

**1. Write a program to implement Quick sort algorithm for sorting a list of integers in ascending order.**

```
#include<iostream>
using namespace std;

int partition(int arr[], int s, int e)
{
    int pivot = arr[s];
    int count = 0;
    for(int i=s+1; i<=e; i++)
    {
        if(arr[i]<=pivot)
        {
            count++;
        }
    }

    int pivotIndex = s + count;
    swap(arr[s], arr[pivotIndex]);
    int i=s, j=e;
    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 s, int e)
{
    //base case
    if(s>=e)
```

```

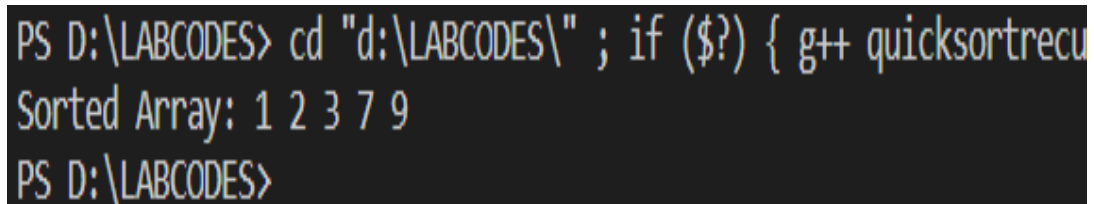
        return;

    int p= partition(arr,s,e);

    quickSort(arr,s,p-1);
    quickSort(arr,p+1,e);
}
int main()
{
    int arr[5]= {3,7,1,2,9};
    int n=5;
    quickSort(arr, 0 ,n-1);
    for(int i=0; i<n; i++)
    {
        cout<<arr[i]<<" ";
    }cout<<endl;
    return 0;
}

```

### OUTPUT:-



```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g++ quicksortrecu
Sorted Array: 1 2 3 7 9
PS D:\LABCODES>

```

**2. Write a program to implement Merge sort algorithm for sorting a list of integers in ascending order.**

```
#include<iostream>
using namespace std;
void merge(int *arr, int s, int e) {
    int mid = (s+e)/2;
    int len1 = mid - s + 1;
    int len2 = e - mid;
    int *first = new int[len1];
    int *second = new int[len2];
    int mainArrayIndex = s;
    for(int i=0; i<len1; i++) {
        first[i] = arr[mainArrayIndex++];
    }

    mainArrayIndex = mid+1;
    for(int i=0; i<len2; i++) {
        second[i] = arr[mainArrayIndex++];
    }

    //merge 2 sorted arrays
    int index1 = 0;
    int index2 = 0;
    mainArrayIndex = s;

    while(index1 < len1 && index2 < len2) {
        if(first[index1] < second[index2]) {
            arr[mainArrayIndex++] = first[index1++];
        }
        else{
            arr[mainArrayIndex++] = second[index2++];
        }
    }

    while(index1 < len1) {
        arr[mainArrayIndex++] = first[index1++];
    }

    while(index2 < len2 ) {
        arr[mainArrayIndex++] = second[index2++];
    }
}
```

```

    delete []first;
    delete []second;
}

void mergeSort(int *arr, int s, int e) {
    if(s >= e) {
        return;
    }

    int mid = (s+e)/2;
    mergeSort(arr, s, mid);
    mergeSort(arr, mid+1, e);
    merge(arr, s, e);
}

int main() {
    int arr[15] = {3,7,0,1,5,8,3,2,34,66,87,23,12,12,12};
    int n = 15;
    mergeSort(arr, 0, n-1);
    for(int i=0;i<n;i++){
        cout << arr[i] << " ";
    } cout << endl;
    return 0;
}

```

## OUTPUT:-

```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g++ me
Sorted Array: 0 1 2 3 3 5 7 8 12 12 12 23 34 66 87
PS D:\LABCODES>

```

**3. i) Write a program to implement the DFS algorithm for a graph.**

```
#include<iostream>
#include<vector>
#include<stack>

using namespace std;

class Graph {
    int V;
    vector<int> *adj;

public:
    Graph(int V);

    void addEdge(int v, int w);

    void DFSUtil(int v, vector<bool> &visited);

    void DFS(int v);
};

Graph::Graph(int V) {
    this->V = V;
    adj = new vector<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
}

void Graph::DFSUtil(int v, vector<bool> &visited) {
    visited[v] = true;
    cout << v << " ";

    vector<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i) {
        if (!visited[*i]) {
            DFSUtil(*i, visited);
        }
    }
}
```

```

void Graph::DFS(int v) {
    vector<bool> visited(V, false);

    DFSUtil(v, visited);
}

int main() {
    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 << "Depth First Traversal (start vertex 2) ";
    g.DFS(2);

    return 0;
}

```

**OUTPUT:-**

```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g+
Depth First Traversal (start vertex 2) 2 0 1 3
PS D:\LABCODES>

```

**ii) Write a program to implement the BFS algorithm for a graph.**

```
#include<iostream>
#include<list>

using namespace std;

class Graph {
    int V;
    list<int> *adj;

public:
    Graph(int V);

    void addEdge(int v, int w);

    void BFS(int s);
};

Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
}

void Graph::BFS(int s) {
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    list<int> queue;

    visited[s] = true;
    queue.push_back(s);

    while(!queue.empty()) {
        s = queue.front();
        cout << s << " ";
        queue.pop_front();
    }
}
```

```

        for (auto i = adj[s].begin(); i != adj[s].end(); ++i) {
            if (!visited[*i]) {
                queue.push_back(*i);
                visited[*i] = true;
            }
        }
    }
}

int main() {
    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 << "Breadth First Search (start vertex 2) ";
    g.BFS(2);

    return 0;
}

```

**OUTPUT:-**

```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g-
Breadth First Search (start vertex 2) 2 0 3 1
PS D:\LABCODES>

```



**4. Write a programs to implement backtracking algorithm for the N-queens problem.**

```
#include<iostream>
using namespace std;

bool isSafe(int board[], int row, int col, int n) {
    for (int i = 0; i < col; i++)
        if (board[i] == row || abs(board[i] - row) == abs(i - col))
            return false;
    return true;
}

bool solveNQUtil(int board[], int col, int n) {
    if (col >= n)
        return true;

    for (int i = 0; i < n; i++) {
        if (isSafe(board, i, col, n)) {
            board[col] = i;
            if (solveNQUtil(board, col + 1, n) == true)
                return true;
            board[col] = -1;
        }
    }
    return false;
}

void print(int board[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i] == j)
                cout << "Q ";
            else
                cout << "- ";
        }
        cout << "\n";
    }
}

bool solveNQ(int n) {
    int board[n];
    for (int i = 0; i < n; i++)
```

```

        board[i] = -1;

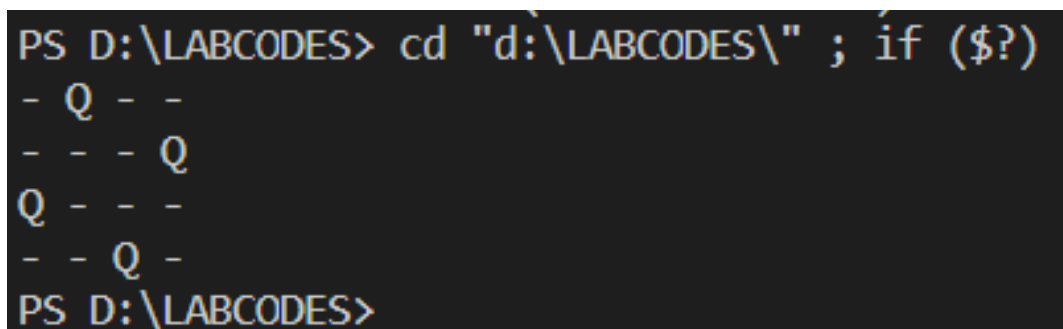
    if (solveNQUtil(board, 0, n) == false) {
        cout << "Solution does not exist";
        return false;
    }

    print(board, n);
    return true;
}

int main() {
    int n = 4;
    solveNQ(n);
    return 0;
}

```

**OUTPUT:-**



```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?)
- Q - -
- - - Q
Q - - -
- - Q -
PS D:\LABCODES>

```

**5. Write a program to implement the backtracking algorithm for the sum of subsets problem.**

```
#include<iostream>
#include<vector>
using namespace std;
bool isSafe(vector<int> &sums, int sum, int pos, int subset_sum) {
    if (pos == subset_sum) {
        return sum == 0;
    }
    if (pos > subset_sum) {
        return false;
    }
    if (sums[pos] == 0) {
        return isSafe(sums, sum - 1, pos + 1, subset_sum);
    }
    return isSafe(sums, sum - 1, pos + 1, subset_sum) ||
        isSafe(sums, sum + 1, pos + 1, subset_sum);
}
bool subsetSum(vector<int> &sums, int subset_sum) {
    return isSafe(sums, 0, 0, subset_sum);
}

int main() {
    vector<int> sums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int subset_sum = 15;
    if (subsetSum(sums, subset_sum)) {
        cout << "Found a subset with the sum equal to " << subset_sum << "." <<
endl;
    } else {
        cout << "No subset with the sum equal to " << subset_sum << " was found."
<< endl;
    }
    return 0;
}
```

**OUTPUT:-**

```
PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g++
No subset with the sum equal to 15 was found.
PS D:\LABCODES>
```

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

```
#include<iostream>
using namespace std;

bool graph[5][5] = {
    {0, 1, 1, 0, 0},
    {1, 0, 0, 1, 0},
    {1, 0, 0, 1, 0},
    {0, 1, 1, 0, 1},
    {0, 0, 0, 1, 0}
};

bool used[5];

bool checkVertex(int v) {
    used[v] = true;

    for (int i = 0; i < 5; i++) {
        if (graph[v][i] && !used[i]) {
            if (checkVertex(i)) {
                return true;
            }
        }
    }

    return false;
}

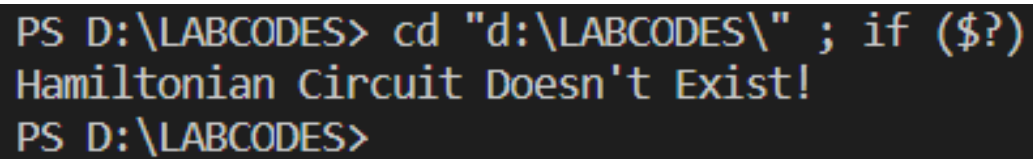
bool hamiltonianCircuit() {
    int startVertex = 0;

    for (int i = 0; i < 5; i++) {
        fill(used, used + 5, false);
        if (checkVertex(startVertex)) {
            return true;
        }
        startVertex++;
    }

    return false;
}
```

```
int main() {  
    if (hamiltonianCircuit()) {  
        cout << "Hamiltonian Circuit Exists!" << endl;  
    } else {  
        cout << "Hamiltonian Circuit Doesn't Exist!" << endl;  
    }  
  
    return 0;  
}
```

**OUTPUT:-**



```
PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?)  
Hamiltonian Circuit Doesn't Exist!  
PS D:\LABCODES>
```

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

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

// Structure for a job with a deadline and a profit.
struct Job {
    int id;        // Job Id
    int deadline;  // Deadline of the job
    int profit;    // Profit if the job is over before the deadline
};

// Comparison function to sort jobs based on their profit.
bool compare(Job a, Job b) {
    return a.profit > b.profit;
}

// Function to print the sequence of jobs selected by the greedy algorithm.
void printJobScheduling(vector<Job> jobs) {
    int size = jobs.size();
    for (int i = 0; i < size; i++) {
        cout << jobs[i].id << " ";
    }
    cout << "\n";
}

// Main function to implement the greedy algorithm for job sequencing with
// deadlines.
void jobScheduling(vector<Job> jobs) {
    // Sort jobs in decreasing order of profit.
    sort(jobs.begin(), jobs.end(), compare);

    int size = jobs.size();
    int maxDeadline = 0;
    for (int i = 0; i < size; i++) {
        maxDeadline = max(maxDeadline, jobs[i].deadline);
    }

    vector<bool> result(maxDeadline + 1, false);
```

```

vector<int> maxProfit(maxDeadline + 1, 0);

for (int i = 0; i < size; i++) {
    for (int j = min(maxDeadline, jobs[i].deadline); j >= 0; j--) {
        if (!result[j] && maxProfit[j] + jobs[i].profit >= maxProfit[j + 1]) {
            result[j] = true;
            maxProfit[j] += jobs[i].profit;
            break;
        }
    }
}

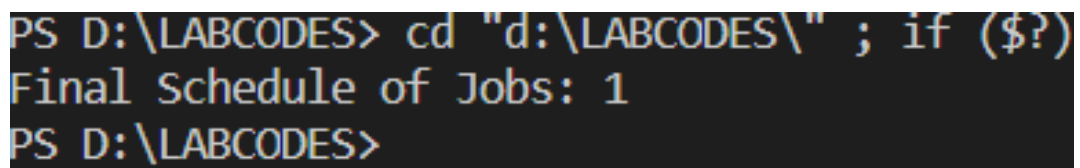
// Store the result.
vector<Job> jobSch;
for (int i = 0; i <= maxDeadline; i++) {
    if (result[i]) {
        jobSch.push_back(jobs[i]);
    }
}

// Print the final schedule of jobs.
cout << "Final Schedule of Jobs: ";
printJobScheduling(jobSch);
}

int main() {
    // Example usage of the jobScheduling function.
    vector<Job> jobs = {{1, 2, 100}, {2, 1, 50}, {3, 2, 200}, {4, 1, 70}};
    jobScheduling(jobs);
    return 0;
}

```

**OUTPUT:-**



```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) {
Final Schedule of Jobs: 1
PS D:\LABCODES>

```

**8. Write a program to implement Dijkstra's algorithm for the Single source shortest path problem.**

```
#include <bits/stdc++.h>
using namespace std;
#define INF 99999999

class Graph {
    int V; // number of vertices
    vector<pair<int, int>> *adj; // dynamic array of adjacency lists

public:
    Graph(int V); // Constructor
    void addEdge(int v, int w, int weight); // function to add an edge to the graph
    void shortestPath(int src); // prints shortest path from src to all other vertices
};

Graph::Graph(int V) {
    this->V = V;
    adj = new vector<pair<int, int>>[V];
}

void Graph::addEdge(int v, int w, int weight) {
    adj[v].push_back(make_pair(w, weight));
    adj[w].push_back(make_pair(v, weight));
}

void print(vector<int> dist) {
    cout << "Vertex Distance from Source\n";
    for (int i = 0; i < dist.size(); i++) {
        cout << i << " " << dist[i] << endl;
    }
}

void Graph::shortestPath(int src) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
    pq;

    vector<int> dist(V, INF);

    pq.push(make_pair(0, src));
    dist[src] = 0;
```



```

while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();

    for (auto x : adj[u]) {
        int v = x.first;
        int weight = x.second;

        if (dist[u] != INF && dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
            pq.push(make_pair(dist[v], v));
        }
    }
}

print(dist);
}

int main() {
    int V = 5;
    Graph g(V);

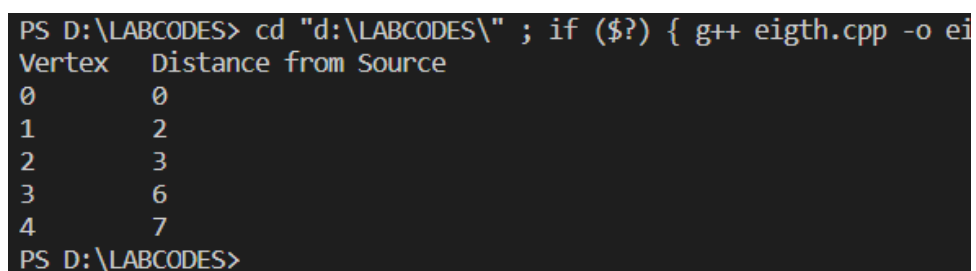
    g.addEdge(0, 1, 2);
    g.addEdge(0, 2, 4);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 3, 7);
    g.addEdge(2, 3, 3);
    g.addEdge(3, 4, 1);

    g.shortestPath(0);

    return 0;
}

```

## OUTPUT:-



```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g++ eigth.cpp -o ei
Vertex    Distance from Source
0         0
1         2
2         3
3         6
4         7
PS D:\LABCODES>

```

**9. Write a program that implements Prim's algorithm to generate minimum cost spanning tree.**

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <algorithm>
using namespace std;

const int INF = INT_MAX;
void printMST(vector<pair<int, int>> edges) {
    cout << "Edge \tWeight\n";
    for (const auto& edge : edges) {
        cout << edge.first << " - " << edge.second << "\t" << edge.first +
edge.second << "\n";
    }
}

void primMST(vector<vector<pair<int, int>>>& graph) {
    int V = graph.size();
    vector<pair<int, int>> edges;
    vector<int> key(V, INF);
    vector<bool> mstSet(V, false);

    key[0] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
pq;
    pq.push({0, 0});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        if (mstSet[u]) continue;

        mstSet[u] = true;

        for (const auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;

            if (!mstSet[v] && weight < key[v]) {
```

```

        key[v] = weight;
        pq.push({key[v], v});
        edges.push_back({u, v});
    }
}

printMST(edges);
}

int main() {
    int V; // Number of vertices
    cin >> V;
    vector<vector<pair<int, int>>> > graph(V);
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            int weight;
            cin >> weight;
            if (i != j) {
                graph[i].push_back({j, weight});
                graph[j].push_back({i, weight});
            }
        }
    }
    primMST(graph);

    return 0;
}

```

## OUTPUT:-

```

PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g++ ninth.cpp -o
4
0 2 4 1
2 0 3 0
4 3 0 5
1 0 5 0
Edge    weight
0 - 1    1
0 - 2    2
0 - 3    3
3 - 1    4
1 - 2    3
PS D:\LABCODES>

```

**10. Write a program that implements Kruskal's algorithm to generate minimum cost spanning tree.**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Edge {
    int src, dest, weight;
};

struct DisjointSet {
    vector<int> parent, rank;

    DisjointSet(int n) {
        parent.resize(n);
        rank.assign(n, 0);
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    int find(int x) {
        if (x != parent[x])
            parent[x] = find(parent[x]);
        return parent[x];
    }

    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY])
                parent[rootX] = rootY;
            else if (rank[rootX] > rank[rootY])
                parent[rootY] = rootX;
            else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};
```

```

bool compareEdges(const Edge &a, const Edge &b) {
    return a.weight < b.weight;
}

void kruskalMST(vector<Edge> &edges, int V) {
    sort(edges.begin(), edges.end(), compareEdges);

    vector<Edge> result;
    DisjointSet ds(V);

    for (const Edge &edge : edges) {
        int rootSrc = ds.find(edge.src);
        int rootDest = ds.find(edge.dest);

        if (rootSrc != rootDest) {
            result.push_back(edge);
            ds.unionSets(rootSrc, rootDest);
        }
        if (result.size() == V - 1)
            break;
    }

    cout << "Edges in the Minimum Spanning Tree:\n";
    for (const Edge &edge : result) {
        cout << edge.src << " - " << edge.dest << " (Weight: " << edge.weight <<
    "\n";
    }
}

int main() {
    int V, E; // Number of vertices and edges
    cin >> V >> E;

    vector<Edge> edges(E);

    cout << "Enter edges (source destination weight):\n";
    for (int i = 0; i < E; i++) {
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }

    kruskalMST(edges, V);

    return 0;
}

```

## OUTPUT:-

```
PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g++ tenth.cpp -o tenth } ; if ($?) { .\tenth }
5 7
Enter edges (source destination weight):
0 1 2
0 2 3
1 2 1
1 3 4
2 3 5
2 4 6
3 4 7
Edges in the Minimum Spanning Tree:
1 - 2 (Weight: 1)
0 - 1 (Weight: 2)
1 - 3 (Weight: 4)
2 - 4 (Weight: 6)
PS D:\LABCODES> █
```

**11. Write a program to implement Dynamic Programming algorithm for the 0/1 Knapsack.**

```
#include<iostream>
#include<vector>
using namespace std;

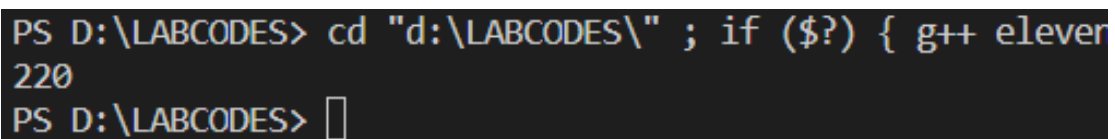
int knapSack(int W, int wt[], int val[], int n) {
    int i, w;
    int K[n + 1][W + 1];

    // Build K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(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[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    cout << knapSack(W, wt, val, n);
    return 0;
}
```

**OUTPUT:-**



```
PS D:\LABCODES> cd "d:\LABCODES\" ; if ($?) { g++ eleven 220
PS D:\LABCODES> 
```