**EXP NO 1 :- IMPLEMENT INSERTION SORT**

CODE:-

#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;

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");

}

int main()

{

int arr[] = {12, 11, 13, 5, 6};

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

insertionSort(arr, n);

printArray(arr, n);

return 0;

}

OUTPUT:-



**EXP NO 2 :- IMPLEMENT SELECTION SORT**

CODE:-

#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\_idx], &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;

}

OUTPUT:-

\

**EXP NO 3:- IMPLEMENT QUICK SORT**

Input:-

#include <stdio.h>

void swap(int \*a, int \*b)https://learn.onlinegdb.com/ {

int t = \*a; +

\*a = \*b;

\*b = t;

}

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

int pivot = array[high];

int i = (low - 1);

for (int 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) {

for (int 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);

quickSort(data, 0, n - 1);

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

printArray(data, n);

}

Output:-



**EXP NO 4:- IMPLEMENT BINARY SORT**

CODE:-

#include <stdio.h>

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

{

while (l <= r) {

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

if (arr[m] == x)

return m;

if (arr[m] < x)

l = m + 1;

else

r = m - 1;

}

return -1;

}

int main(void)

{

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

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

int x = 10;

int result = binarySearch(arr, 0, n - 1, x);

(result == -1) ? printf("Element is not present"

" in array")

: printf("Element is present at "

"index %d",

result);

return 0;

}

OUTPUT:-



**EXP NO 5:- IMPLEMENT KNAPSACK ALGORITHM**

INPUT:-

#include<stdio.h>

int max(int a, int b) {

if(a>b){

return a;

} else {

return b;

}

}

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

int i, w;

int knap[n+1][W+1];

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

for (w = 0; w <= W; w++) {

if (i==0 || w==0)

knap[i][w] = 0;

else if (wt[i-1] <= w)

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

else

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

}

}

return knap[n][W];

}

int main() {

int val[] = {20, 25, 40};

int wt[] = {25, 20, 30};

int W = 50;

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

printf("The solution is : %d", knapsack(W, wt, val, n));

return 0;

}

OUTPUT:-



**EXP NO 6:- IMPLEMENT PRIMS ALGORITHM**

INPUT:-

#include <stdio.h>

int max(int a, int b) { return (a > b) ? a : b; }

int knapSack(int W, int wt[], int val[], int n)

{

if (n == 0 || W == 0)

return 0;

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

else

return max(

val[n - 1]

+ knapSack(W - wt[n - 1], wt, val, n - 1),

knapSack(W, wt, val, n - 1));

}

int main()

{

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

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

int W = 50;

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

printf("%d", knapSack(W, weight, profit, n));

return 0;

}

OUTPUT:-



**EXP NO 7:- IMPLEMENT KRUSKALS ALGORITHM**

INPUT:-

#include <stdio.h>

#include <stdlib.h>

#define MAX\_EDGES 1000

typedef struct Edge {

int src, dest, weight;

} Edge;

typedef struct Graph {

int V, E;

Edge edges[MAX\_EDGES];

} Graph;

typedef struct Subset {

int parent, rank;

} Subset;

Graph\* createGraph(int V, int E) {

Graph\* graph = (Graph\*) malloc(sizeof(Graph));

graph->V = V;

graph->E = E;

return graph;

}

int find(Subset subsets[], int i) {

if (subsets[i].parent != i) {

subsets[i].parent = find(subsets, subsets[i].parent);

}

return subsets[i].parent;

}

void Union(Subset subsets[], int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank) {

subsets[xroot].parent = yroot;

} else if (subsets[xroot].rank > subsets[yroot].rank) {

subsets[yroot].parent = xroot;

} else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

int compare(const void\* a, const void\* b) {

Edge\* a\_edge = (Edge\*) a;

Edge\* b\_edge = (Edge\*) b;

return a\_edge->weight - b\_edge->weight;

}

void kruskalMST(Graph\* graph) {

Edge mst[graph->V];

int e = 0, i = 0;

qsort(graph->edges, graph->E, sizeof(Edge), compare);

Subset\* subsets = (Subset\*) malloc(graph->V \* sizeof(Subset));

for (int v = 0; v < graph->V; ++v) {

subsets[v].parent = v;

subsets[v].rank = 0;

}

while (e < graph->V - 1 && i < graph->E) {

Edge next\_edge = graph->edges[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

if (x != y) {

mst[e++] = next\_edge;

Union(subsets, x, y);

}

}

printf("Minimum Spanning Tree:\n");

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

printf("(%d, %d) -> %d\n", mst[i].src, mst[i].dest, mst[i].weight);

}

}

int main() {

int V, E;

printf("Enter number of vertices and edges: ");

scanf("%d %d", &V, &E);

Graph\* graph = createGraph(V, E);

printf("Enter edges and their weights:\n");

for (int i = 0; i < E; ++i) {

scanf("%d %d %d", &graph->edges[i].src, &graph->edges[i].dest, &graph->edges[i].weight);

}

kruskalMST(graph);

return 0;

}

OUTPUT:-



**EXP NO 8:- SINGLE SOURCE SHORTEST PATH ALGORITHM**

INPUT:-

#include <stdio.h>

#include <stdlib.h>

#define INFINITY 99999

//struct for the edges of the graph

struct Edge {

int u; //start vertex of the edge

int v; //end vertex of the edge

int w; //weight of the edge (u,v)

};

//Graph - it consists of edges

struct Graph {

int V; //total number of vertices in the graph

int E; //total number of edges in the graph

struct Edge \*edge; //array of edges

};

void bellmanford(struct Graph \*g, int source);

void display(int arr[], int size);

int main(void) {

//create graph

struct Graph \*g = (struct Graph \*)malloc(sizeof(struct Graph));

g->V = 4; //total vertices

g->E = 5; //total edges

//array of edges for graph

g->edge = (struct Edge \*)malloc(g->E \* sizeof(struct Edge));

//------- adding the edges of the graph

/\*

edge(u, v)

where u = start vertex of the edge (u,v)

v = end vertex of the edge (u,v)

w is the weight of the edge (u,v)

\*/

//edge 0 --> 1

g->edge[0].u = 0;

g->edge[0].v = 1;

g->edge[0].w = 5;

//edge 0 --> 2

g->edge[1].u = 0;

g->edge[1].v = 2;

g->edge[1].w = 4;

//edge 1 --> 3

g->edge[2].u = 1;

g->edge[2].v = 3;

g->edge[2].w = 3;

//edge 2 --> 1

g->edge[3].u = 2;

g->edge[3].v = 1;

g->edge[3].w = 6;

//edge 3 --> 2

g->edge[4].u = 3;

g->edge[4].v = 2;

g->edge[4].w = 2;

bellmanford(g, 0); //0 is the source vertex

return 0;

}

void bellmanford(struct Graph \*g, int source) {

//variables

int i, j, u, v, w;

//total vertex in the graph g

int tV = g->V;

//total edge in the graph g

int tE = g->E;

//distance array

//size equal to the number of vertices of the graph g

int d[tV];

//predecessor array

//size equal to the number of vertices of the graph g

int p[tV];

//step 1: fill the distance array and predecessor array

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

d[i] = INFINITY;

p[i] = 0;

}

//mark the source vertex

d[source] = 0;

//step 2: relax edges |V| - 1 times

for (i = 1; i <= tV - 1; i++) {

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

//get the edge data

u = g->edge[j].u;

v = g->edge[j].v;

w = g->edge[j].w;

if (d[u] != INFINITY && d[v] > d[u] + w) {

d[v] = d[u] + w;

p[v] = u;

}

}

}

//step 3: detect negative cycle

//if value changes then we have a negative cycle in the graph

//and we cannot find the shortest distances

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

u = g->edge[i].u;

v = g->edge[i].v;

w = g->edge[i].w;

if (d[u] != INFINITY && d[v] > d[u] + w) {

printf("Negative weight cycle detected!\n");

return;

}

}

//No negative weight cycle found!

//print the distance and predecessor array

printf("Distance array: ");

display(d, tV);

printf("Predecessor array: ");

display(p, tV);

}

void display(int arr[], int size) {

int i;

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

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

}

printf("\n");

}

OUTPUT:-



**EXP NO 9:- To implement Travelling Salesperson Problem using Dynamic Approach.**

CODE:-

#include <stdio.h>

#include <limits.h>

#define MAX 9999

int n = 4;

int distan[20][20] = {

{0, 22, 26, 30},

{30, 0, 45, 35},

{25, 45, 0, 60},

{30, 35, 40, 0}};

int DP[32][8];

int TSP(int mark, int position) {

int completed\_visit = (1 << n) - 1;

if (mark == completed\_visit) {

return distan[position][0];

}

if (DP[mark][position] != -1) {

return DP[mark][position];

}

int answer = MAX;

for (int city = 0; city < n; city++) {

if ((mark & (1 << city)) == 0) {

int newAnswer = distan[position][city] + TSP(mark | (1 << city), city);

answer = (answer < newAnswer) ? answer : newAnswer;

}

}

return DP[mark][position] = answer;

}

int main() {

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

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

DP[i][j] = -1;

}

}

printf("Minimum Distance Travelled -> %d\n", TSP(1, 0));

return 0;

}

OUTPUT:-



**EXP NO 10:- To implement Sub of Subset problem using Backtracking method.**

CODE:-

#include <stdio.h>

#define SIZE 7

void displaySubset(int subSet[], int size) {

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

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

}

printf("\n");

}

void subsetSum(int set[], int subSet[], int n, int subSize, int total, int nodeCount ,int sum) {

if( total == sum) {

displaySubset(subSet, subSize);

if (subSize != 0)

subsetSum(set,subSet,n,subSize-2,total-set[nodeCount],nodeCount+1,sum);

return;

}else {

for( int i = nodeCount; i < n; i++ ) {

subSet[subSize] = set[i];

subsetSum(set,subSet,n,subSize+1,total+set[i],i+1,sum);

}

}

}

void findSubset(int set[], int size, int sum) {

int subSet[size];

subsetSum(set, subSet, size, 0, 0, 0, sum);

}

int main() {

int weights[] = {1, 9, 7, 5, 18, 12, 20, 15};

int size = SIZE;

findSubset(weights, size, 35);

return 0;

}

OUTPUT:-



**EXP NO 11:- To implement N queen problem using Branch and Bound Method.**

#include <stdio.h>

#include <stdbool.h>

#define N 8 // Change N to any desired value

int board[N][N];

// Function to check if a queen can be placed at board[row][col]

bool isSafe(int row, int col) {

int i, j;

// Check this row on left side

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

if (board[row][i])

return false;

// Check upper diagonal on left side

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

// Check lower diagonal on left side

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j])

return false;

return true;

}

// Recursive function to solve N-Queens problem

bool solveNQueensUtil(int col) {

// Base case: If all queens are placed then return true

if (col >= N)

return true;

// Consider this column and try placing this queen in all rows one by one

for (int i = 0; i < N; i++) {

// Check if the queen can be placed on board[i][col]

if (isSafe(i, col)) {

// Place this queen in board[i][col]

board[i][col] = 1;

// Recur to place rest of the queens

if (solveNQueensUtil(col + 1))

return true;

// If placing queen in board[i][col] doesn't lead to a solution then backtrack

board[i][col] = 0; // BACKTRACK

}

}

// If the queen can not be placed in any row in this column col, then return false

return false;

}

// Function to solve N-Queens problem

void solveNQueens() {

// Initialize the board to 0

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

board[i][j] = 0;

if (solveNQueensUtil(0) == false) {

printf("Solution does not exist");

return;

}

// Print the solution

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

printf("%d ", board[i][j]);

printf("\n");

}

}

// Driver program to test above function

int main() {

solveNQueens();

return 0;

}

OUTPUT:-



**EXP NO 11:- Implement the Naïve string-matching algorithm and analyze its complexity.**

#include <stdio.h>

#include <string.h>

// Function to perform naive string matching

void naiveStringMatch(char \*text, char \*pattern) {

int n = strlen(text);

int m = strlen(pattern);

// Iterate through the text

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

int j;

// Check for pattern match starting from index i in the text

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

if (text[i + j] != pattern[j])

break; // Move to the next position in the text if characters don't match

}

if (j == m) {

printf("Pattern found at index %d\n", i);

}

}

}

int main() {

char text[] = "AABAACAADAABAAABAA";

char pattern[] = "AABA";

printf("Text: %s\n", text);

printf("Pattern: %s\n", pattern);

printf("Pattern found at the following indices:\n");

naiveStringMatch(text, pattern);

return 0;

}

OUTPUT:-

