**1 EXP->BINARY SEARCH(ITERATIVE METHOD)**

#include<stdio.h>

int main()

{

int i,a,b[20],search,first,last,middle;

printf("enter no of elements to be entered:");

scanf("%d",&a);

printf("enter elements in ascending order:\n");

for(i=0;i<a;i++)

{

scanf("%d",&b[i]);

}

printf("enter the element to be searched:\n");

scanf("%d",&search);

first=0;

last=a-1;

middle=(first+last)/2;

while(first<last)

{

if(b[middle]<search)

{

first=middle+1;

}

else if (b[middle]==search)

{

printf("the element %d is at positon %d",search,middle+1);

break;

}

else

{

last=middle-1;

}

}

middle=(first+last)/2;

if(first>last)

{

printf("element not found in array.");

}

getch();

return 0;

}

**2->RECURSIVE METHOD**

#include <stdio.h>

int binarySearch(int arr[], int l, int r, int x)

{

if (r >= l) {

int mid = l + (r - l) / 2;

if (arr[mid] == x)

return mid;

if (arr[mid] > x) {

return binarySearch(arr, l, mid - 1, x);

}

return binarySearch(arr, mid + 1, r, x);

}

return -1;

}

int main(void)

{

int arr[] = { 2, 3, 4, 10, 40 };

int size = sizeof(arr) / sizeof(arr[0]);

int x = 10;

int index = binarySearch(arr, 0, size - 1, x);

if (index == -1) {

printf("Element is not present in array");

}

else {

printf("Element is present at index %d", index);

}

getch();

return 0;

}

**3->KNAPSACK**

#include<stdio.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

int knapsack(int W, int wt[], int val[], int n) {

int i, w;

int 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 - 1] <= w)

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

else

K[i][w] = K[i - 1][w];

}

}

return K[n][W];

}

int main() {

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

int n = sizeof(val)/sizeof(val[0]);

printf("Maximum value that can be obtained: %d\n", knapsack(W, wt, val, n));

getch();

return 0;

}

**4->DIJKSTRA ALGO**

#include <stdio.h>

#define INFINITY 9999

#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {

int cost[MAX][MAX], distance[MAX], pred[MAX];

int visited[MAX], count, mindistance, nextnode, i, j;

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

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

if (Graph[i][j] == 0)

cost[i][j] = INFINITY;

else

cost[i][j] = Graph[i][j];

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

distance[i] = cost[start][i];

pred[i] = start;

visited[i] = 0;

}

distance[start] = 0;

visited[start] = 1;

count = 1;

while (count < n - 1) {

mindistance = INFINITY;

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

if (distance[i] < mindistance && !visited[i]) {

mindistance = distance[i];

nextnode = i;

}

visited[nextnode] = 1;

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

if (!visited[i])

if (mindistance + cost[nextnode][i] < distance[i]) {

distance[i] = mindistance + cost[nextnode][i];

pred[i] = nextnode;

}

count++;

}

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

if (i != start) {

printf("\nDistance from source to %d: %d", i, distance[i]);

}

}

int main() {

int Graph[MAX][MAX], i, j, n, u;

n = 7;

Graph[0][0] = 0;

Graph[0][1] = 0;

Graph[0][2] = 1;

Graph[0][3] = 2;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 0;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 3;

Graph[1][6] = 0;

Graph[2][0] = 1;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 1;

Graph[2][4] = 3;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 2;

Graph[3][1] = 0;

Graph[3][2] = 1;

Graph[3][3] = 0;

Graph[3][4] = 0;

Graph[3][5] = 0;

Graph[3][6] = 1;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 3;

Graph[4][3] = 0;

Graph[4][4] = 0;

Graph[4][5] = 2;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 3;

Graph[5][2] = 0;

Graph[5][3] = 0;

Graph[5][4] = 2;

Graph[5][5] = 0;

Graph[5][6] = 1;

Graph[6][0] = 0;

Graph[6][1] = 0;

Graph[6][2] = 0;

Graph[6][3] = 1;

Graph[6][4] = 0;

Graph[6][5] = 1;

Graph[6][6] = 0;

u = 0;

Dijkstra(Graph, n, u);

getch();

return 0;

}

**5->KRUSHKAL**

#include <stdio.h>

#define MAX 30

typedef struct edge {

int u, v, w;

} edge;

typedef struct edge\_list {

edge data[MAX];

int n;

} edge\_list;

edge\_list elist;

int Graph[MAX][MAX], n;

edge\_list spanlist;

void kruskalAlgo();

int find(int belongs[], int vertexno);

void applyUnion(int belongs[], int c1, int c2);

void sort();

void print();

void kruskalAlgo() {

int belongs[MAX], i, j, cno1, cno2;

elist.n = 0;

for (i = 1; i < n; i++)

for (j = 0; j < i; j++) {

if (Graph[i][j] != 0) {

elist.data[elist.n].u = i;

elist.data[elist.n].v = j;

elist.data[elist.n].w = Graph[i][j];

elist.n++;

}

}

sort();

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

belongs[i] = i;

spanlist.n = 0;

for (i = 0; i < elist.n; i++) {

cno1 = find(belongs, elist.data[i].u);

cno2 = find(belongs, elist.data[i].v);

if (cno1 != cno2) {

spanlist.data[spanlist.n] = elist.data[i];

spanlist.n = spanlist.n + 1;

applyUnion(belongs, cno1, cno2);

}

}

}

int find(int belongs[], int vertexno) {

return (belongs[vertexno]);

}

void applyUnion(int belongs[], int c1, int c2) {

int i;

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

if (belongs[i] == c2)

belongs[i] = c1;

}

void sort() {

int i, j;

edge temp;

for (i = 1; i < elist.n; i++)

for (j = 0; j < elist.n - 1; j++)

if (elist.data[j].w > elist.data[j + 1].w) {

temp = elist.data[j];

elist.data[j] = elist.data[j + 1];

elist.data[j + 1] = temp;

}

}

void print() {

int i, cost = 0;

for (i = 0; i < spanlist.n; i++) {

printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);

cost = cost + spanlist.data[i].w;

}

printf("\nSpanning tree cost: %d", cost);

}

int main() {

int i, j, total\_cost;

n = 6;

Graph[0][0] = 0;

Graph[0][1] = 4;

Graph[0][2] = 4;

Graph[0][3] = 0;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 4;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 0;

Graph[1][6] = 0;

Graph[2][0] = 4;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 3;

Graph[2][4] = 4;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 0;

Graph[3][1] = 0;

Graph[3][2] = 3;

Graph[3][3] = 0;

Graph[3][4] = 3;

Graph[3][5] = 0;

Graph[3][6] = 0;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 4;

Graph[4][3] = 3;

Graph[4][4] = 0;

Graph[4][5] = 0;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 0;

Graph[5][2] = 2;

Graph[5][3] = 0;

Graph[5][4] = 3;

Graph[5][5] = 0;

Graph[5][6] = 0;

kruskalAlgo();

print();

getch();

return 0;

}

**6->PRIMS**

#include<stdio.h>

#include<stdlib.h>

#define infinity 9999

#define MAX 20

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

int prims();

int main()

{

int i,j,total\_cost;

printf("Enter no. of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

scanf("%d",&G[i][j]);

total\_cost=prims();

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

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

{

printf("\n");

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

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

}

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

getch();

return 0;

}

int prims()

{

int cost[MAX][MAX];

int u,v,min\_distance,distance[MAX],from[MAX];

int visited[MAX],no\_of\_edges,i,min\_cost,j;

//create cost[][] matrix,spanning[][]

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

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

{

if(G[i][j]==0)

cost[i][j]=infinity;

else

cost[i][j]=G[i][j];

spanning[i][j]=0;

}

distance[0]=0;

visited[0]=1;

for(i=1;i<n;i++)

{

distance[i]=cost[0][i];

from[i]=0;

visited[i]=0;

}

min\_cost=0; //cost of spanning tree

no\_of\_edges=n-1; //no. of edges to be added

while(no\_of\_edges>0)

{

min\_distance=infinity;

for(i=1;i<n;i++)

if(visited[i]==0&&distance[i]<min\_distance)

{

v=i;

min\_distance=distance[i];

}

u=from[v];

spanning[u][v]=distance[v];

spanning[v][u]=distance[v];

no\_of\_edges--;

visited[v]=1;

for(i=1;i<n;i++)

if(visited[i]==0&&cost[i][v]<distance[i])

{

distance[i]=cost[i][v];

from[i]=v;

}

min\_cost=min\_cost+cost[u][v];

}

return(min\_cost);

}

**7->RANDOMIZED QUICK SORT**

#include <stdio.h>

void swap(int \*a, int \*b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

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

int pivot = array[high];

int i = (low - 1);

int j;

for ( j = low; j < high; j++) {

if (array[j] <= pivot) {

i++;

swap(&array[i], &array[j]);

}

}

swap(&array[i + 1], &array[high]);

return (i + 1);

}

void quickSort(int array[], int low, int high) {

if (low < high) {

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

quickSort(array, low, pi - 1);

quickSort(array, pi + 1, high);

}

}

void printArray(int array[], int size) {

int i;

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

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

}

printf("\n");

}

int main() {

int data[] = {8, 7, 2, 1, 0, 9, 6};

int n = sizeof(data) / sizeof(data[0]);

printf("Unsorted Array\n");

printArray(data, n);

// perform quicksort on data

quickSort(data, 0, n - 1);

printf("Sorted array in ascending order: \n");

printArray(data, n);

getch();

return 0;

}