Exp1. implement insertion sort

#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;

}

}

{

int i;

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

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

printf("\n");

}

int main()

{

int arr[] = { 23, 69, 52, 74, 21 };

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

insertionSort(arr, n);

printArray(arr, n);

return 0;

}

Output:-

21

23

52

69

74

Exp2: selection sort

#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;

// One by one move boundary of unsorted subarray

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

{

// Find the minimum element in unsorted array

min\_idx = i;

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

if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first element

if(min\_idx != i)

swap(&arr[min\_idx], &arr[i]);

}

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

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

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

printf("\n");

}

// Driver program to test above functions

int main()

{

int arr[] = {34,23,45,16,13};

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

selectionSort(arr, n);

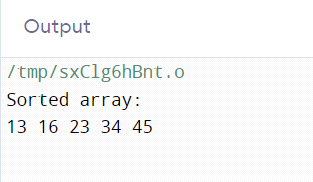
printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output:



Exp4: Binary Search in C

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {

// Repeat until the pointers low and high meet each other

while (low <= high) {

int mid = low + (high - low) / 2;

if (array[mid] == x)

return mid;

if (array[mid] < x)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main(void) {

int array[] = {3, 4, 5, 6, 7, 8, 9};

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

int x = 4;

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

if (result == -1)

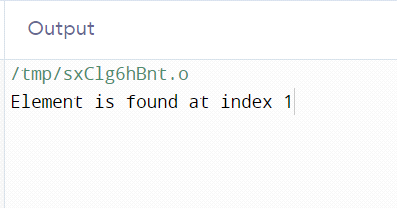
printf("Not found");

else

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

return 0;

}



Exp 3:quick sort

#include <stdio.h>

#include<conio.h>

// Function to swap two elements

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to partition the array and return the pivot index

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

int pivot = arr[high]; // Choose the last element as the pivot

int i = (low - 1); // Index of smaller element

int j;

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

// If current element is smaller than or equal to pivot

if (arr[j] <= pivot) {

i++; // Increment index of smaller element

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

}

}

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

return (i + 1); // Return the partition index

}

// Function to perform Quick Sort

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

if (low < high) {

// Get the partition index

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

// Recursively sort the left and right subarrays

quickSort(arr, low, partitionIndex - 1);

quickSort(arr, partitionIndex + 1, high);

}

}

// Driver code

int main() {

int i;

int arr[] = {12, 11, 13, 5, 6, 7};

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

printf("Original array: ");

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

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

printf("\n");

quickSort(arr, 0, n - 1);

printf("Sorted array: ");

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

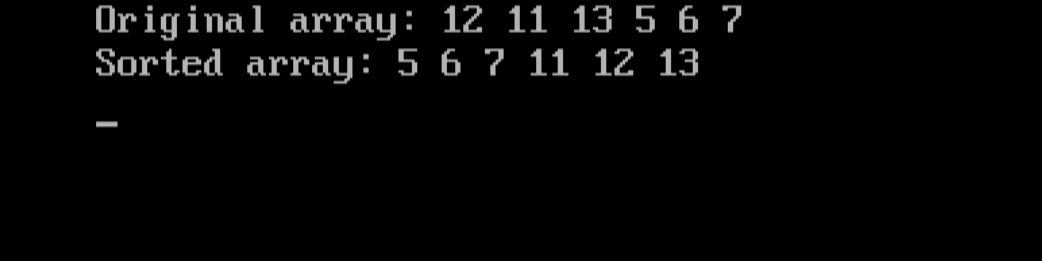
printf("%d ", arr[i]);

printf("\n");

return 0;

}

Output:



Experiment 5: knapsack problems

code:

#include <stdio.h>

void main()

{

int capacity, no\_items, cur\_weight, item;

int used[10];

float total\_profit;

int i;

int weight[10];

int value[10];

printf("Enter the capacity of knapsack:\n");

scanf("%d", &capacity);

printf("Enter the number of items:\n");

scanf("%d", &no\_items);

printf("Enter the weight and value of %d item:\n", no\_items);

for (i = 0; i < no\_items; i++)

{

printf("Weight[%d]:\t", i);

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

printf("Value[%d]:\t", i);

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

}

for (i = 0; i < no\_items; ++i)

used[i] = 0;

cur\_weight = capacity;

while (cur\_weight > 0)

{

item = -1;

for (i = 0; i < no\_items; ++i)

if ((used[i] == 0) &&

((item == -1) || ((float) value[i] / weight[i] > (float) value[item] / weight[item])))

item = i;

used[item] = 1;

cur\_weight -= weight[item];

total\_profit += value[item];

if (cur\_weight >= 0)

printf("Added object %d (%d Rs., %dKg) completely in the bag. Space left: %d.\n", item + 1, value[item], weight[item], cur\_weight);

else

{

int item\_percent = (int) ((1 + (float) cur\_weight / weight[item]) \* 100);

printf("Added %d%% (%d Rs., %dKg) of object %d in the bag.\n", item\_percent, value[item], weight[item], item + 1);

total\_profit -= value[item];

total\_profit += (1 + (float)cur\_weight / weight[item]) \* value[item];

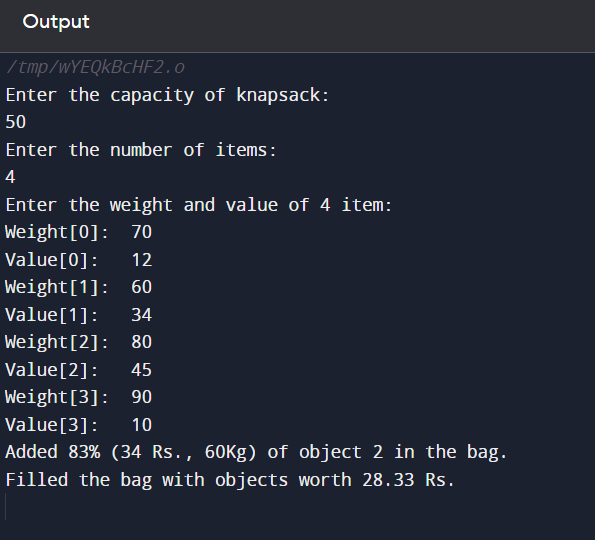
}

}

printf("Filled the bag with objects worth %.2f Rs.\n", total\_profit);

}

output:



Experiment 6: To implement Prim’s MST Algorithm using Greedy Method.

Code:

// A C program for Prim's Minimum

// Spanning Tree (MST) 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 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]]);

}

// 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 } };

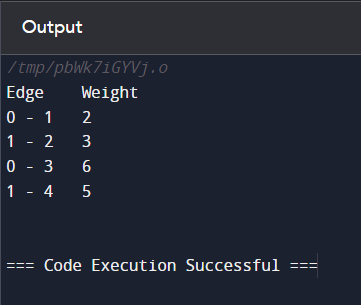
// Print the solution

primMST(graph);

return 0;

}

Output:

`

Experiment 7: To implement Kruskal’s MST Algorithm using Greedy Method.

Code:

// C code to implement Kruskal's algorithm

#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 },

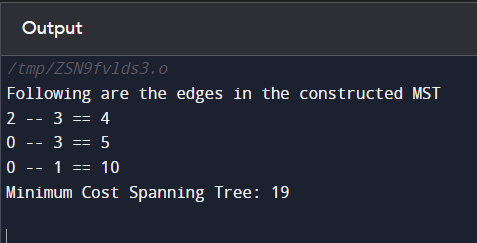
{ 2, 3, 4 } };

kruskalAlgo(5, edge);

return 0;

}

Output:



Experiment 8: To implement Single Source Shortest Path Algorithm using Dynamic (Bellman Ford) Method

Code: // Bellman Ford Algorithm in C

#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");

}

