**1. Matrix multiplication**

#include <iostream>

using namespace std;

int main()

{

int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;

cout<<"enter the number of row=";

cin>>r;

cout<<"enter the number of column=";

cin>>c;

cout<<"enter the first matrix element=\n";

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

{

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

{

cin>>a[i][j];

}

}

cout<<"enter the second matrix element=\n";

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

{

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

{

cin>>b[i][j];

}

}

cout<<"multiply of the matrix=\n";

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

{

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

{

mul[i][j]=0;

for(k=0;k<c;k++)

{

mul[i][j]+=a[i][k]\*b[k][j];

}

}

}

//for printing result

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

{

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

{

cout<<mul[i][j]<<" ";

}

cout<<"\n";

}

return 0;

}

**2. Stack using array**

/\* C++ program to implement basic stack

operations \*/

#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack {

int top;

public:

int a[MAX]; // Maximum size of Stack

Stack() { top = -1; }

bool push(int x);

int pop();

int peek();

bool isEmpty();

};

bool Stack::push(int x)

{

if (top >= (MAX - 1)) {

cout << "Stack Overflow";

return false;

}

else {

a[++top] = x;

cout << x << " pushed into stack\n";

return true;

}

}

int Stack::pop()

{

if (top < 0) {

cout << "Stack Underflow";

return 0;

}

else {

int x = a[top--];

return x;

}

}

int Stack::peek()

{

if (top < 0) {

cout << "Stack is Empty";

return 0;

}

else {

int x = a[top];

return x;

}

}

bool Stack::isEmpty()

{

return (top < 0);

}

// Driver program to test above functions

int main()

{

class Stack s;

s.push(10);

s.push(20);

s.push(30);

cout << s.pop() << " Popped from stack\n";

//print top element of stack after popping

cout << "Top element is : " << s.peek() << endl;

//print all elements in stack :

cout <<"Elements present in stack : ";

while(!s.isEmpty())

{

// print top element in stack

cout << s.peek() <<" ";

// remove top element in stack

s.pop();

}

return 0;

}

**3. Queue**

#include <iostream>

using namespace std;

int queue[100], n = 100, front = - 1, rear = - 1;

void Insert() {

int val;

if (rear == n - 1)

cout<<"Queue Overflow"<<endl;

else {

if (front == - 1)

front = 0;

cout<<"Insert the element in queue : "<<endl;

cin>>val;

rear++;

queue[rear] = val;

}

}

void Delete() {

if (front == - 1 || front > rear) {

cout<<"Queue Underflow ";

return ;

} else {

cout<<"Element deleted from queue is : "<< queue[front] <<endl;

front++;;

}

}

void Display() {

if (front == - 1)

cout<<"Queue is empty"<<endl;

else {

cout<<"Queue elements are : ";

for (int i = front; i <= rear; i++)

cout<<queue[i]<<" ";

cout<<endl;

}

}

int main() {

int ch;

cout<<"1) Insert element to queue"<<endl;

cout<<"2) Delete element from queue"<<endl;

cout<<"3) Display all the elements of queue"<<endl;

cout<<"4) Exit"<<endl;

do {

cout<<"Enter your choice : "<<endl;

cin>>ch;

switch (ch) {

case 1: Insert();

break;

case 2: Delete();

break;

case 3: Display();

break;

case 4: cout<<"Exit"<<endl;

break;

default: cout<<"Invalid choice"<<endl;

}

} while(ch!=4);

return 0;

}

**4. Single linked list**

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node \*next;

};

struct Node\* head = NULL;

void insert(int new\_data) {

struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = head;

head = new\_node;

}

void display() {

struct Node\* ptr;

ptr = head;

while (ptr != NULL) {

cout<< ptr->data <<" ";

ptr = ptr->next;

}

}

int main() {

insert(3);

insert(1);

insert(7);

insert(2);

insert(9);

cout<<"The linked list is: ";

display();

return 0;

}

**5. Stack using link list**

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node \*next;

};

struct Node\* top = NULL;

void push(int val) {

struct Node\* newnode = (struct Node\*) malloc(sizeof(struct Node));

newnode->data = val;

newnode->next = top;

top = newnode;

}

void pop() {

if(top==NULL)

cout<<"Stack Underflow"<<endl;

else {

cout<<"The popped element is "<< top->data <<endl;

top = top->next;

}

}

void display() {

struct Node\* ptr;

if(top==NULL)

cout<<"stack is empty";

else {

ptr = top;

cout<<"Stack elements are: ";

while (ptr != NULL) {

cout<< ptr->data <<" ";

ptr = ptr->next;

}

}

cout<<endl;

}

int main() {

int ch, val;

cout<<"1) Push in stack"<<endl;

cout<<"2) Pop from stack"<<endl;

cout<<"3) Display stack"<<endl;

cout<<"4) Exit"<<endl;

do {

cout<<"Enter choice: "<<endl;

cin>>ch;

switch(ch) {

case 1: {

cout<<"Enter value to be pushed:"<<endl;

cin>>val;

push(val);

break;

}

case 2: {

pop();

break;

}

case 3: {

display();

break;

}

case 4: {

cout<<"Exit"<<endl;

break;

}

default: {

cout<<"Invalid Choice"<<endl;

}

}

}while(ch!=4);

return 0;

}

**6. Circular queue**

#include <iostream>

using namespace std;

int cqueue[5];

int front = -1, rear = -1, n=5;

void insertCQ(int val) {

if ((front == 0 && rear == n-1) || (front == rear+1)) {

cout<<"Queue Overflow \n";

return;

}

if (front == -1) {

front = 0;

rear = 0;

} else {

if (rear == n - 1)

rear = 0;

else

rear = rear + 1;

}

cqueue[rear] = val ;

}

void deleteCQ() {

if (front == -1) {

cout<<"Queue Underflow\n";

return ;

}

cout<<"Element deleted from queue is : "<<cqueue[front]<<endl;

if (front == rear) {

front = -1;

rear = -1;

} else {

if (front == n - 1)

front = 0;

else

front = front + 1;

}

}

void displayCQ() {

int f = front, r = rear;

if (front == -1) {

cout<<"Queue is empty"<<endl;

return;

}

cout<<"Queue elements are :\n";

if (f <= r) {

while (f <= r){

cout<<cqueue[f]<<" ";

f++;

}

} else {

while (f <= n - 1) {

cout<<cqueue[f]<<" ";

f++;

}

f = 0;

while (f <= r) {

cout<<cqueue[f]<<" ";

f++;

}

}

cout<<endl;

}

int main() {

int ch, val;

cout<<"1)Insert\n";

cout<<"2)Delete\n";

cout<<"3)Display\n";

cout<<"4)Exit\n";

do {

cout<<"Enter choice : "<<endl;

cin>>ch;

switch(ch) {

case 1:

cout<<"Input for insertion: "<<endl;

cin>>val;

insertCQ(val);

break;

case 2:

deleteCQ();

break;

case 3:

displayCQ();

break;

case 4:

cout<<"Exit\n";

break;

default: cout<<"Incorrect!\n";

}

} while(ch != 4);

return 0;

}

**7. Double link list**

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node \*prev;

struct Node \*next;

};

struct Node\* head = NULL;

void insert(int newdata) {

struct Node\* newnode = (struct Node\*) malloc(sizeof(struct Node));

newnode->data = newdata;

newnode->prev = NULL;

newnode->next = head;

if(head != NULL)

head->prev = newnode ;

head = newnode;

}

void display() {

struct Node\* ptr;

ptr = head;

while(ptr != NULL) {

cout<< ptr->data <<" ";

ptr = ptr->next;

}

}

int main() {

insert(3);

insert(1);

insert(7);

insert(2);

insert(9);

cout<<"The doubly linked list is: ";

display();

return 0;

}

**8. Headed link list**

#include <iostream>

using namespace std;

// Define a structure for each node of the list

struct Node {

int data;

Node\* next;

};

int main() {

Node\* head = new Node; // Create the head node

head->data = 0; // Initialize head's data to 0

head->next = NULL; // Initialize head's next pointer to NULL

// Add new nodes to the list

for (int i = 1; i <= 5; i++) {

Node\* newNode = new Node; // Create a new node

newNode->data = i; // Set the new node's data to i

newNode->next = head->next; // Set the new node's next pointer to head's next pointer

head->next = newNode; // Set head's next pointer to the new node

}

// Print the contents of the list

Node\* current = head->next; // Start at the first node (after the head)

while (current != NULL) {

cout << current->data << " "; // Print the current node's data

current = current->next; // Move to the next node

}

cout << endl;

return 0;

}

**9. Convert infix to postfix**

#include <bits/stdc++.h>

using namespace std;

// Function to return precedence of operators

int prec(char c)

{

if (c == '^')

return 3;

else if (c == '/' || c == '\*')

return 2;

else if (c == '+' || c == '-')

return 1;

else

return -1;

}

// The main function to convert infix expression

// to postfix expression

void infixToPostfix(string s)

{

stack<char> st;

string result;

for (int i = 0; i < s.length(); i++) {

char c = s[i];

// If the scanned character is

// an operand, add it to output string.

if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')

|| (c >= '0' && c <= '9'))

result += c;

// If the scanned character is an

// ‘(‘, push it to the stack.

else if (c == '(')

st.push('(');

// If the scanned character is an ‘)’,

// pop and add to output string from the stack

// until an ‘(‘ is encountered.

else if (c == ')') {

while (st.top() != '(') {

result += st.top();

st.pop();

}

st.pop();

}

// If an operator is scanned

else {

while (!st.empty()

&& prec(s[i]) <= prec(st.top())) {

result += st.top();

st.pop();

}

st.push(c);

}

}

// Pop all the remaining elements from the stack

while (!st.empty()) {

result += st.top();

st.pop();

}

cout << result << endl;

}

// Driver code

int main()

{

string exp = "a+b\*(c^d-e)^(f+g\*h)-i";

// Function call

infixToPostfix(exp);

return 0;

}

**10. DFS graph traversing**

#include <bits/stdc++.h>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph {

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFS(int v)

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

// Driver's code

int main()

{

// Create a graph given in the above diagram

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal"

" (starting from vertex 2) \n";

// Function call

g.DFS(2);

return 0;

}

// improved by Vishnudev C

**11. BFS graph traversing**

#include <bits/stdc++.h>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph {

// No. of vertices

int V;

// Pointer to an array containing adjacency lists

vector<list<int> > adj;

public:

// Constructor

Graph(int V);

// Function to add an edge to graph

void addEdge(int v, int w);

// Prints BFS traversal from a given source s

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj.resize(V);

}

void Graph::addEdge(int v, int w)

{

// Add w to v’s list.

adj[v].push\_back(w);

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

vector<bool> visited;

visited.resize(V, false);

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

while (!queue.empty()) {

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (auto adjacent : adj[s]) {

if (!visited[adjacent]) {

visited[adjacent] = true;

queue.push\_back(adjacent);

}

}

}

}

// Driver code

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "

<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}

**12. Union of disjoint set**

#include <bits/stdc++.h>

using namespace std;

class DisjSet {

int \*rank, \*parent, n;

public:

// Constructor to create and

// initialize sets of n items

DisjSet(int n)

{

rank = new int[n];

parent = new int[n];

this->n = n;

makeSet();

}

// Creates n single item sets

void makeSet()

{

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

parent[i] = i;

}

}

// Finds set of given item x

int find(int x)

{

// Finds the representative of the set

// that x is an element of

if (parent[x] != x) {

// if x is not the parent of itself

// Then x is not the representative of

// his set,

parent[x] = find(parent[x]);

// so we recursively call Find on its parent

// and move i's node directly under the

// representative of this set

}

return parent[x];

}

// Do union of two sets represented

// by x and y.

void Union(int x, int y)

{

// Find current sets of x and y

int xset = find(x);

int yset = find(y);

// If they are already in same set

if (xset == yset)

return;

// Put smaller ranked item under

// bigger ranked item if ranks are

// different

if (rank[xset] < rank[yset]) {

parent[xset] = yset;

}

else if (rank[xset] > rank[yset]) {

parent[yset] = xset;

}

// If ranks are same, then increment

// rank.

else {

parent[yset] = xset;

rank[xset] = rank[xset] + 1;

}

}

};

// Driver Code

int main()

{

// Function Call

DisjSet obj(5);

obj.Union(0, 2);

obj.Union(4, 2);

obj.Union(3, 1);

if (obj.find(4) == obj.find(0))

cout << "Yes\n";

else

cout << "No\n";

if (obj.find(1) == obj.find(0))

cout << "Yes\n";

else

cout << "No\n";

return 0;

}

**: 13. Knapsack problem**

#include <iostream>

#include <climits>

using namespace std;

int knapSack(int v[], int w[], int n, int W) {

if (W < 0)

return INT\_MIN;

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

return 0;

int in = v[n] + knapSack(v, w, n - 1, W - w[n]);

int ex = knapSack(v, w, n - 1, W);

return max (in, ex);

}

int main() {

int v[] = { 10, 20, 30, 40, 60, 70 };

int w[] = { 1, 2, 3, 6, 7, 4 };

int W = 7;

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

cout << "Knapsack value is " << knapSack(v, w, n - 1, W);

return 0;

}

**14. Tower of Hanoi**

: #include <bits/stdc++.h>

using namespace std;

void towerOfHanoi(int n, char from\_rod, char to\_rod,

char aux\_rod)

{

if (n == 0) {

return;

}

towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);

cout << "Move disk " << n << " from rod " << from\_rod

<< " to rod " << to\_rod << endl;

towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

// Driver code

int main()

{

int N = 3;

// A, B and C are names of rods

towerOfHanoi(N, 'A', 'C', 'B');

return 0;

}

**15. Binary search tree**

#include <iostream>

using namespace std;

// Node class

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int data) {

this->data = data;

left = right = NULL;

}

};

// Binary search tree class

class BST {

public:

Node\* root;

BST() {

root = NULL;

}

// Insert a node in the binary search tree

void insert(int data) {

root = insert(root, data);

}

Node\* insert(Node\* node, int data) {

if (node == NULL) {

node = new Node(data);

} else if (data <= node->data) {

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

} else {

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

}

return node;

}

// Search for a node in the binary search tree

bool search(int data) {

return search(root, data);

}

bool search(Node\* node, int data) {

if (node == NULL) {

return false;

} else if (node->data == data) {

return true;

} else if (data <= node->data) {

return search(node->left, data);

} else {

return search(node->right, data);

}

}

// Inorder traversal of the binary search tree

void inorder() {

inorder(root);

}

void inorder(Node\* node) {

if (node == NULL) {

return;

}

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

// Delete a node from the binary search tree

void remove(int data) {

root = remove(root, data);

}

Node\* remove(Node\* node, int data) {

if (node == NULL) {

return NULL;

} else if (data < node->data) {

node->left = remove(node->left, data);

} else if (data > node->data) {

node->right = remove(node->right, data);

} else {

// Case 1: No child

if (node->left == NULL && node->right == NULL) {

delete node;

node = NULL;

}

// Case 2: One child

else if (node->left == NULL) {

Node\* temp = node;

node = node->right;

delete temp;

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

Node\* temp = node;

node = node->left;

delete temp;

}

// Case 3: Two children

else {

Node\* temp = findMin(node->right);

node->data = temp->data;

node->right = remove(node->right, temp->data);

}

}

return node;

}

// Find the node with the minimum value in the binary search tree

Node\* findMin(Node\* node) {

while (node->left != NULL) {

node = node->left;

}

return node;

}

};

int main() {

BST bst;

// Insert nodes

bst.insert(50);

bst.insert(30);

bst.insert(20);

bst.insert(40);

bst.insert(70);

bst.insert(60);

bst.insert(80);

// Print inorder traversal

bst.inorder();

cout << endl;

// Search for a

**16. Merge sort**

#include<iostream>

using namespace std;

void swapping(int &a, int &b) { //swap the content of a and b

int temp;

temp = a;

a = b;

b = temp;

}

void display(int \*array, int size) {

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

cout << array[i] << " ";

cout << endl;

}

void merge(int \*array, int l, int m, int r) {

int i, j, k, nl, nr;

//size of left and right sub-arrays

nl = m-l+1; nr = r-m;

int larr[nl], rarr[nr];

//fill left and right sub-arrays

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

larr[i] = array[l+i];

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

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

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

//marge temp arrays to real array

while(i < nl && j<nr) {

if(larr[i] <= rarr[j]) {

array[k] = larr[i];

i++;

}else{

array[k] = rarr[j];

j++;

}

k++;

}

while(i<nl) { //extra element in left array

array[k] = larr[i];

i++; k++;

}

while(j<nr) { //extra element in right array

array[k] = rarr[j];

j++; k++;

}

}

void mergeSort(int \*array, int l, int r) {

int m;

if(l < r) {

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

// Sort first and second arrays

mergeSort(array, l, m);

mergeSort(array, m+1, r);

merge(array, l, m, r);

}

}

int main() {

int n;

cout << "Enter the number of elements: ";

cin >> n;

int arr[n]; //create an array with given number of elements

cout << "Enter elements:" << endl;

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

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

mergeSort(arr, 0, n-1); //(n-1) for last index

cout << "Array after Sorting: ";

display(arr, n);

}

**17. Quick sort**

// C++ Implementation of the Quick Sort Algorithm.

#include <iostream>

using namespace std;

int partition(int arr[], int start, int end)

{

int pivot = arr[start];

int count = 0;

for (int i = start + 1; i <= end; i++) {

if (arr[i] <= pivot)

count++;

}

// Giving pivot element its correct position

int pivotIndex = start + count;

swap(arr[pivotIndex], arr[start]);

// Sorting left and right parts of the pivot element

int i = start, j = end;

while (i < pivotIndex && j > pivotIndex) {

while (arr[i] <= pivot) {

i++;

}

while (arr[j] > pivot) {

j--;

}

if (i < pivotIndex && j > pivotIndex) {

swap(arr[i++], arr[j--]);

}

}

return pivotIndex;

}

void quickSort(int arr[], int start, int end)

{

// base case

if (start >= end)

return;

// partitioning the array

int p = partition(arr, start, end);

// Sorting the left part

quickSort(arr, start, p - 1);

// Sorting the right part

quickSort(arr, p + 1, end);

}

int main()

{

int arr[] = { 9, 3, 4, 2, 1, 8 };

int n = 6;

quickSort(arr, 0, n - 1);

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

cout << arr[i] << " ";

}

return 0;

}

**18. Minimum cost spanning tree**

#include <bits/stdc++.h>

using namespace std;

// 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[]

void printMST(int parent[], int graph[V][V])

{

cout << "Edge \tWeight\n";

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

cout << parent[i] << " - " << i << " \t"

<< graph[i][parent[i]] << " \n";

}

// 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;

}

**19. Single source shortest path**

#include <limits.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

int printSolution(int dist[], int n)

{

printf("Vertex Distance from Source\n");

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

printf("%d \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, V);

}

// driver program to test above function

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 } };

dijkstra(graph, 0);

return 0;

}

**20. Multi stage graph**

// CPP program to find shortest distance

// in a multistage graph.

#include<bits/stdc++.h>

using namespace std;

#define N 8

#define INF INT\_MAX

// Returns shortest distance from 0 to

// N-1.

int shortestDist(int graph[N][N]) {

// dist[i] is going to store shortest

// distance from node i to node N-1.

int dist[N];

dist[N-1] = 0;

// Calculating shortest path for

// rest of the nodes

for (int i = N-2 ; i >= 0 ; i--)

{

// Initialize distance from i to

// destination (N-1)

dist[i] = INF;

// Check all nodes of next stages

// to find shortest distance from

// i to N-1.

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

{

// Reject if no edge exists

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

continue;

// We apply equation to

// distance to target through j.

// and compare with minimum distance

// so far.

dist[i] = min(dist[i], graph[i][j] +

dist[j]);

}

}

return dist[0];

}

// Driver code

int main()

{

// Graph stored in the form of an

// adjacency Matrix

int graph[N][N] =

{{INF, 1, 2, 5, INF, INF, INF, INF},

{INF, INF, INF, INF, 4, 11, INF, INF},

{INF, INF, INF, INF, 9, 5, 16, INF},

{INF, INF, INF, INF, INF, INF, 2, INF},

{INF, INF, INF, INF, INF, INF, INF, 18},

{INF, INF, INF, INF, INF, INF, INF, 13},

{INF, INF, INF, INF, INF, INF, INF, 2},

{INF, INF, INF, INF, INF, INF, INF, INF}};

cout << shortestDist(graph);

return 0;

}

**21. All pair shortest path**

#include<iostream>

#include<iomanip>

#define NODE 7

#define INF 999

using namespace std;

//Cost matrix of the graph

int costMat[NODE][NODE] = {

{0, 3, 6, INF, INF, INF, INF},

{3, 0, 2, 1, INF, INF, INF},

{6, 2, 0, 1, 4, 2, INF},

{INF, 1, 1, 0, 2, INF, 4},

{INF, INF, 4, 2, 0, 2, 1},

{INF, INF, 2, INF, 2, 0, 1},

{INF, INF, INF, 4, 1, 1, 0}

};

void floydWarshal(){

int cost[NODE][NODE]; //defind to store shortest distance from any node to any node

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

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

cost[i][j] = costMat[i][j]; //copy costMatrix to new matrix

for(int k = 0; k<NODE; k++){

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

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

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

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

}

cout << "The matrix:" << endl;

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

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

cout << setw(3) << cost[i][j];

cout << endl;

}

}

int main(){

floydWarshal();

}

**22. 0/1 knapsack problem**

#include <iostream>

#include <climits>

using namespace std;

int knapSack(int v[], int w[], int n, int W) {

if (W < 0)

return INT\_MIN;

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

return 0;

int in = v[n] + knapSack(v, w, n - 1, W - w[n]);

int ex = knapSack(v, w, n - 1, W);

return max (in, ex);

}

int main() {

int v[] = { 10, 20, 30, 40, 60, 70 };

int w[] = { 1, 2, 3, 6, 7, 4 };

int W = 7;

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

cout << "Knapsack value is " << knapSack(v, w, n - 1, W);

return 0;

}

**23. Traveling salesman problem**

#include <bits/stdc++.h>

using namespace std;

#define V 4

// implementation of traveling Salesman Problem

int travllingSalesmanProblem(int graph[][V], int s)

{

// store all vertex apart from source vertex

vector<int> vertex;

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

if (i != s)

vertex.push\_back(i);

// store minimum weight Hamiltonian Cycle.

int min\_path = INT\_MAX;

do {

// store current Path weight(cost)

int current\_pathweight = 0;

// compute current path weight

int k = s;

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

current\_pathweight += graph[k][vertex[i]];

k = vertex[i];

}

current\_pathweight += graph[k][s];

// update minimum

min\_path = min(min\_path, current\_pathweight);

} while (

next\_permutation(vertex.begin(), vertex.end()));

return min\_path;

}

// Driver Code

int main()

{

// matrix representation of graph

int graph[][V] = { { 0, 10, 15, 20 },

{ 10, 0, 35, 25 },

{ 15, 35, 0, 30 },

{ 20, 25, 30, 0 } };

int s = 0;

cout << travllingSalesmanProblem(graph, s) << endl;

return 0;

}

**24. Sum of subset**

#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;

}

**25. Hamiltonian cycle**

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 20;

int n;

int graph[MAXN][MAXN];

bool hamCycleUtil(int path[], int pos, bool visited[]) {

if (pos == n) {

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

return true;

else

return false;

}

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

if (graph[path[pos - 1]][v] == 1 && !visited[v]) {

visited[v] = true;

path[pos] = v;

if (hamCycleUtil(path, pos + 1, visited))

return true;

visited[v] = false;

}

}

return false;

}

void hamCycle() {

int path[MAXN];

bool visited[MAXN];

memset(visited, false, sizeof visited);

path[0] = 0;

visited[0] = true;

if (hamCycleUtil(path, 1, visited)) {

cout << "Hamiltonian cycle exists:" << endl;

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

cout << path[i] << " ";

cout << path[0] << endl;

} else {

cout << "Hamiltonian cycle does not exist." << endl;

}

}

int main() {

cout << "Enter the number of vertices: ";

cin >> n;

cout << "Enter the adjacency matrix: " << endl;

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

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

cin >> graph[i][j];

}

}

hamCycle();

return 0;

}