13. Given a (directed/undirected) graph, design an algorithm and implement it using a program to find if a path exists between two given vertices or not. (Hint: use DFS)

#include <iostream>

#include <vector>

using namespace std;

bool DFS(vector<vector<int>>& graph, int source, int destination, vector<bool>& visited) {

// Mark the current node as visited

visited[source] = true;

if (source == destination) {

return true;

}

// Traverse all the adjacent vertices of the current node

for (int i = 0; i < graph[source].size(); i++) {

// If the adjacent vertex is not visited and there is an edge between the current node and the adjacent vertex

if (!visited[i] && graph[source][i] == 1) {

// Recursively call DFS on the adjacent vertex

if (DFS(graph, i, destination, visited)) {

return true;

}

}

}

return false;

}

// Function to check if a path exists between two vertices in a graph

bool isPathExists(vector<vector<int>>& graph, int source, int destination) {

// Get the number of vertices in the graph

int numVertices = graph.size();

// Create a visited array to keep track of visited vertices

vector<bool> visited(numVertices, false);

// Perform DFS traversal from the source vertex

return DFS(graph, source, destination, visited);

}

int main() {

int numVertices;

cin >> numVertices;

// Create an adjacency matrix to represent the graph

vector<vector<int>> graph(numVertices, vector<int>(numVertices));

// Input the adjacency matrix

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

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

cin >> graph[i][j];

}

}

int source, destination;

cin >> source >> destination;

if (isPathExists(graph, source - 1, destination - 1)) {

cout << "Yes Path Exists" << endl;

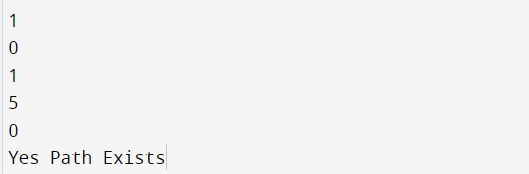
} else {

cout << "No Such Path Exists" << endl;

}

return 0;

}



14. Given a graph, design an algorithm and implement it using a program to find if a graph is bipartite or not. (Hint: use BFS)

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Function to check if a graph is bipartite or not

bool isBipartite(vector<vector<int>>& graph, int source) {

int numVertices = graph.size();

// Create a color array to store the color of each vertex

vector<int> color(numVertices, -1);

// Assign the color of the source vertex as 0

color[source] = 0;

// Create a queue for BFS traversal

queue<int> q;

q.push(source);

// Perform BFS traversal

while (!q.empty()) {

int current = q.front();

q.pop();

// Check all the neighbors of the current vertex

for (int i = 0; i < graph[current].size(); i++) {

int neighbor = graph[current][i];

// If the neighbor has not been visited yet

if (color[neighbor] == -1) {

// Assign a different color to the neighbor

color[neighbor] = 1 - color[current];

q.push(neighbor);

}

// If the neighbor has already been visited and has the same color as the current vertex

else if (color[neighbor] == color[current]) {

return false;

}

}

}

return true;

}

int main() {

int numVertices;

cin >> numVertices;

// Create an adjacency list to represent the graph

vector<vector<int>> graph(numVertices);

// Input the adjacency list

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

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

int edge;

cin >> edge;

if (edge == 1) {

graph[i].push\_back(j);

}

}

}

if (isBipartite(graph, 0)) {

cout << "Yes Bipartite" << endl;

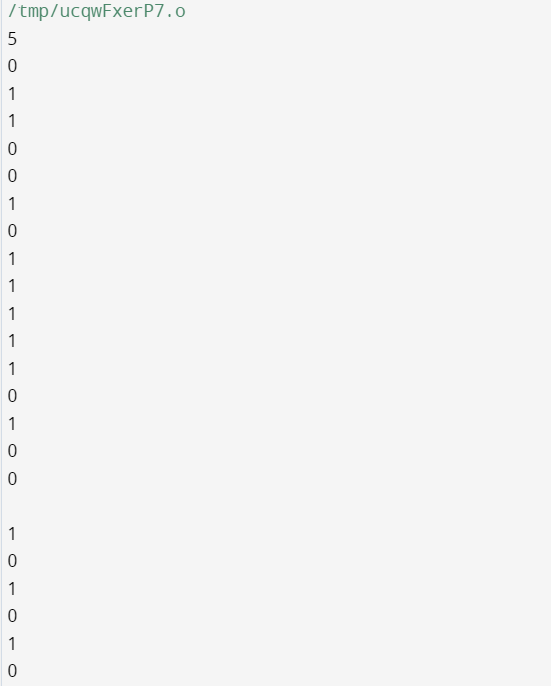
} else {

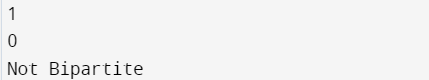
cout << "Not Bipartite" << endl;

}

return 0;

}





15. Given a directed graph, design an algorithm and implement it using a program to find whether cycle exists in the graph or not.

#include <iostream>

#include <vector>

using namespace std;

bool isCyclicHelper(vector<vector<int>>& graph, int current, vector<bool>& visited, vector<bool>& recursionStack) {

visited[current] = true;

recursionStack[current] = true;

// Check all the neighbors of the current vertex

for (int i = 0; i < graph[current].size(); i++) {

int neighbor = graph[current][i];

// If the neighbor has not been visited, recursively check if it is part of a cycle

if (!visited[neighbor]) {

if (isCyclicHelper(graph, neighbor, visited, recursionStack)) {

return true;

}

}

// If the neighbor is already visited and present in the recursion stack, it is part of a cycle

else if (recursionStack[neighbor]) {

return true;

}

}

// Remove the current vertex from the recursion stack

recursionStack[current] = false;

return false;

}

bool isCyclic(vector<vector<int>>& graph) {

int numVertices = graph.size();

// Create arrays to keep track of visited vertices and vertices in the recursion stack

vector<bool> visited(numVertices, false);

vector<bool> recursionStack(numVertices, false);

// Perform DFS traversal for each unvisited vertex

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

if (!visited[i]) {

if (isCyclicHelper(graph, i, visited, recursionStack)) {

return true;

}

}

}

return false;

}

int main() {

int numVertices;

cin >> numVertices;

// Create an adjacency list to represent the graph

vector<vector<int>> graph(numVertices);

// Input the adjacency list

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

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

int edge;

cin >> edge;

if (edge == 1) {

graph[i].push\_back(j);

}

}

}

if (isCyclic(graph)) {

cout << "Yes Cycle Exists" << endl;

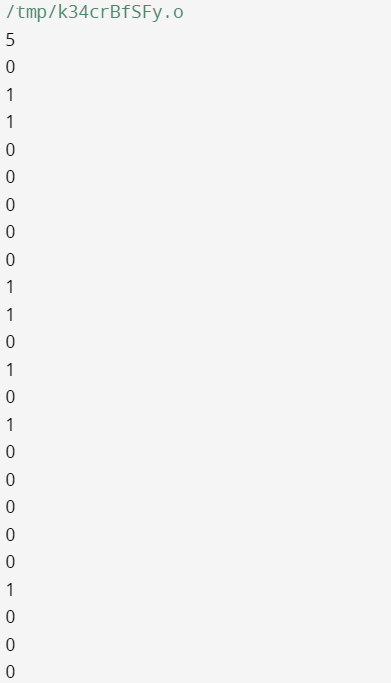
} else {

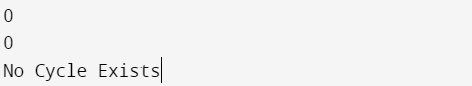
cout << "No Cycle Exists" << endl;

}

return 0;

}





16. Given a graph, Design an algorithm and implement it using a program to implement FloydWarshall all pair shortest path algorithm.

#include <iostream>

#include <vector>

using namespace std;

#define INF 99999

void floydWarshall(vector<vector<int>>& graph, int V) {

vector<vector<int>> dist(graph); // Initialize the distance matrix with the input graph

// Compute shortest distances between all pairs of vertices

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

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

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

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

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

}

}

}

}

// Print the resulting distance matrix

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

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

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

cout << "INF ";

} else {

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

}

}

cout << endl;

}

}

int main() {

int V;

cin >> V;

vector<vector<int>> graph(V, vector<int>(V));

// Read the input graph

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

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

cin >> graph[i][j];

}

}

floydWarshall(graph, V);

return 0;

}

17. Given a knapsack of maximum capacity w. N items are provided, each having its own value and weight. You have to Design an algorithm and implement it using a program to find the list of the selected items such that the final selected content has weight w and has maximum value. You can take fractions of items,i.e. the items can be broken into smaller pieces so that you have to carry only a fraction xi of item i, where 0 ≤xi≤ 1.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Item {

int weight;

int value;

};

bool compare(Item a, Item b) {

double ratioA = (double)a.value / a.weight;

double ratioB = (double)b.value / b.weight;

return ratioA > ratioB;

}

void fractionalKnapsack(vector<int>& weights, vector<int>& values, int capacity) {

int n = weights.size();

vector<Item> items(n);

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

items[i].weight = weights[i];

items[i].value = values[i];

}

sort(items.begin(), items.end(), compare);

int currentWeight = 0;

double totalValue = 0.0;

vector<pair<int, double>> selectedItems;

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

if (currentWeight + items[i].weight <= capacity) {

currentWeight += items[i].weight;

totalValue += items[i].value;

selectedItems.push\_back({ items[i].weight, 1.0 });

} else {

double remainingWeight = capacity - currentWeight;

double fraction = remainingWeight / items[i].weight;

currentWeight += remainingWeight;

totalValue += fraction \* items[i].value;

selectedItems.push\_back({ items[i].weight, fraction });

break;

}

}

// Print the maximum value achieved

cout << "Maximum value: " << totalValue << endl;

// Print the list of selected items with their fractions

cout << "Item-weight" << endl;

for (auto item : selectedItems) {

cout << item.first << "-" << item.second << endl;

}

}

int main() {

int n;

cin >> n;

vector<int> weights(n);

vector<int> values(n);

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

cin >> weights[i];

}

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

cin >> values[i];

}

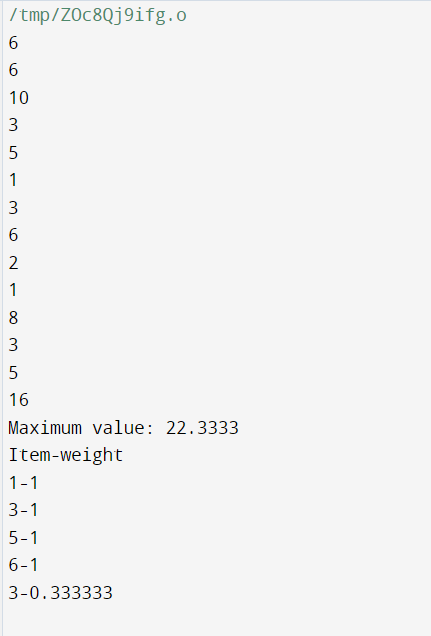
int capacity;

cin >> capacity;

fractionalKnapsack(weights, values, capacity);

return 0;

}



18. Given a knapsack of maximum capacity w. N items are provided, each having its own value and weight. Design an algorithm and implement it using a program to find the list of the selected items such that the final selected content has weight <= w and has maximum value. Here, you cannot break an item i.e. either pick the complete item or don't pick it. (0-1 property).

#include <iostream>

#include <vector>

using namespace std;

struct Item {

int weight;

int value;

};

void knapsack(int capacity, vector<Item>& items) {

int n = items.size();

vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

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

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

if (items[i - 1].weight <= j) {

dp[i][j] = max(dp[i - 1][j], items[i - 1].value + dp[i - 1][j - items[i - 1].weight]);

} else {

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

}

}

}

int maxValue = dp[n][capacity];

cout << "Value = " << maxValue << endl;

cout << "Weights selected: ";

int w = capacity;

for (int i = n; i > 0 && maxValue > 0; i--) {

if (maxValue == dp[i - 1][w]) {

continue;

} else {

cout << items[i - 1].weight << " ";

maxValue -= items[i - 1].value;

w -= items[i - 1].weight;

}

}

cout << endl;

cout << "Values of selected weights: ";

w = capacity;

for (int i = n; i > 0 && maxValue > 0; i--) {

if (maxValue == dp[i - 1][w]) {

continue;

} else {

cout << items[i - 1].value << " ";

maxValue -= items[i - 1].value;

w -= items[i - 1].weight;

}

}

cout << endl;

}

int main() {

int n;

cin >> n;

vector<Item> items(n);

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

cin >> items[i].weight;

}

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

cin >> items[i].value;

}

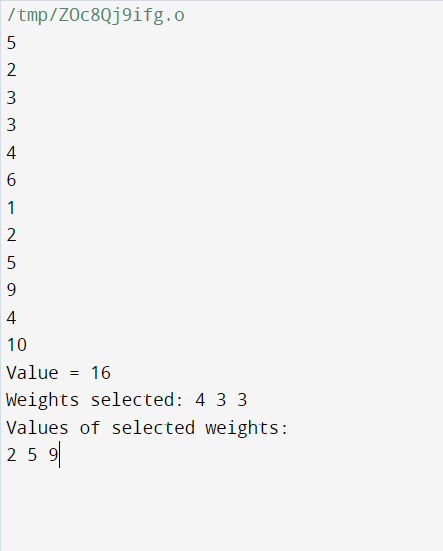
int capacity;

cin >> capacity;

knapsack(capacity, items);

return 0;

}



20. Given an unsorted array of elements, design an algorithm and implement it using a program to find whether majority element exists or not. Also find median of the array. A majority element is an element that appears more than n/2 times, where n is the size of array.

#include <iostream>

#include <algorithm>

using namespace std;

bool hasMajorityElement(int arr[], int n) {

sort(arr, arr + n);

int middle = arr[n / 2];

int count = 0;

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

if (arr[i] == middle) {

count++;

}

}

if (count > n / 2) {

return true;

}

return false;

}

int findMedian(int arr[], int n) {

sort(arr, arr + n);

if (n % 2 == 0) {

return (arr[n / 2 - 1] + arr[n / 2]) / 2;

}

return arr[n / 2];

}

int main() {

int n;

cin >> n;

int arr[n];

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

cin >> arr[i];

}

if (hasMajorityElement(arr, n)) {

cout << "yes" << endl;

} else {

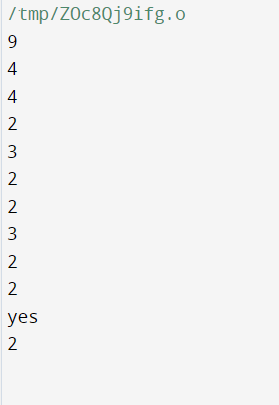
cout << "no" << endl;

}

cout << findMedian(arr, n) << endl;

return 0;

}



19--- Input Format:

First line of input will take number of matrices n that you need to multiply.

For each line i in n, take two inputs which will represent dimensions aXb of matrix i.

Output Format:

Output will be the minimum number of operations that are required to multiply the list of

matrices.

Sample I/O Problem I:

Input: Output:

3 4500

10 30

305

5 60

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

int matrixChainOrder(const vector<pair<int, int>>& dimensions) {

int n = dimensions.size();

// Create a 2D table to store intermediate results

vector<vector<int>> dp(n, vector<int>(n, 0));

// Fill the table using dynamic programming

for (int len = 2; len < n; len++) {

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

int j = i + len - 1;

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

for (int k = i; k <= j - 1; k++) {

int cost = dp[i][k] + dp[k + 1][j] + dimensions[i - 1].first \* dimensions[k].second \* dimensions[j].second;

if (cost < dp[i][j]) {

dp[i][j] = cost;

}

}

}

}

// The result is stored at dp[1][n-1]

return dp[1][n - 1];

}

int main() {

int n;

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

cin >> n;

vector<pair<int, int>> dimensions(n);

cout << "Enter the dimensions (aXb) of each matrix:\n";

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

cout << "Matrix " << i + 1 << ": ";

cin >> dimensions[i].first >> dimensions[i].second;

}

int minOperations = matrixChainOrder(dimensions);

cout << "Minimum number of operations: " << minOperations << endl;

return 0;

}

Output-- Enter the number of matrices: 3

Enter the dimensions (aXb) of each matrix:

Matrix 1: 10 50

Matrix 2: 50 5

Matrix 3: 5 90

Minimum number of operations: 4500