BINARY SEARCH

```
//DAA ass 1
//binary search
#include<iostream>
using namespace std;

int main() {
    int A[] = {1, 2, 4, 6, 7, 9, 12};
    int target = 4;
    int r = 6; // The last index of the array
    int p = 0;

    while (p <= r) {
        int mid = p + (r - p) / 2;

        if (A[mid] == target) {
            cout << "Element found at index " << mid << endl;
            return 0; // Exit the program as the target is found
        }
        else if (A[mid] < target) {
            p = mid + 1;
        }
        else {
            r = mid - 1;
        }
    }

    cout << "Element not found in the array" << endl;
```

```
    return 0;
}
```

QUICK SORT

```
//DAA ass 1
//Quick_sort
#include<iostream>
using namespace std;

int partition(int A[], int p, int r) {
    int pivot = A[r];
    int i = p - 1;

    for (int j = p; j <= r - 1; j++) {
        if (A[j] <= pivot) {
            i++;
            swap(A[i], A[j]);
        }
    }

    swap(A[i + 1], A[r]);
    return i + 1;
}

void Quicksort(int A[], int p, int r) {
    if (p < r) {
        int q = partition(A, p, r);
        Quicksort(A, p, q - 1);
```

```
        Quicksort(A, q + 1, r);
    }
}

int main() {
    int A[] = {2, 8, 7, 1, 3, 5, 6, 4};
    int start = 0;  // corrected start index
    int end = 7;    // corrected end index
    int n = sizeof(A) / sizeof(A[0]);

    Quicksort(A, start, end);

    // Print the sorted array
    for (int i = 0; i < n; i++) {
        cout << A[i] << " ";
    }

    return 0;
}
```

MERGE SORT

```
//DAA ass
//merge sort
#include<iostream>
using namespace std;

void Mergesort(int A[], int p, int q, int r) {
```

```
    int n1 = q - p + 1;
    int n2 = r - q;
    int L[n1 + 1];
    int R[n2 + 1];

    for (int i = 1; i <= n1; i++) {
        L[i] = A[p + i - 1];
    }

    for (int j = 1; j <= n2; j++) {
        R[j] = A[q + j];
    }

    L[n1 + 1] = 39876;
    R[n2 + 1] = 39876;

    int i = 1;
    int j = 1;

    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) { // Fixed the condition here
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

```
void mergesort(int A[], int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;
        mergesort(A, p, q);
        mergesort(A, q + 1, r);
        Mergesort(A, p, q, r);
    }
}

int main() {
    int A[] = {6, 4, 2, 1, 9, 8, 3, 5};
    mergesort(A, 0, 7);

    for (int i = 0; i < 8; i++) {
        cout << A[i] << " ";
    }

    return 0;
}
```

MAX SUM SUBARRAY

```
//DAA ass
//maximum subarray
#include <iostream>
using namespace std;
```

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

int max(int a, int b, int c) {
    return max(max(a, b), c);
}

int maxCrossingSum(int arr[], int l, int m, int h) {
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--) {
        sum = sum + arr[i];
        if (sum > left_sum)
            left_sum = sum;
    }

    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m; i <= h; i++) {
        sum = sum + arr[i];
        if (sum > right_sum)
            right_sum = sum;
    }

    return max(left_sum + right_sum - arr[m], left_sum, right_sum);
}

int maxSubArraySum(int arr[], int l, int h) {
```

```
    if (l > h)
        return INT_MIN;
    if (l == h)
        return arr[l];

    int m = (l + h) / 2;

    return max(maxSubArraySum(arr, l, m - 1),
               maxSubArraySum(arr, m + 1, h),
               maxCrossingSum(arr, l, m, h));
}

int main() {
    int arr[] = {2, 3, 4, 5, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    int max_sum = maxSubArraySum(arr, 0, n - 1);
    cout << "Maximum contiguous sum is " << max_sum;
    return 0;
}
```

ACTIVITY SELECTION

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

vector<int> activity_selection(int s[], int f[], int n) {
    vector<int> A;
```

```
    A.push_back(0);
    int k = 0;

    for (int m = 1; m < n; m++) {
        if (s[m] >= f[k]) {
            A.push_back(m);
            k = m;
        }
    }

    return A;
}

int main() {
    int s[] = {1, 3, 0, 5, 8, 5};
    int f[] = {2, 4, 6, 7, 9, 9}; // should be in sorted order corresponding to starting

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

    vector<int> result = activity_selection(s, f, n);

    cout << "Selected activities: ";
    for (int i = 0; i < result.size(); i++) {
        cout << result[i] << " ";
    }

    return 0;
}
```

JOB SEQUENCING

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

using namespace std;

struct Job {
    char id;
    int dead;
    int profit;
};

class JobScheduler {
private:
    static bool comparison(Job a, Job b) {
        return (a.profit > b.profit);
    }

public:
    static void scheduleJobs(Job arr[], int n) {
        sort(arr, arr + n, comparison);

        int result[n];
        bool slot[n];

        for (int i = 0; i < n; i++)
            slot[i] = false;
```

```
        for (int i = 0; i < n; i++) {
            for (int j = min(n, arr[i].dead) - 1; j >= 0; j--) {
                if (slot[j] == false) {
                    result[j] = i;
                    slot[j] = true;
                    break;
                }
            }
        }

        for (int i = 0; i < n; i++)
            if (slot[i])
                cout << arr[result[i]].id << " ";
    }
};

int main() {
    Job arr[] = { { 'a', 2, 100 },
                  { 'b', 1, 19 },
                  { 'c', 2, 27 },
                  { 'd', 1, 25 },
                  { 'e', 3, 15 } };

    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum profit sequence of jobs: ";

    JobScheduler::scheduleJobs(arr, n);
    return 0;
}
```

KNAPSACK :

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

using namespace std;

class Item {
public:
    int profit;
    int weight;
    double ratio;
};

bool compareItems(const Item& item1, const Item& item2) {
    return item1.ratio > item2.ratio;
}

double fractionalKnapsack(vector<Item>& items, int capacity, int n) {
    for (int i = 0; i < n; i++) {
        items[i].ratio = static_cast<double>(items[i].profit) / items[i].weight;
    }

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

    double totalValue = 0.0;
    int remainingCapacity = capacity;
```

```
    for (int i = 0; i < n; i++) {
        if (items[i].weight <= remainingCapacity) {
            totalValue += items[i].profit;
            remainingCapacity -= items[i].weight;
        } else {
            totalValue += items[i].ratio * remainingCapacity;
            break;
        }
    }

    return totalValue;
}

int main() {
    int numItems;
    cout << "Enter the number of items: ";
    cin >> numItems;

    vector<Item> items(numItems);
    for (int i = 0; i < numItems; ++i) {
        cout << "Enter weight and profit for item " << i + 1 << ": ";
        cin >> items[i].weight >> items[i].profit;
        items[i].ratio = 0.0;
    }

    int capacity;
    cout << "Enter the knapsack capacity: ";
    cin >> capacity;
```

```
    double maxValue = fractionalKnapsack(items, capacity, numItems);
    cout << "Maximum value in the knapsack: " << maxValue << endl;

    return 0;
}
```

HUFFMAN CODING

LONGEST COMMON SUBSEQUENCE

```
// A Naive recursive implementation of LCS problem

#include <bits/stdc++.h>
using namespace std;

// Returns length of LCS for X[0..m-1], Y[0..n-1]
int lcs(string X, string Y, int m, int n)
{
	if (m == 0 || n == 0)
		return 0;
	if (X[m - 1] == Y[n - 1])
		return 1 + lcs(X, Y, m - 1, n - 1);
	else
		return max(lcs(X, Y, m, n - 1),
				lcs(X, Y, m - 1, n));
}

// Driver code
```

```
int main()
{
	string S1 = "AGGTAB";
	string S2 = "GXTXAYB";
	int m = S1.size();
	int n = S2.size();

	cout << "Length of LCS is " << lcs(S1, S2, m, n);

	return 0;
}
```

COIN EXCHANGE METHOD

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

using namespace std;

long getNumberOfWays(long N, vector<long> Coins) {
   vector<long> ways(N + 1);
   ways[0] = 1;

   for(int i = 0; i < Coins.size(); i++) {
      for(int j = 0; j < ways.size(); j++) {
         if (Coins[i] <= j) {
            ways[j] += ways[(j - Coins[i])];
         }
      }
   }
```

```
    return ways[N];
}

void printArray(vector<long> coins) {
    for(long i = 0; i < coins.size(); ++i)
        cout << coins[i] << "\n";
}

int main() {
    vector<long> Coins;
    Coins.push_back(1);
    Coins.push_back(5);
    Coins.push_back(10);

    cout << "The Coins Array:" << endl;
    printArray(Coins);

    cout << "Solution:" << endl;
    cout << getNumberOfWays(12, Coins) << endl;

    return 0;
}
```

MATRIX CHAIN MULTIPLICATION

```
// C++ code to implement the
// matrix chain multiplication using recursion
```

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

int MatrixChainOrder(int p[], int i, int j)
{
	if (i == j)
		return 0;
	int k;
	int mini = INT_MAX;
	int count;

	for (k = i; k < j; k++)
	{
		count = MatrixChainOrder(p, i, k)
				+ MatrixChainOrder(p, k + 1, j)
				+ p[i - 1] * p[k] * p[j];

		mini = min(count, mini);
	}

	return mini;
}


int main()
{
	int arr[] = { 1, 2, 3, 4, 3 };
	int N = sizeof(arr) / sizeof(arr[0]);
```

```
    cout << "Minimum number of multiplications is "
        << MatrixChainOrder(arr, 1, N - 1);
    return 0;
}
```

0/1 knapsack

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

int max(int a, int b) { return (a > b) ? a : b; }

int knapSack(int W, int wt[], int val[], int n, vector<int> &pickedItems)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1, pickedItems);
    else
    {
        int include = val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1, pickedItems);
        int exclude = knapSack(W, wt, val, n - 1, pickedItems);

        if (include > exclude) // if including the item gives more value
            pickedItems.push_back(n - 1); // add this item to picked items

        return max(include, exclude);
```

```
	}
}

int main()
{
	int profit[] = { 1,2,5,6 };
	int weight[] = {2,3,4,5 };
	int W = 8;
	int n = sizeof(profit) / sizeof(profit[0]);
	vector<int> pickedItems;

	cout << "Maximum profit: " << knapSack(W, weight, profit, n, pickedItems) << endl;

	return 0;
}
```

OBST

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

int sum(int freq[], int i, int j);

int optCost(int freq[], int i, int j)
{
	if (j < i)
		return 0;
	if (j == i)
		return freq[i];
```

```
	int fsum = sum(freq, i, j);

	int min = INT_MAX;

	for (int r = i; r <= j; ++r)
	{
		int cost = optCost(freq, i, r - 1) +
				optCost(freq, r + 1, j);
		if (cost < min)
			min = cost;
	}

	// Return minimum value
	return min + fsum;
}



int optimalSearchTree(int keys[],
					int freq[], int n)
{

	return optCost(freq, 0, n - 1);
}

int sum(int freq[], int i, int j)
{
	int s = 0;
	for (int k = i; k <= j; k++)
```

```
        s += freq[k];
        return s;
}

// Driver Code
int main()
{
        int keys[] = {10, 12, 20};
        int freq[] = {34, 8, 50};
        int n = sizeof(keys) / sizeof(keys[0]);
        cout << "Cost of Optimal BST is "
                << optimalSearchTree(keys, freq, n);
        return 0;
}
```

N QUEEN

```
#include <bits/stdc++.h>
#define N 4
using namespace std;

void printSolution(int board[N][N])
{
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++)
                if(board[i][j])
                        cout << "Q ";
                else cout<<". ";
```

```
		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 solveNQUtil(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 (solveNQUtil(board, col + 1))
				return true;

			board[i][col] = 0;
		}
	}

	return false;
}

bool solveNQ()
{
	int board[N][N] = { { 0, 0, 0, 0 },
						{ 0, 0, 0, 0 },
						{ 0, 0, 0, 0 },
						{ 0, 0, 0, 0 } };

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

	printSolution(board);
```

```
        return true;
}

int main()
{
        solveNQ();
        return 0;
}
```

SUM OF SUBSETS

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


bool flag = 0;
void PrintSubsetSum(int i, int n, int set[], int targetSum,
                                        vector<int>& subset)
{

        if (targetSum == 0) {
                flag = 1;
                cout << "[ ";
                for (int i = 0; i < subset.size(); i++) {
                        cout << subset[i] << " ";
                }
                cout << "]";
                return;
        }
```

```
    if (i == n) {
        return;
    }

    PrintSubsetSum(i + 1, n, set, targetSum, subset);

    if (set[i] <= targetSum) {

        subset.push_back(set[i]);

        PrintSubsetSum(i + 1, n, set, targetSum - set[i],subset);

        subset.pop_back();
    }
}


int main()
{
    // Test case 1
    int set[] = { 1, 2, 1 };
    int sum = 3;
    int n = sizeof(set) / sizeof(set[0]);
    vector<int> subset;
    cout << "Output 1:" << endl;
    PrintSubsetSum(0, n, set, sum, subset);
    cout << endl;
    if (!flag) {
```

```
            cout << "There is no such subset";
        }

        return 0;
}
```

M COLORING

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

#define V 4

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] = {0};
    if (!graphColoringUtil(graph, m, color, 0)) {
        cout << "Solution does not exist" << endl;
        return false;
    }
    cout << "Solution Exists: Following are the assigned colors" << endl;
    for (int i = 0; i < V; i++)
        cout << color[i] << " ";
    cout << endl;
    return true;
}

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;
```

```
}
```

EULERIAN PATH

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

using namespace std;

class Graph {
public:
   vector<vector<int> > adjList; // Modified declaration
   int V;

   Graph(int V) : V(V), adjList(V) {}

   void addEdge(int u, int v) {
      adjList[u].push_back(v);
      adjList[v].push_back(u);
   }

   bool hasEulerianPath() {
      int oddDegreeCount = 0;
      int startVertex = 0;

      // Check for odd degree vertices
      for (int i = 0; i < V; i++) {
         if (adjList[i].size() % 2 == 1) {
            oddDegreeCount++;
```

```
            startVertex = i; // Update startVertex to odd degree vertex
        }
    }

    if (oddDegreeCount != 0 && oddDegreeCount != 2) // Eulerian path requires 0 or 2 odd
degree vertices
        return false;

    vector<bool> visited(V, false);
    dfs(startVertex, visited);

    // Check for connectivity
    for (int i = 0; i < V; ++i) {
        if (!visited[i] && adjList[i].size() > 0) // Check for unreachable vertices
            return false;
    }

    return true; // Eulerian path or circuit exists
  }

  void dfs(int v, vector<bool>& visited) {
    visited[v] = true;
    for (size_t j = 0; j < adjList[v].size(); j++) {
        int neighbor = adjList[v][j];
        if (!visited[neighbor])
            dfs(neighbor, visited);
    }
  }
};
```

```
int main() {
    Graph g3(5);
            g3.addEdge(1, 0);
            g3.addEdge(0, 2);
            g3.addEdge(2, 1);
            g3.addEdge(0, 3);
            g3.addEdge(3, 4);
            g3.addEdge(1, 3);

    if (g3.hasEulerianPath())
        cout << "The graph has an Eulerian path or circuit" << endl;
    else
        cout << "The graph does not have an Eulerian path or circuit" << endl;

    return 0;
}
```

HAMILTONIAN PATH :

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

using namespace std;

class Graph {
public:
    vector<vector<int> > adjList; // Modified declaration
    int V;
```

```
Graph(int V) : V(V), adjList(V) {}

void addEdge(int u, int v) {
    adjList[u].push_back(v);
    adjList[v].push_back(u);
}

bool hasHamiltonianPath() {
    vector<bool> visited(V, false);

    // Start DFS traversal from each vertex
    for (int i = 0; i < V; ++i) {
        visited.assign(V, false); // Reset visited array
        if (dfs(i, visited, 0)) // If a Hamiltonian path is found starting from vertex i
            return true;
    }

    return false; // No Hamiltonian path found
}

bool dfs(int v, vector<bool>& visited, int count) {
    visited[v] = true;
    ++count;

    if (count == V) // All vertices visited
        return true;

    for (size_t j = 0; j < adjList[v].size(); j++) {
        int neighbor = adjList[v][j];
```

```
            if (!visited[neighbor]) {
                if (dfs(neighbor, visited, count)) // Recursively visit adjacent vertices
                    return true;
            }
        }

        // Backtrack
        visited[v] = false;
        return false;
    }
};

int main() {
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);

    if (g1.hasHamiltonianPath())
        cout << "The graph has a Hamiltonian path" << endl;
    else
        cout << "The graph does not have a Hamiltonian path" << endl;

    return 0;
}
```

FORD FULKERSON :

```
// Ford-Fulkerson algorith in C++

#include <limits.h>
#include <string.h>

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

#define V 6

// Using BFS as a searching algorithm
bool bfs(int rGraph[V][V], int s, int t, int parent[]) {
  bool visited[V];
  memset(visited, 0, sizeof(visited));

  queue<int> q;
  q.push(s);
  visited[s] = true;
  parent[s] = -1;

  while (!q.empty()) {
    int u = q.front();
    q.pop();

    for (int v = 0; v < V; v++) {
      if (visited[v] == false && rGraph[u][v] > 0) {
        q.push(v);
        parent[v] = u;
```

```
      visited[v] = true;
    }
  }
 }

 return (visited[t] == true);
}

// Applying fordfulkerson algorithm
int fordFulkerson(int graph[V][V], int s, int t) {
 int u, v;

 int rGraph[V][V];
 for (u = 0; u < V; u++)
  for (v = 0; v < V; v++)
   rGraph[u][v] = graph[u][v];

 int parent[V];
 int max_flow = 0;

 // Updating the residual values of edges
 while (bfs(rGraph, s, t, parent)) {
  int path_flow = INT_MAX;
  for (v = t; v != s; v = parent[v]) {
   u = parent[v];
   path_flow = min(path_flow, rGraph[u][v]);
  }

  for (v = t; v != s; v = parent[v]) {
```

```
      u = parent[v];
      rGraph[u][v] -= path_flow;
      rGraph[v][u] += path_flow;
    }

    // Adding the path flows
    max_flow += path_flow;
  }

  return max_flow;
}

int main() {
  int graph[V][V] = {{0, 8, 0, 0, 3, 0},
            {0, 0, 9, 0, 0, 0},
            {0, 0, 0, 0, 7, 2},
            {0, 0, 0, 0, 0, 5},
            {0, 0, 7, 4, 0, 0},
            {0, 0, 0, 0, 0, 0}};

  cout << "Max Flow: " << fordFulkerson(graph, 0, 5) << endl;
}
```

KAHNS ALGO:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;
```

```
class Graph {
public:
    int V; // Number of vertices
    vector<vector<int>> adjList; // Adjacency list

    // Constructor
    Graph(int V) : V(V), adjList(V) {}

    // Function to add an edge to the graph
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
    }

    // Function to perform topological sorting using Kahn's algorithm
    vector<int> topologicalSort() {
        vector<int> inDegree(V, 0); // Initialize in-degree of all vertices to 0
        for (int u = 0; u < V; ++u) {
            for (int v : adjList[u]) {
                inDegree[v]++; // Increment in-degree of adjacent vertices
            }
        }

        queue<int> q;
        for (int u = 0; u < V; ++u) {
            if (inDegree[u] == 0) {
                q.push(u); // Enqueue vertices with in-degree 0
            }
        }
```

```
        vector<int> result;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            result.push_back(u); // Add vertex to the result

            // Decrease in-degree of adjacent vertices and enqueue if in-degree becomes 0
            for (int v : adjList[u]) {
                if (--inDegree[v] == 0) {
                    q.push(v);
                }
            }
        }

        // If result size is less than V, there is a cycle in the graph
        if (result.size() < V) {
            cout << "Graph contains a cycle!" << endl;
            return {};
        }

        return result;
    }
};

int main() {
    // Create a graph
    Graph g(6);
    g.addEdge(5, 2);
```

```
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    // Perform topological sorting
    vector<int> result = g.topologicalSort();

    // Print the sorted order
    cout << "Topological sorting order: ";
    for (int vertex : result) {
        cout << vertex << " ";
    }
    cout << endl;

    return 0;
}
```

TOPOLOGICAL SORTING BY DFS:

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

using namespace std;

class Graph {
public:
```

```
int V; // Number of vertices
vector<vector<int>> adjList; // Adjacency list

// Constructor
Graph(int V) : V(V), adjList(V) {}

// Function to add an edge to the graph
void addEdge(int u, int v) {
    adjList[u].push_back(v);
}

// DFS function for topological sorting
void dfsTopologicalSort(int v, vector<bool>& visited, stack<int>& stack) {
    visited[v] = true; // Mark current vertex as visited

    // Recur for all adjacent vertices
    for (int adj : adjList[v]) {
        if (!visited[adj]) {
            dfsTopologicalSort(adj, visited, stack);
        }
    }

    // Push current vertex to the stack after visiting all its adjacent vertices
    stack.push(v);
}

// Function to perform topological sorting using DFS
vector<int> topologicalSort() {
    vector<bool> visited(V, false); // Initialize visited array
```

```
        stack<int> stack; // Stack to store the topological order

        // Perform DFS for each vertex
        for (int i = 0; i < V; ++i) {
            if (!visited[i]) {
                dfsTopologicalSort(i, visited, stack);
            }
        }

        // Create the topological order by popping elements from the stack
        vector<int> result;
        while (!stack.empty()) {
            result.push_back(stack.top());
            stack.pop();
        }

        return result;
    }
};

int main() {
    // Create a graph
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);
```

```
    // Perform topological sorting
    vector<int> result = g.topologicalSort();

    // Print the sorted order
    cout << "Topological sorting order: ";
    for (int vertex : result) {
        cout << vertex << " ";
    }
    cout << endl;

    return 0;
}
```