**Experiment No: 0**1

**Title: Insertion sort Program**

#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

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;

}

**Experiment No: 02**

**Title: Selection sort 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\_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;

}

**Experiment No 3**

**Title:- Merge sort**

#include <stdio.h>

#include <stdlib.h>

// Merges two subarrays of arr[]

void merge(int arr[], int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

// Temporary arrays

int L[n1], R[n2];

// Copy data to temp arrays L[] and R[]

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

// Merge the temp arrays back into arr[l..r]

i = 0; j = 0; k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j])

arr[k++] = L[i++];

else

arr[k++] = R[j++];

}

// Copy remaining elements of L[]

while (i < n1)

arr[k++] = L[i++];

// Copy remaining elements of R[]

while (j < n2)

arr[k++] = R[j++];

}

// Merge sort function

void mergeSort(int arr[], int l, int r) {

if (l < r) {

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

mergeSort(arr, l, m); // Sort left half

mergeSort(arr, m + 1, r); // Sort right half

merge(arr, l, m, r); // Merge them

}

}

// Function to print an array

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

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

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

printf("\n");

}

// Main function

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;

}

**EXPERIMENT 4**

**Title:- Implementation of Quick sort algorithm**

#include <stdio.h>

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

// Partition function

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

// Choose the pivot

int pivot = arr[high];

// Index of smaller element and indicates

// the right position of pivot found so far

int i = low - 1;

// Traverse arr[low..high] and move all smaller

// elements to the left side. Elements from low to

// i are smaller after every iteration

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

if (arr[j] < pivot) {

i++;

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

} }

// Move pivot after smaller elements and

// return its position

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

return i + 1;

}

// The QuickSort function implementation

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

if (low < high) {

// pi is the partition return index of pivot

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

// Recursion calls for smaller elements

// and greater or equals elements

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

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

int t = \*a;

\*a = \*b;

\*b = t;

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

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

quickSort(arr, 0, n - 1);

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

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

}

return 0;

}

**Experiment NO . 5**

**Title:- Shortest path using Dijkstra Algorithm.**

/\*C program for Dijkstra's single source shortest path algorithm. The program is for adjacency matrix representation of the graph \*/

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 9

/\*A utility function to find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree \*/

int minDistance(int dist[], bool sptSet[])

{

// Initialize min value

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;

}

// A utility function to print the constructed distance array

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]);

}

/\*Function that implements Dijkstra's single source shortest path algorithm for a graph represented using adjacency matrix representation \*/

void dijkstra(int graph[V][V], int src)

{

int dist[V]; // The output array. dist[i] will hold the shortest distance from src to i

bool sptSet[V]; /\* sptSet[i] will be true if vertex i is included in shortest path tree or shortest distance from src to i is finalized \*/

// Initialize all distances as INFINITE and stpSet[] as false

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

dist[i] = INT\_MAX, sptSet[i] = false;

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

/\*Pick the minimum distance vertex from the set of vertices not yet processed. u is always equal to src in the first iteration.\*/

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.

for (int v = 0; v < V; v++)

/\* Update dist[v] only if is not in sptSet, there is an edge from u to v, and total weight of path from src to v through u is smaller than current value of dist[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];

}

// print the constructed distance array

printSolution(dist);

}

// driver's code

int main()

{

/\* Let us create the example graph discussed above \*/

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 } };

// Function call

dijkstra(graph, 0);

return 0;

}

**EXPERIMENT 6**

**Title:- Single source shortest path- Bellman Ford**

#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) {

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

g->V = 4; // Total vertices

g->E = 5; // Total edges

// Array of edges for the graph

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

// Adding the edges of the graph

g->edge[0].u = 0; g->edge[0].v = 1; g->edge[0].w = 5;

g->edge[1].u = 0; g->edge[1].v = 2; g->edge[1].w = 4;

g->edge[2].u = 1; g->edge[2].v = 3; g->edge[2].w = 3;

g->edge[3].u = 2; g->edge[3].v = 1; g->edge[3].w = 6;

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) {

int i, j, u, v, w;

int tV = g->V;

int tE = g->E;

int d[tV]; // Distance array

int p[tV]; // Predecessor array

// Step 1: Initialize distances and predecessors

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

d[i] = INFINITY;

p[i] = -1;

}

d[source] = 0;

// Step 2: Relax all edges |V| - 1 times

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

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

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: Check for negative-weight cycles

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;

}

}

// Output the results

printf("Distance array: ");

display(d, tV);

printf("Predecessor array: ");

display(p, tV);

}

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

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

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

printf("\n");

}

**EXPERIMENT 7**

**Title:- All pairs shortest path using Floyd Warshall Algorithm.**

#include <stdio.h>

#define V 4

/\* Define Infinite as a large enough value. This value will be used for vertices not connected to each other \*/

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

void floydWarshall(int dist[][V])

{

int i, j, k;

for (k = 0; k < V; k++) {

// Pick all vertices as source one by one

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

// Pick all vertices as destination for the above picked source

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

// If vertex k is on the shortest path from

// i to j, then update the value of

// dist[i][j]

if (dist[i][k] + dist[k][j] < dist[i][j])

dist[i][j] = dist[i][k] + dist[k][j];

} } }

// Print the shortest distance matrix

printSolution(dist);

} /\* A utility function to print solution \*/

void printSolution(int dist[][V])

{

printf( "The following matrix shows the shortest distances" " between every pair of vertices \n");

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

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

if (dist[i][j] == INF)

printf("%7s", "INF");

else

printf("%7d", dist[i][j]);

} printf("\n");

} }

// driver's code

int main()

{ /\* Let us create the following weighted graph

10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3 \*/

int graph[V][V] = { { 0, 5, INF, 10 },

{ INF, 0, 3, INF },

{ INF, INF, 0, 1 },

{ INF, INF, INF, 0 } };

// Function call

floydWarshall(graph);

return 0;

}

**EXPERIEMNT 8**

**Titile:- Travelling salesman problem using dynamic programming**

#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;

}

**EXPERIMENT 9**

**Title:- sum of subsets using backtracking.**

#include <stdio.h>

#include <stdlib.h>

static int total\_nodes;

void printValues(int A[], int size){

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

printf("%\*d", 5, A[i]);

}

printf("\n");

}

void subset\_sum(int s[], int t[], int s\_size, int t\_size, int sum, int ite, int const target\_sum){

total\_nodes++;

if (target\_sum == sum) {

printValues(t, t\_size);

subset\_sum(s, t, s\_size, t\_size - 1, sum - s[ite], ite + 1, target\_sum);

return;

}

else {

for (int i = ite; i < s\_size; i++) {

t[t\_size] = s[i];

subset\_sum(s, t, s\_size, t\_size + 1, sum + s[i], i + 1, target\_sum);

}

}

}

void generateSubsets(int s[], int size, int target\_sum){

int\* tuplet\_vector = (int\*)malloc(size \* sizeof(int));

subset\_sum(s, tuplet\_vector, size, 0, 0, 0, target\_sum);

free(tuplet\_vector);

}

int main(){

int set[] = { 5, 6, 12 , 54, 2 , 20 , 15 };

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

printf("The set is ");

printValues(set , size);

generateSubsets(set, size, 25);

printf("Total Nodes generated %d\n", total\_nodes);

return 0;

}

**EXPERIMENT 10**

**Title:-  The Naïve string-matching Algorithms.**

#include <stdio.h>

#include <string.h>

void search(char\* pat, char\* txt) {

int M = strlen(pat);

int N = strlen(txt);

// A loop to slide pat[] one by one

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

int j;

// For current index i, check for pattern match

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

if (txt[i + j] != pat[j]) {

break; } }

// If pattern matches at index i

if (j == M) { printf("Pattern found at index %d\n", i); } }}

int main() {

// Example 1

char txt1[] = "AABAACAADAABAABA";

char pat1[] = "AABA";

printf("Example 1:\n");

search(pat1, txt1);

// Example 2

char txt2[] = "agd";

char pat2[] = "g";

printf("\nExample 2:\n");

search(pat2, txt2);

return 0; }