**NAME ANIMESH KUMAR**

**CLASS BTECH 4th SEMESTER [C.S.E]**

**ROLL NO 07**

**SUBJECT DESIGN AND ANALYSIS OF ALGORITHMS**

**LAB ASSIGNMENT**

**Design and Analysis of Algorithms**

**Lab Assignment**

1. **Write a program that implements Bubble sort.**

#include <stdio.h>

// perform the bubble sort

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

for (int step = 0; step < size - 1; ++step) {

for (int i = 0; i < size - step - 1; ++i) {

if (array[i] > array[i + 1]) {

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

}

}

}

}

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

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

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

}

printf("\n");

}

int main() {

int data[] = {-2, 45, 0, 11, -9};

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

bubbleSort(data, size);

printf("Sorted Array in Ascending Order:\n");

printArray(data, size);

}

**OUTPUT** 🡪 Sorted Array in Ascending Order:

-9 -2 0 11 45

1. **Write a program that implements insertion sort.**

#include <stdio.h>

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

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

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

}

printf("\n");

}

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

for (int step = 1; step < size; step++) {

int key = array[step];

int j = step - 1;

while (key < array[j] && j >= 0) {

array[j + 1] = array[j];

--j;

}

array[j + 1] = key;

}

}

// Driver code

int main() {

int data[] = {9, 5, 1, 4, 3};

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

insertionSort(data, size);

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

printArray(data, size);

}

**OUTPUT** 🡪 Sorted array in ascending order:

1 3 4 5 9

1. **Write a program that implements selection sort.**

#include <stdio.h>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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

for (int step = 0; step < size - 1; step++) {

int min\_idx = step;

for (int i = step + 1; i < size; i++) {

if (array[i] < array[min\_idx])

min\_idx = i;

}

swap(&array[min\_idx], &array[step]);

}

}

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

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

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

}

printf("\n");

}

// driver code

int main() {

int data[] = {20, 12, 10, 15, 2};

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

selectionSort(data, size);

printf("Sorted array in Acsending Order:\n");

printArray(data, size);

}

**OUTPUT** 🡪 Sorted array in Acsending Order:

2 10 12 15 20

1. **Write a program to implement merge sort.**

#include <stdio.h>

void merge(int arr[], int p, int q, int r) {

int n1 = q - p + 1;

int n2 = r - q;

int L[n1], M[n2];

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

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

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

M[j] = arr[q + 1 + j];

int i, j, k;

i = 0;

j = 0;

k = p;

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

if (L[i] <= M[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = M[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = M[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 arr[], int size) {

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

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

printf("\n");

}

// Driver program

int main() {

int arr[] = {6, 5, 12, 10, 9, 1};

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

mergeSort(arr, 0, size - 1);

printf("Sorted array: \n");

printArray(arr, size);

}

**OUTPUT** 🡪 Sorted array:

1 5 6 9 10 12

1. **Write a program to Sort a given set of elements using the 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);

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");

}

// main function

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** 🡪 Unsorted Array

8 7 2 1 0 9 6

Sorted array in ascending order:

0 1 2 6 7 8 9

1. **Write a program that implements Linear search.**

// Linear Search in C

#include <stdio.h>

int search(int array[], int n, int x) {

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

if (array[i] == x)

return i;

return -1;

}

int main() {

int array[] = {2, 4, 0, 1, 9};

int x = 4;

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

int result = search(array, n, x);

if (result =-1)

printf("Not found");

else:

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

}

**OUTPUT** 🡪 Element found at index: 1

1. **Write a program that implements binary search.**

#include <stdio.h>

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

if (high >= low) {

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

if (array[mid] == x)

return mid;

if (array[mid] > x)

return binarySearch(array, x, low, mid - 1);

return binarySearch(array, x, mid + 1, high);

}

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);

}

**OUTPUT** 🡪 Element is found at index 1

1. **Write a program to implement Binary search tree.**

#include <stdio.h>

#include <stdlib.h>

struct node {

int key;

struct node \*left, \*right;

};

// Create a node

struct node \*newNode(int item) {

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// Inorder Traversal

void inorder(struct node \*root) {

if (root != NULL) {

inorder(root->left);

printf("%d -> ", root->key);

inorder(root->right);

}

}

// Insert a node

struct node \*insert(struct node \*node, int key) {

if (node = NULL) return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

// Find the inorder successor

struct node \*minValueNode(struct node \*node) {

struct node \*current = node;

// Find the leftmost leaf

while (current && current->left != NULL)

current = current->left;

return current;

}

// Deleting a node

struct node \*deleteNode(struct node \*root, int key) {

if (root == NULL) return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else {

// If the node is with only one child or no child

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct node \*temp = root->left;

free(root);

return temp;

}

// If the node has two children

struct node \*temp = minValueNode(root->right);

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

return root;

}

// Driver code

int main() {

struct node \*root = NULL;

root = insert(root, 8);

root = insert(root, 3);

root = insert(root, 1);

root = insert(root, 6);

root = insert(root, 7);

root = insert(root, 10);

root = insert(root, 14);

root = insert(root, 4);

printf("Inorder traversal: ");

inorder(root);

printf("\nAfter deleting 10\n");

root = deleteNode(root, 10);

printf("Inorder traversal: ");

inorder(root);

}

**OUTPUT** 🡪 Inorder traversal: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 10 -> 14 ->

After deleting 10

Inorder traversal: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 14 ->

1. . **Write a program to find optimal ordering of matrix multiplication.**

#include <stdio.h>

#include <string.h>

#define INT\_MAX 999999

int mc[50][50];

int min(int a, int b){

if(a < b)

return a;

else

return b;

}

int DynamicProgramming(int c[], int i, int j){

if (i == j) {

return 0;

}

if (mc[i][j] != -1) {

return

mc[i][j];

}

mc[i][j] = INT\_MAX;

for (int k = i; k < j; k++) {

mc[i][j] = min(mc[i][j], DynamicProgramming(c, i, k) + DynamicProgramming(c, k + 1, j) + c[i - 1] \* c[k] \* c[j]);

}

return mc[i][j];

}

int Matrix(int c[], int n){

int i = 1, j = n - 1;

return DynamicProgramming(c, i, j);

}

int main(){

int arr[] = { 1, 2, 3, 4, 3 };

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

memset(mc, -1, sizeof mc);

printf("Minimum number of multiplications is: %d", Matrix(arr, n));

}

**OUTPUT** 🡪 Minimum number of multiplications is: 30

1. **Implement 0/1 Knapsack problem using Dynamic Programming.**

#include <stdio.h>

#include <string.h>

int findMax(int n1, int n2){

if(n1>n2) {

return n1;

} else {

return n2;

}

}

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

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

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

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

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

K[i][w] = 0;

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

K[i][w] = findMax(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[5] = {70, 20, 50};

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

int W = 30;

int len = sizeof val / sizeof val[0];

printf("Maximum Profit achieved with this knapsack: %d", knapsack(W, wt, val, len));

}

**OUTPUT** 🡪 Maximum Profit achieved with this knapsack: 120

1. **Write a program that implements knapsack using greedy**.

#include <stdio.h>

int n = 5;

int p[10] = {3, 3, 2, 5, 1};

int w[10] = {10, 15, 10, 12, 8};

int W = 10;

int main(){

int cur\_w;

float tot\_v;

int i, maxi;

int used[10];

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

used[i] = 0;

cur\_w = W;

while (cur\_w > 0) {

maxi = -1;

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

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

((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))

maxi = i;

used[maxi] = 1;

cur\_w -= p[maxi];

tot\_v += w[maxi];

if (cur\_w >= 0)

printf("Added object %d (%d, %d) completely in the bag. Space left: %d.\n", maxi + 1, w[maxi], p[maxi], cur\_w);

else {

printf("Added %d%% (%d, %d) of object %d in the bag.\n", (int)((1 + (float)cur\_w/p[maxi]) \* 100), w[maxi], p[maxi], maxi + 1);

tot\_v -= w[maxi];

tot\_v += (1 + (float)cur\_w/p[maxi]) \* w[maxi];

}

}

printf("Filled the bag with objects worth %.2f.\n", tot\_v);

return 0;

}

**OUTPUT** 🡪 Added object 5 (8, 1) completely in the bag. Space left: 9.

Added object 2 (15, 3) completely in the bag. Space left: 6.

Added object 3 (10, 2) completely in the bag. Space left: 4.

Added object 1 (10, 3) completely in the bag. Space left: 1.

Added 19% (12, 5) of object 4 in the bag.

Filled the bag with objects worth 45.40.

1. **Write a program to implement file compression (and un-compression) using Huffman’s algorithm**.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_TREE\_HT 50

struct MinHNode {

char item;

unsigned freq;

struct MinHNode \*left, \*right;

};

struct MinHeap {

unsigned size;

unsigned capacity;

struct MinHNode \*\*array;

};

// Create nodes

struct MinHNode \*newNode(char item, unsigned freq) {

struct MinHNode \*temp = (struct MinHNode \*)malloc(sizeof(struct MinHNode));

temp->left = temp->right = NULL;

temp->item = item;

temp->freq = freq;

return temp;

}

// Create min heap

struct MinHeap \*createMinH(unsigned capacity) {

struct MinHeap \*minHeap = (struct MinHeap \*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHNode \*\*)malloc(minHeap->capacity \* sizeof(struct MinHNode \*));

return minHeap;

}

// Function to swap

void swapMinHNode(struct MinHNode \*\*a, struct MinHNode \*\*b) {

struct MinHNode \*t = \*a;

\*a = \*b;

\*b = t;

}

// Heapify

void minHeapify(struct MinHeap \*minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// Check if size if 1

int checkSizeOne(struct MinHeap \*minHeap) {

return (minHeap->size == 1);

}

// Extract min

struct MinHNode \*extractMin(struct MinHeap \*minHeap) {

struct MinHNode \*temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// Insertion function

void insertMinHeap(struct MinHeap \*minHeap, struct MinHNode \*minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

void buildMinHeap(struct MinHeap \*minHeap) {

int n = minHeap->size - 1;

int i;

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

minHeapify(minHeap, i);

}

int isLeaf(struct MinHNode \*root) {

return !(root->left) && !(root->right);

}

struct MinHeap \*createAndBuildMinHeap(char item[], int freq[], int size) {

struct MinHeap \*minHeap = createMinH(size);

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

minHeap->array[i] = newNode(item[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

struct MinHNode \*buildHuffmanTree(char item[], int freq[], int size) {

struct MinHNode \*left, \*right, \*top;

struct MinHeap \*minHeap = createAndBuildMinHeap(item, freq, size);

while (!checkSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

void printHCodes(struct MinHNode \*root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

printHCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printHCodes(root->right, arr, top + 1);

}

if (isLeaf(root)) {

printf(" %c | ", root->item);

printArray(arr, top);

}

}

// Wrapper function

void HuffmanCodes(char item[], int freq[], int size) {

struct MinHNode \*root = buildHuffmanTree(item, freq, size);

int arr[MAX\_TREE\_HT], top = 0;

printHCodes(root, arr, top);

}

void printArray(int arr[], int n) {

int i;

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

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

printf("\n");

}

int main() {

char arr[] = {'A', 'B', 'C', 'D'};

int freq[] = {5, 1, 6, 3};

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

printf(" Char | Huffman code ");

printf("\n--------------------\n");

HuffmanCodes(arr, freq, size);

}

**OUTPUT** 🡪 Char | Huffman code

--------------------

C | 0

B | 100

D | 101

A | 11

1. **Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.**

#include <stdio.h>

#include <stdlib.h>

const int inf = 999999;

int k, a, b, u, v, n, ne = 1;

int mincost = 0;

int cost[3][3] = {{0, 10, 20},{12, 0,15},{16, 18, 0}};

int p[9] = {0};

int applyfind(int i)

{

while(p[i] != 0)

i=p[i];

return i;

}

int applyunion(int i,int j)

{

if(i!=j) {

p[j]=i;

return 1;

}

return 0;

}

int main()

{

n = 3;

int i, j;

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

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

if (cost[i][j] == 0) {

cost[i][j] = inf;

} } }

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

while(ne < n) {

int min\_val = inf;

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

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

if(cost[i][j] < min\_val) {

min\_val = cost[i][j];

a = u = i;

b = v = j;

}

}

}

u = applyfind(u);

v = applyfind(v);

if(applyunion(u, v) != 0) {

printf("%d -> %d\n", a, b);

mincost +=min\_val;

}

cost[a][b] = cost[b][a] = 999;

ne++;

}

printf("Minimum cost = %d",mincost);

return 0;

}

**OUTPUT** 🡪 Minimum Cost Spanning Tree:

0 -> 1

1 -> 2

Minimum cost = 25

1. **Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.**

#include<stdio.h>

#include<stdlib.h>

#define inf 99999

#define MAX 10

int G[MAX][MAX] = {

{0, 19, 8},

{21, 0, 13},

{15, 18, 0}

};

int S[MAX][MAX], n;

int prims();

int main(){

int i, j, cost;

n = 3;

cost=prims();

printf("Spanning tree:");

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

printf("\n");

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

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

}

printf("\nMinimum cost = %d", cost);

return 0;

}

int prims(){

int C[MAX][MAX];

int u, v, min\_dist, dist[MAX], from[MAX];

int visited[MAX],ne,i,min\_cost,j;

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

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

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

C[i][j]=inf;

else

C[i][j]=G[i][j];

S[i][j]=0;

}

dist[0]=0;

visited[0]=1;

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

dist[i] = C[0][i];

from[i] = 0;

visited[i] = 0;

}

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

ne = n-1; //no. of edges to be added

while(ne > 0) {

min\_dist = inf;

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

if(visited[i] == 0 && dist[i] < min\_dist) {

v = i;

min\_dist = dist[i];

}

u = from[v];

S[u][v] = dist[v];

S[v][u] = dist[v];

ne--;

visited[v]=1;

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

if(visited[i] == 0 && C[i][v] < dist[i]) {

dist[i] = C[i][v];

from[i] = v;

}

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

}

return(min\_cost);

}

**OUTPUT** 🡪 Spanning tree:

0 0 8

0 0 13

8 13 0

Minimum cost = 26

1. **Write a program to implements Dijkstra’s algorithm.**

#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);

return 0;

}

**OUTPUT** 🡪 Distance from source to 1: 3

Distance from source to 2: 1

Distance from source to 3: 2

Distance from source to 4: 4

Distance from source to 5: 4

Distance from source to 6: 3

1. **Write a program to implement All-Pairs Shortest Paths Problem using Floyd's algorithm.**

#include <stdio.h>

// defining the number of vertices

#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

void floydWarshall(int graph[][nV]) {

int matrix[nV][nV], i, j, k;

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

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

matrix[i][j] = graph[i][j];

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

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

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

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

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

}

}

}

printMatrix(matrix);

}

void printMatrix(int matrix[][nV]) {

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

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

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

printf("%4s", "INF");

else

printf("%4d", matrix[i][j]);

}

printf("\n"); } }

int main() {

int graph[nV][nV] = {{0, 3, INF, 5},

{2, 0, INF, 4},

{INF, 1, 0, INF},

{INF, INF, 2, 0}};

floydWarshall(graph);

}

**OUTPUT** 🡪 0 3 7 5

2 0 6 4

3 1 0 5

5 3 2 0

1. **Find a subset of a given set S = {s1, s2 ,..,sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1, 2, 6} and {1, 8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.**

#include <stdio.h>

#include <stdbool.h>

bool isSubsetSum(int set[], int n, int sum)

{

bool subset[n + 1][sum + 1];

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

subset[i][0] = true;

for (int i = 1; i <= sum; i++)

subset[0][i] = false;

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

for (int j = 1; j <= sum; j++) {

if (j < set[i - 1])

subset[i][j] = subset[i - 1][j];

if (j >= set[i - 1])

subset[i][j]

= subset[i - 1][j]

|| subset[i - 1][j - set[i - 1]];

} }

return subset[n][sum];

}

// Driver code

int main()

{

int set[] = { 3, 34, 4, 12, 5, 2 };

int sum = 9;

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

if (isSubsetSum(set, n, sum) == true)

printf("Found a subset with given sum");

else

printf("No subset with given sum");

return 0;

}

**OUTPUT** 🡪 Found a subset with given sum

1. **Implement N Queen's problem using back tracking.**

#include <stdio.h>

#include <stdbool.h>

#define N 8

void printBoard(int board[N][N]) {

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

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

printf("%2d ", board[i][j]);

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col) {

int i, j;

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

if (board[row][i])

return false;

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

if (board[i][j])

return false;

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

if (board[i][j])

return false;

return true;

}

bool solveNQueens(int board[N][N], int col) {

if (col >= N)

return true;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQueens(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

void nQueens() {

int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0} };

if (solveNQueens(board, 0) == false) {

printf("Solution does not exist\n");

return;

}

printBoard(board);

}

// Main function

int main() {

nQueens();

return 0;

}

**OUTPUT**🡪 1 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 1

0 1 0 0 0 0 0 0

0 0 0 1 0 0 0 0

0 0 0 0 0 1 0 0

0 0 1 0 0 0 0 0

1. **Write a program to implement Graph Colouring using backtracking method.**

#include <stdio.h>

#include <stdbool.h>

#define V 4 // Number of vertices in the graph

bool isSafe(int v, bool graph[V][V], int color[], int c) {

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

if (graph[v][i] && c = color[i])

return false;

return true;

}

bool graphColoringUtil(bool graph[V][V], int m, int color[], int v) {

if (v == V)

return true;

for (int c = 1; c <= m; c++) {

if (isSafe(v, graph, color, c)) {

color[v] = c;

if (graphColoringUtil(graph, m, color, v + 1))

return true;

color[v] = 0;

}

}

return false;

}

bool graphColoring(bool graph[V][V], int m) {

int color[V];

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

color[i] = 0;

if (!graphColoringUtil(graph, m, color, 0)) {

printf("Solution does not exist\n");

return false;

}

printf("Solution exists and the assigned colors are: \n");

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

printf("Vertex %d --> Color %d\n", i, color[i]);

return true;

}

// Main function

int main() {

bool graph[V][V] = { {0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0} };

int m = 3;

graphColoring(graph, m);

return 0;

}

**OUTPUT** 🡪 Solution exists and the assigned colors are:

Vertex 0 --> Color 1

Vertex 1 --> Color 2

Vertex 2 --> Color 3

Vertex 3 --> Color 2

1. **Write a program to implement Travelling sales person using branch and bound.**

#include <stdio.h>

using namespace std;

const int N = 4;

int final\_path[N+1];

bool visited[N];

int final\_res = INT\_MAX;

void copyToFinal(int curr\_path[])

{

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

final\_path[i] = curr\_path[i];

final\_path[N] = curr\_path[0];

}

int firstMin(int adj[N][N], int i)

{

int min = INT\_MAX;

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

if (adj[i][k]<min && i != k)

min = adj[i][k];

return min;

}

int secondMin(int adj[N][N], int i)

{

int first = INT\_MAX, second = INT\_MAX;

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

{

if (i == j)

continue;

if (adj[i][j] <= first)

{

second = first;

first = adj[i][j];

}

else if (adj[i][j] <= second &&adj[i][j] != first)

second = adj[i][j];

}

return second;

}

void TSPRec(int adj[N][N], int curr\_bound, int curr\_weight,int level, int curr\_path[])

{

if (level==N)

{

if (adj[curr\_path[level-1]][curr\_path[0]] != 0)

{

int curr\_res = curr\_weight +adj[curr\_path[level-1]][curr\_path[0]];

if (curr\_res < final\_res)

{

copyToFinal(curr\_path);

final\_res = curr\_res;

}

}

return;

}

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

{

if (adj[curr\_path[level-1]][i] != 0 &&visited[i] == false)

{

int temp = curr\_bound;

curr\_weight += adj[curr\_path[level-1]][i];

if (level==1)

curr\_bound -= ((firstMin(adj, curr\_path[level-1]) +

firstMin(adj, i))/2);

else:

curr\_bound -= ((secondMin(adj, curr\_path[level-1]) +

firstMin(adj, i))/2);

if (curr\_bound + curr\_weight < final\_res)

{

curr\_path[level] = i;

visited[i] = true;

TSPRec(adj, curr\_bound, curr\_weight, level+1,

curr\_path);

}

curr\_weight -= adj[curr\_path[level-1]][i];

curr\_bound = temp;

memset(visited, false, sizeof(visited));

for (int j=0; j<=level-1; j++)

visited[curr\_path[j]] = true;

}

}

}

void TSP(int adj[N][N])

{

int curr\_path[N+1];

int curr\_bound = 0;

memset(curr\_path, -1, sizeof(curr\_path));

memset(visited, 0, sizeof(curr\_path));

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

curr\_bound += (firstMin(adj, i) +secondMin(adj, i));

curr\_bound = (curr\_bound&1)? curr\_bound/2 + 1 : curr\_bound/2;

visited[0] = true;

curr\_path[0] = 0;

TSPRec(adj, curr\_bound, 0, 1, curr\_path);

}

// Driver code

int main()

{

int adj[N][N] = { {0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

TSP(adj);

printf("Minimum cost : %d\n", final\_res);

printf("Path Taken : ");

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

printf("%d ", final\_path[i]);

return 0;

}

**OUTPUT**🡪 Minimum cost : 80

Path Taken : 0 1 3 2 0

1. **Write a program to implement Travelling sales person using dynamic programming.**

#include <stdio.h>

#include <limits.h>

#define MAX 9999

int n = 4;

int distan[20][20] = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 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** 🡪 Minimum Distance Travelled -> 80

1. **Write a program to implement the backtracking algorithm for the Hamiltonian Circuits problem.**

#include <stdio.h>

#include <stdbool.h>

#define V 5 // Number of vertices

bool isSafe(int v, bool graph[V][V], int path[], int pos) {

if (!graph[path[pos - 1]][v])

return false;

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

if (path[i] == v)

return false;

}

return true;

}

void printSolution(int path[]) {

printf("Hamiltonian Circuit exists: ");

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

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

printf("%d\n", path[0]);

}

bool hamiltonianCircuitUtil(bool graph[V][V], int path[], int pos) {

if (pos == V) {

if (graph[path[pos - 1]][path[0]])

return true;

else

return false;

}

for (int v = 1; v < V; v++) {

if (isSafe(v, graph, path, pos)) {

path[pos] = v;

if (hamiltonianCircuitUtil(graph, path, pos + 1))

return true;

path[pos] = -1;

}

}

return false;

}

void hamiltonianCircuit(bool graph[V][V]) {

int path[V];

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

path[i] = -1;

path[0] = 0;

if (!hamiltonianCircuitUtil(graph, path, 1)) {

printf("No Hamiltonian Circuit exists in the graph\n");

return;

}

printSolution(path);

}

// Main function

int main() {

bool graph[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0},

};

hamiltonianCircuit(graph);

return 0;

}

**OUTPUT** 🡪 Hamiltonian Circuit exists: 0 1 2 4 3 0

1. **Write a program to implement greedy algorithm for job sequencing with deadlines.**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id; // Job ID

int deadline; // Deadline of the job

int profit; // Profit associated with the job

} Job;

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

return (\*(Job\*)b).profit - (\*(Job\*)a).profit;

}

void jobSequencing(Job jobs[], int n) {

qsort(jobs, n, sizeof(Job), compare);

int maxDeadline = 0;

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

if (jobs[i].deadline > maxDeadline)

maxDeadline = jobs[i].deadline; }

int result[maxDeadline];

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

result[i] = -1;

int totalProfit = 0;

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

for (int j = jobs[i].deadline - 1; j >= 0; j--) {

if (result[j] == -1) {

result[j] = jobs[i].id;

totalProfit += jobs[i].profit;

break;

} } }

printf("Scheduled Jobs: ");

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

if (result[i] != -1)

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

}

printf("\nTotal Profit: %d\n", totalProfit);

}

// Main function

int main() {

Job jobs[] = {{1, 2, 100}, {2, 1, 19}, {3, 2, 27}, {4, 1, 25}, {5, 3, 15}};

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

jobSequencing(jobs, n);

return 0;

}

**OUTPUT**🡪 Scheduled Jobs: 3 1 5

Total Profit: 142