ABESIT

ABES Institute of Technology Ghaziabad

Affiliated to Dr. A.P.J. AKTU, Lucknow

LAB FILE (2024-25)

Department of Computer Science & Engineering

Subject Name: Design Analysis of Algorithm lab
Subject Code: BCS-553

Semester: 5th

LIST OF PROGRAMS

Lab No.	Ref. Pt.	Name of program
1	1	Program for recursive linear search
2	2	Program for recursive binary search
3	3	Program for insertion sort
4	4	Program for selection sort
5	5	Program for merge sort
6	6	Program for quick sort
7	7	Program for heap sort
8	8	Program for shell sort
9	9	Program for minimum spanning tree using Kruskal's algorithm
10	10	Program for minimum spanning tree using Prim's algorithm
11	11	Program for N-Queen problem using backtracking
12	12	Program for Knap-Sack problem using greedy approach
13	13	Program for Dijkstra's algorithm
14	14	Program for travelling salesman person

PROGRAM 1

Objective: - Program for Recursive Linear Search.

Brief Theory:- Linear search or **sequential search** is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched. Linear search runs in at worst linear time and makes at most *n* comparisons, where *n* is the length of the list

Procedure:-

LinearSearch(value, list)

1.if the list is empty, return item_not_found;
2.else
3.if the first item of the list has the desired value, return its location;
4. else return LinearSearch(value, remainder of the list)

Program:

```
#include<stdio.h
> int main(){
   int a[10],i,n,m,c=0;
   printf("Enter the size of an array: ");
   scanf("%d",&n);
   printf("Enter the elements of the array: ");
   for(i=0;i<=n-1;i++){
      scanf("%d",&a[i]);
   }
   printf("Enter the number to be search: ");
   scanf("%d",&m);
   for(i=0;i<=n-1;i++){
      if(a[i]==m){
         c=1;
         break;
      }
   }
   if(c==0)
      printf("The number is not in the list");
   else
```

```
        printf("The number is found");
    return 0;

}
```

PROGRAM 2

OBJECTIVE: - Program for Recursive Binary Search.

Brief Theory:-A **binary search** will start by examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space.

Procedure:-Given an array A of n elements with values or records $A0$ ... $An-1$ and target value T, the following subroutine uses binary search to find the index of T in A.

1. Set L to 0 and R to $n - 1$.
2. If $L > R$, the search terminates as unsuccessful. Set m (the position of the middle element) to the floor of $(L + R)/2$.
3. If A$m < T$, set L to $m + 1$ and go to step 2.
4. If A$m >$ T, set R to $m - 1$ and go to step 2.
5. If A$m = T$, the search is done; return m.

Program:-

```
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int key) {

   if (right >= left) {


      int mid = left + (right - left) / 2;


      if (arr[mid] == key)
         return mid;


      if (arr[mid] > key)  {
         return binarySearch(arr, left, mid - 1, key);
      }

      return binarySearch(arr, mid + 1, right, key);
   }
```

```
    return -1;
}

int main(void) {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int size = sizeof(arr) / sizeof(arr[0]);


    int key = 23;


    int index = binarySearch(arr, 0, size - 1, key);
    if (index == -1) {
        printf("Element is not present in array");
    }
    else {
        printf("Element is present at index %d", index);
    }

    return 0;
}
```

PROGRAM 3

Objective: - Program for Insertion Sort.

Brief Theory:- Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

Efficient for (quite) small data sets, much like other quadratic sorting algorithms
More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort
Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(nk)$ when each element in the input is no more than k places away from its sorted position
Stable; i.e., does not change the relative order of elements with equal keys

Procedure:-

1. for i ← 1 to length(A)-1
2. j ← i
3. while j > 0 and A[j-1] > A[j]
4. swap A[j] and A[j-1] 5.j ← j – 1
6. end while
7. end for

Program:

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


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

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

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


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

// Driver code
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

PROGRAM 4

Objective: - Program for Selection Sort.

Brief Theory:- Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n^2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Procedure:-

SELECTION_SORT (A)

```
for i ← 1 to n-1 do
   min j ← i;
   min x ← A[i]
   for j ← i + 1 to n do
      If A[j] < min x
      then
         min j ← j
         min x ← A[j]
   A[min j] ← A [i]
   A[i] ← min x
```

Program:

```
#include
<stdio.h>

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

```
*xp;
        *xp = *yp;
        *yp =
temp;
    }

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


        for (i = 0; i
< n-1; i++)
        {

            min_idx
= i;
            for (j =
i+1; j < n; j++)
              if
(arr[j] <
arr[min_idx])

min_idx = j;



if(min_idx != i)

swap(&arr[min_i
dx], &arr[i]);
        }
    }

    void
printArray(int
arr[], int size)
    {
        int i;
```

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

printf("%d ",
arr[i]);

printf("\n");
    }


    int main()
    {
        int arr[] =
{64, 25, 12, 22,
11};
        int n =
sizeof(arr)/sizeof(
arr[0]);

selectionSort(arr,
n);

printf("Sorted
array: \n");

printArray(arr, n);
        return 0;
    }
```

PROGRAM 5

Objective: - Program for Merge Sort.

Brief Theory:- **M**erge sort is based on the **divide-and-conquer** paradigm. Its worst- case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a sub array $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurs through sub problems. To sort $A[p .. r]$:

1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two subarrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2. Conquer Step

Conquer by recursively sorting the two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. Combine Step

Combine the elements back in $A[p .. r]$ by merging the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Procedure:-

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, r)

```
1.  IF p < r                          // Check for base case
2.     THEN q = FLOOR[(p + r)/2]        // Divide step
3.          MERGE (A, p, q)           // Conquer step.
4.          MERGE (A, q + 1, r)       // Conquer step.
5.          MERGE (A, p, q, r)        // Conquer step.
```

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

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


 int L[n1], R[n2];


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


i = 0;
j = 0;
k = l;
 while (i < n1 && j < n2) {
 if (L[i] <= R[j]) {
 arr[k] = L[i];
    i++;
}
 else {
       arr[k] = R[j];
     j++;
    }
      k++;
       }


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


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

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


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

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


    int main()
  {
     int arr[] = { 12, 11, 13, 5, 6, 7 };
     int arr_size = sizeof(arr) / sizeof(arr[0]);

      printf("Given array is \n");
        printArray(arr, arr_size);

         mergeSort(arr, 0, arr_size - 1);

       printf("\nSorted array is \n");
         printArray(arr, arr_size);
    return 0;
      }
```

PROGRAM 6

Objective: - Program for Quick Sort.

Brief Theory:- Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition*operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Procedure:-

```
quicksort(A, lo, hi)
if lo < hi then
 p := partition(A, lo, hi)
 quicksort(A, lo, p – 1)
quicksort(A, p + 1,
hi) partition(A, lo, hi)
pivot := A[hi]
i := lo        // place for swapping
for j := lo to hi – 1 do
if A[j] ≤ pivot then
swap A[i] with A[j]
i:= i + 1
swap A[i] with A[hi]
return i
```

Program:-

```
#include <stdio.h>

void swap(int* a, int* b)
```

```
    {
    int t = *a;
    *a = *b;
    *b = t;
}


int partition(int arr[], int low, int high)
{

    int pivot = arr[high];



    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {


        if (arr[j] < pivot) {


            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}


void quickSort(int arr[], int low, int high)
{
    if (low < high) {


        int pi = partition(arr, low, high);


        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int N = sizeof(arr) / sizeof(arr[0]);


    quickSort(arr, 0, N - 1);
    printf("Sorted array: \n");
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
```

```
    return 0;
}
```

PROGRAM 7

Objective: Program for heap sort

Brief Theory:- Heapsort is a comparison-based sorting algorithm. Heap sort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts:

Creating a Heap of the unsorted list.
Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Procedure:-

The steps are:

1. Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in O(n) operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the Adjust() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

Program:-

```
#include <stdio.h>
void swap(int* a, int* b)
{

    int temp = *a;
    *a = *b;
    *b = temp;
}
void heapify(int arr[], int N, int i)
{
```

```
    if (left < N && arr[left] > arr[largest])

        largest = left;

    if (right < N && arr[right] > arr[largest])

        largest = right;

    if (largest != i) {

        swap(&arr[i], &arr[largest]);

        heapify(arr, N, largest);
    }
}

void heapSort(int arr[], int N)
{

    for (int i = N / 2 - 1; i >= 0; i--)

        heapify(arr, N, i);
    for (int i = N - 1; i >= 0; i--) {

        swap(&arr[0], &arr[i]);

        heapify(arr, i, 0);
    }
}

void printArray(int arr[], int N)
{
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, N);
    printf("Sorted array is\n");
    printArray(arr, N);
}
```

PROGRAM 8

Objective : Program for shell sort

Shell sort, also known as Shell sort or Shell's method, is an in-place comparison sort. It can either be seen as a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort). The method starts by sorting elements far apart from each other and progressively reducing the gap between them. Starting with far apart elements can move some out-of-place elements into position faster than a simple nearest neighbor exchange. Here is the source code of the C program to sort integers using Shell Sort technique. The C program is successfully compiled and run on a Linux system. The program output is also shown below.

Program:

```
#include <stdio.h>
void shellSort(int array[], int n) {
  // Rearrange elements at each n/2, n/4, n/8, ... intervals
  for (int interval = n / 2; interval > 0; interval /= 2) {
    for (int i = interval; i < n; i += 1) {
      int temp = array[i];
      int j;
      for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
        array[j] = array[j - interval];
      }
      array[j] = temp;
    }
  }
}
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

int main() {
  int data[] = {9, 8, 3, 7, 5, 6, 4, 1};
  int size = sizeof(data) / sizeof(data[0]);
  shellSort(data, size);
  printf("Sorted array: \n");
```

```
  printArray(data, size);
}
```

PROGRAM 9

Objective: - Program to find the minimum spanning tree using Kruskal's algorithm.

Brief Theory:- Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. 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).

Procedure:-

```
KRUSKAL(G):
 A = ∅
   foreach v ∈ G.V:
     MAKE-SET(v)
   foreach (u, v) in G.E ordered by weight(u, v), increasing:
     if FIND-SET(u) ≠ FIND-SET(v):
        A = A ∪ {(u, v)}
        UNION(u,
v) return A
```

Program:-

```
#include <stdio.h>

#include <stdlib.h>

// Comparator function to use in sorting

int comparator(const void* p1, const void* p2)

{

   const int(*x)[3] = p1;

   const int(*y)[3] = p2;

   return (*x)[2] - (*y)[2];
```

```
}
 // Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}
  // Function to find the parent of a node
int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;

    return parent[component]
           = findParent(parent, parent[component]);
}
 // Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    // Finding the parents
    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;
  // Since the rank increases if
 // the ranks of two sets are same
        rank[u]++;
    }
}
  Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
 First we sort the edge array in ascending order
  so that we can access minimum distances/cost
    qsort(edge, n, sizeof(edge[0]), comparator);
   int parent[n];
    int rank[n];
  // Function to initialize parent[] and rank[]
    makeSet(parent, rank, n);
   // To store the minimun cost
    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 the parents are different that
   // means they are in different sets so
   // union them
```

```
        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);
}
  Driver code
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;
}
```

PROGRAM 10

To find minimum cost of spanning tree using Prims Algorithm

Description:

Let G(V,E) be an undirected graph.
A sub-graph T=(V,E^1) is said to be a spanning tree of G if T is a tree.
A Minimum cost spanning tree is a spanning tree with minimum weight.

Prim's method:

The Greedy method to obtain MCST builds the tree edge by edge.
Choose the edges with minimum cost
Cycles should be avoided

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

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
   // Initialize min value
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;

   return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d \n", parent[i], i,
```

```
            graph[i][parent[i]]);
}
```

Program:

```
// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];
    // Key values used to pick minimum weight edge in cut
    int key[V];
    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;

    // First node is always root of MST
    parent[0] = -1;

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {

        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false
                && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
```

```
    printMST(parent, graph);
}

// Driver's code
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 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

PROGRAM 11

Objective: - Program to implement N Queen Problem using Backtracking.

Brief Theory:-In this problem or puzzle, N queens will be given and the challenge is to place all of them in N x N chessboard so that no two queens are in attacking position. What that means is we need to find the configuration of N queens where any of the two queens do not share a row, column and diagonal paths. One of the better and elegant solutions to this puzzle is 'backtracking algorithm'.

Procedure:-

1. Place the queens column wise, start from the left most column
2. If all queens are placed.
    1. return true and print the solution matrix.
3. Else
    1. Try all the rows in the current column.
    2. Check if queen can be placed here safely if yes mark the current cell in solution matrix as 1 and try to solve the rest of the problem recursively.
    3. If placing the queen in above step leads to the solution return true.
    4. If placing the queen in above step does not lead to the solution , BACKTRACK, mark the current cell in solution matrix as 0 and return false.
4. If all the rows are tried and nothing worked, return false and print NO SOLUTION.

Program:-

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
char a[10][10];
int n;
void printmatrix()
{
int i, j;
printf("\n");
```

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

for (j = 0; j < n; j++)
printf("%c\t", a[i][j]);
printf("\n\n");
}
}
int getmarkedcol(int row)
{
int i;
for (i = 0; i < n; i++)
if (a[row][i] == 'Q')
{
return (i);
break;
}
}
int feasible(int row, int col)
{
int i, tcol;
for (i = 0; i < n; i++)
{
tcol = getmarkedcol(i);
if (col == tcol || abs(row - i) == abs(col - tcol))
return 0;
}
return 1;
}
void nqueen(int row)
{
int i, j;
if (row < n)
{
for (i = 0; i < n; i++)
{
if (feasible(row, i))
```

```
{
a[row][i] = 'Q';
nqueen(row + 1);
a[row][i] = '.';
}
}
}
else

{
printf("\nThe solution is:- ");
printmatrix();
}
}
void main()
{
clrscr();
int i, j;
printf("\nEnter the no. of queens:- ");
scanf("%d", &n);
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
a[i][j] = '.';
nqueen(0);
getch();
}
```

PROGRAM 12

Objective: - Program to perform knapsack problem using Greedy Approach.

Brief Theory:- The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Procedure:-

1. // Input:
2. // Values (stored in array v)
3. // Weights (stored in array w)
4. // Number of distinct items (n)
5. // Knapsack capacity (W)
6. for j from 0 to W do:
7. m[0, j] := 0
8. for i from 1 to n do:
9. for j from 0 to W do:
10. if w[i-1] > j then:
11. m[i, j] := m[i-1, j]
12. else:
13. m[i, j] := max(m[i-1, j], m[i-1, j-w[i-1]] + v[i-1])

Program:-

```
#include<stdio.h>
#include<conio.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);
}
void main()
{
clrscr();
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 weights and profits of each object:- ");
for (i = 0; i < num; i++)
{
scanf("%f %f", &weight[i], &profit[i]);
}
printf("\nEnter the capacity 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);
getch();
}
```

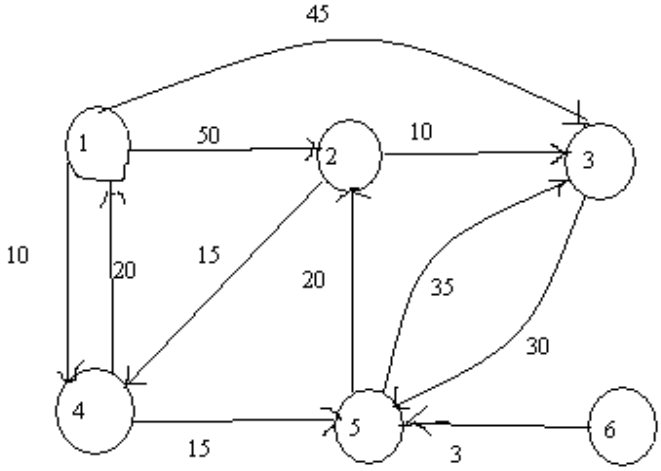
PROGRAM 13

Objective: To write a program to implement Dijkstra's algorithm

Description:

Let G (V,E) be a graph , V-Vertices & E-Edges .We have to choose one vertex as source , the problem aim is to find out minimum distance between source node and all remaining nodes.

This problem is the case of ordered paradigm in Greedy method.



This problem can be implemented by an algorithm called Dijkastra's algorithm.

PATH	COST
1,4	10
1,4,5	25
1,4,5,2	45
1,3	45

Algorithm:

/

```
//G be a graph
//Cost matrix [1:n,1:n] for the graph G
//S={set of vertices that path already generated}
//Let V be source vertex
//dist[j];1<=j<=n denotes distance between V and j
void main()
{
for i:=1 to n do
        {
                s[i]=false; // initialize s with n
                dist[i]=cost[v,i]; //define distance
        }
        s[v]=true; //put v in s
        dist[v]=0.0; //Distance between v and v is 0
        for num:=2 to n-1 do
        {
paths from v //choose u from among those vertices not in S such that dist[u]=min;
s[u]=true;
for(each w adjascent to u with s[w]=false)
        if(dist[w]>dist[u]+cost[u,w])
        then
                dist[w]=dist[u]+cost[u,w]; //update the distance
        }
}
```

Source Code:

```
#include<stdio.h>
#include<conio.h>
#define infinity 32767

int cost[20][20],n,dist[20],s[20],a[20][20];
void setdata();
void getdata();
void path(int);

void setdata()
{
        int i,j,k;
        printf("\nEnter number of nodes: ");
        scanf("%d",&n);
        printf("Enter Adjacency Matrix: ");
        for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                        scanf("%d",&a[i][j]);
        for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                {
                        if(i==j)



                                cost[i][i]=0;
```

```
                else if (a[i][j]!=0)
                {
                        printf("\nEnter cost from %d to %d: ",i,j);
                        scanf("%d",&cost[i][j]);
                }
                else
                        cost[i][j]=infinity;
        }
}

void getdata()
{
        int i;
        for(i=1;i<=n;i++)
                if(dist[i]==32767)
                        printf("not reachable");
                else
                        printf(" %d",dist[i]);
}

void path(int v)
{
        int i,j,min,u;
        for(i=1;i<=n;i++)
        {
                s[i]=0;
                dist[i]=cost[v][i];
        }
        s[v]=1;
        dist[v]=0;
        for(i=2;i<=n;i++)
        {
                min=32767;
                for(j=1;j<=n;j++)
                        if(s[j]==0 && dist[j]<min)
                                u=j;
                s[u]=1;
                for(j=1;j<=n;j++)
                        if(s[j]==0 && a[u][j]==1)
                                if(dist[j]>dist[u]+cost[u][j])
                                        dist[j]=dist[u]+cost[u][j];
        }
}







void main()
{


        int v;
```

```
        clrscr();
        setdata();
        printf("\nEnter the source vertex: ");
        scanf("%d",&v);
        path(v);
        printf("\nShortest paths " );
        getdata();
        getch();
}
```

Output:

```
Enter number of nodes: 6
Enter Adjacency Matrix: 0 1 1 1 0 0
0 0 1 1 0 0
0 0 0 0 1 0
1 0 0 0 1 0
0 1 1 0 0 0
0 0 0 0 1 0
Enter cost from 1 to 2:  50

Enter cost from 1 to 3:  45

Enter cost from 1 to 4:  10

Enter cost from 2 to 3:  10

Enter cost from 2 to 4:  15

Enter cost from 3 to 5:  30

Enter cost from 4 to 1:  20

Enter cost from 4 to 5:  15

Enter cost from 5 to 2:  20

Enter cost from 5 to 3:  35

Enter cost from 6 to 5:  3

Enter the source vertex:  1

Shortest paths 0 45 45 10 25not reachable
```

PROGRAM 14

Objective: - Program to perform Travelling Salesman Problem.

Brief Theory:-The Travelling Salesman Problem describes a salesman who must travel between N cities. The order in which he does so is something he does not care about, as long as he visits each one during his trip, and finishes where he was at first. Each city is connected to other close by cities, or nodes, by airplanes, or by road or railway. Each of those links between the cities has one or more weights (or the cost) attached. The cost describes how "difficult" it is to traverse this edge on the graph, and may be given, for example, by the cost of an airplane ticket or train ticket, or perhaps by the length of the edge, or time required to complete the traversal. The salesman wants to keep both the travel costs, as well as the distance he travels as low as possible.

Procedure:-

1. stand on an arbitrary vertex as current vertex.
2. find out the lightest edge connecting current vertex and an unvisited vertex V.
3. set current vertex to V.
4. mark V as visited.
5. if all the vertices in domain are visited, then terminate.
6. Go to step 2.

Program:-

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int a[10][10],visited[10],n,cost=0;
void get()
{
int i,j;
printf("\n\nEnter Number of Cities: ");
scanf("%d",&n);
printf("\nEnter Cost Matrix:
\n"); for( i=0;i<n;i++)
```

```
{
printf("\n Enter Elements of Row # :
%d\n",i+1); for( j=0;j<n;j++)

scanf("%d",&a[i][j]);
visited[i]=0;
}
printf("\n\nThe Cost Matrix is:\n");
for( i=0;i<n;i++)
{ printf("\n\n");
for(j=0;j<n;j++)
printf("\t%d",a[i][j]);
}
}
void mincost(int city)
{
int i,ncity,least(int city);
visited[city]=1;
printf("%d ===> ",city+1);
ncity=least(city);
if(ncity==999)
{ ncity=0;
printf("%d",ncity+1)
;
cost+=a[city][ncity];
return;
}
mincost(ncity);
}
int least(int c)
{
int i,nc=999;
int min=999,kmin;
for(i=0;i<n;i++)
{
if((a[c][i]!=0)&&(visited[i]==0))
if(a[c][i]<min)
{
```

```
min=a[i][0]+a[c][i];
kmin=a[c][i];
nc=i;
}
}
if(min!=999)
cost+=kmin;

return nc;
}
void put()
{
printf("\n\nMinimum cost:");
printf("%d",cost);
}
void main()
{
clrscr();
get();
printf("\n\nThe Path is:\n\n");
mincost(0);
put();
getch();
}
```