() {
    int V = 6;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 5);
    int sourceVertex = 0;
    DFS(graph, sourceVertex);
    return 0;
}
```

Conclusion - We have successfully written a program to implement an application of DFS on an undirected graph.

PAGE NO.:

Conclusion - We have successfully written a program to implement an application of DFS on an undirected graph.

PAGE NO.

Conclusion - We have successfully written a program to implement an application of DFS on an undirected graph.