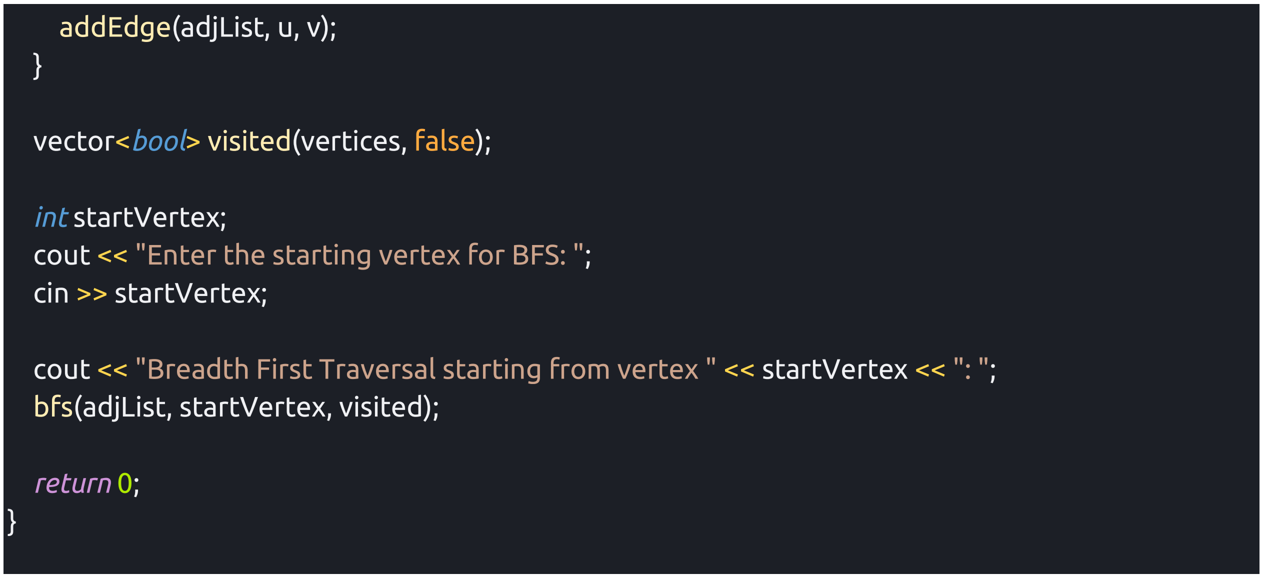
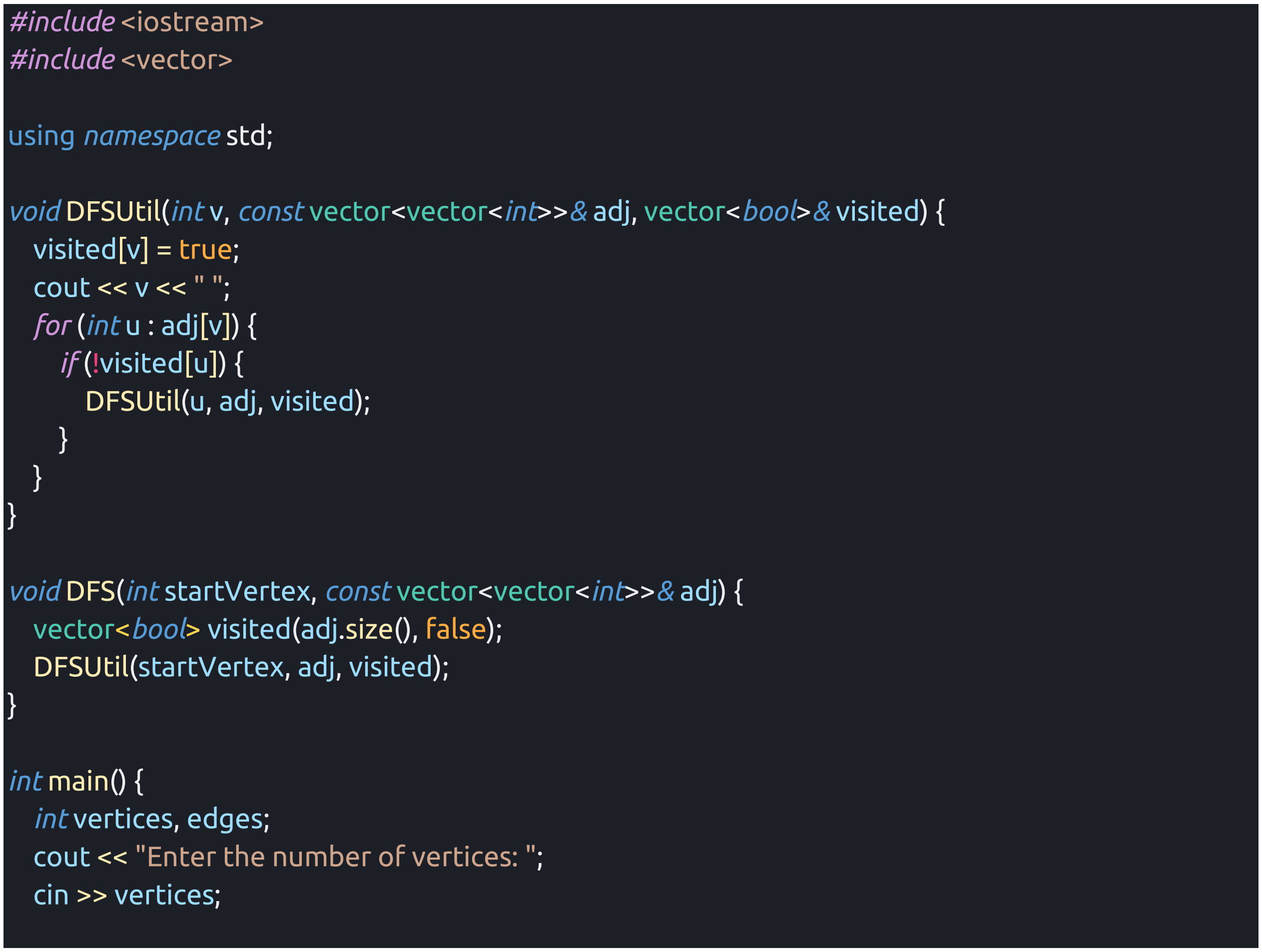
BFS:

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

using namespace std;

void bfs(vector<vector<int>>& adjList, int startNode, vector<bool>& visited) {
    queue<int> q;
    visited[startNode] = true;
    q.push(startNode);

    while (!q.empty()) {
        int currentNode = q.front();
        q.pop();
        cout << currentNode << " ";

        for (int neighbor : adjList[currentNode]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

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

int main() {
    int vertices, edges;
    cout << "Enter the number of vertices: ";
    cin >> vertices;

    cout << "Enter the number of edges: ";
    cin >> edges;

    vector<vector<int>> adjList(vertices);

    cout << "Enter the edges (source and destination pairs):" << endl;
    for (int i = 0; i < edges; ++i) {
        int u, v;
        cin >> u >> v;
```



```
        addEdge(adjList, u, v);
    }

    vector<bool> visited(vertices, false);

    int startVertex;
    cout << "Enter the starting vertex for BFS: ";
    cin >> startVertex;

    cout << "Breadth First Traversal starting from vertex " << startVertex << ": ";
    bfs(adjList, startVertex, visited);

    return 0;
}
```

DFS:



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

using namespace std;

void DFSUtil(int v, const vector<vector<int>>& adj, vector<bool>& visited) {
    visited[v] = true;
    cout << v << " ";
    for (int u : adj[v]) {
        if (!visited[u]) {
            DFSUtil(u, adj, visited);
        }
    }
}

void DFS(int startVertex, const vector<vector<int>>& adj) {
    vector<bool> visited(adj.size(), false);
    DFSUtil(startVertex, adj, visited);
}

int main() {
    int vertices, edges;
    cout << "Enter the number of vertices: ";
    cin >> vertices;
    vector<vector<int>> adj(vertices);

    cout << "Enter the number of edges: ";
    cin >> edges;

    cout << "Enter the edges (source and destination pairs):" << endl;
    for (int i = 0; i < edges; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    int startVertex;
    cout << "Enter the starting vertex for DFS: ";
    cin >> startVertex;

    cout << "Depth First Traversal starting from vertex " << startVertex << ": ";
    DFS(startVertex, adj);

    return 0;
}
```

Maximum Subarray:

```
#include <iostream>
#include <climits>

int maxSubArraySum(int arr[], int size, int &start, int &end) {
    int max_so_far = INT_MIN;
    int max_ending_here = 0;

    start = 0;
    end = 0;
    int s = 0;  // Temporary start index

    for (int i = 0; i < size; i++) {
        max_ending_here += arr[i];

        if (max_so_far < max_ending_here) {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }

        if (max_ending_here < 0) {
            max_ending_here = 0;
            s = i + 1;  // Update temporary start index
        }
    }
    return max_so_far;
}

int main() {
    const int SIZE = 9;
    int arr[SIZE] = {-2, 1, -3, 4, -1, 2, 1, -5, 4};

    int start, end;
    int max_sum = maxSubArraySum(arr, SIZE, start, end);

    std::cout << "Maximum subarray sum is " << max_sum << std::endl;
    std::cout << "The subarray is: [";
    for (int i = start; i <= end; i++) {
        std::cout << arr[i];
        if (i < end) {
            std::cout << ", ";
        }
    }
    std::cout << "]" << std::endl;

    return 0;
}
```

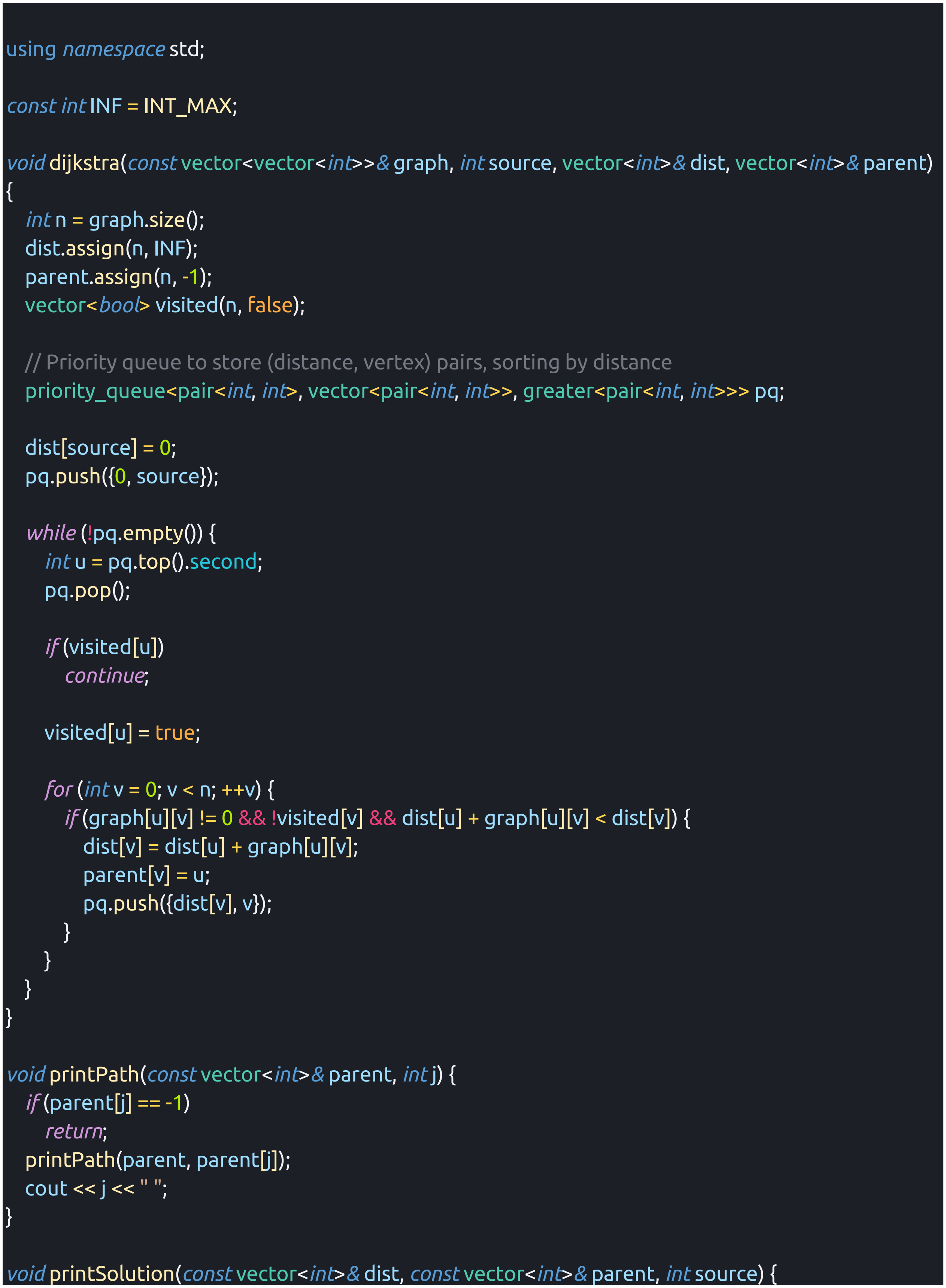
User Input:

Maximum subarray sum is 6

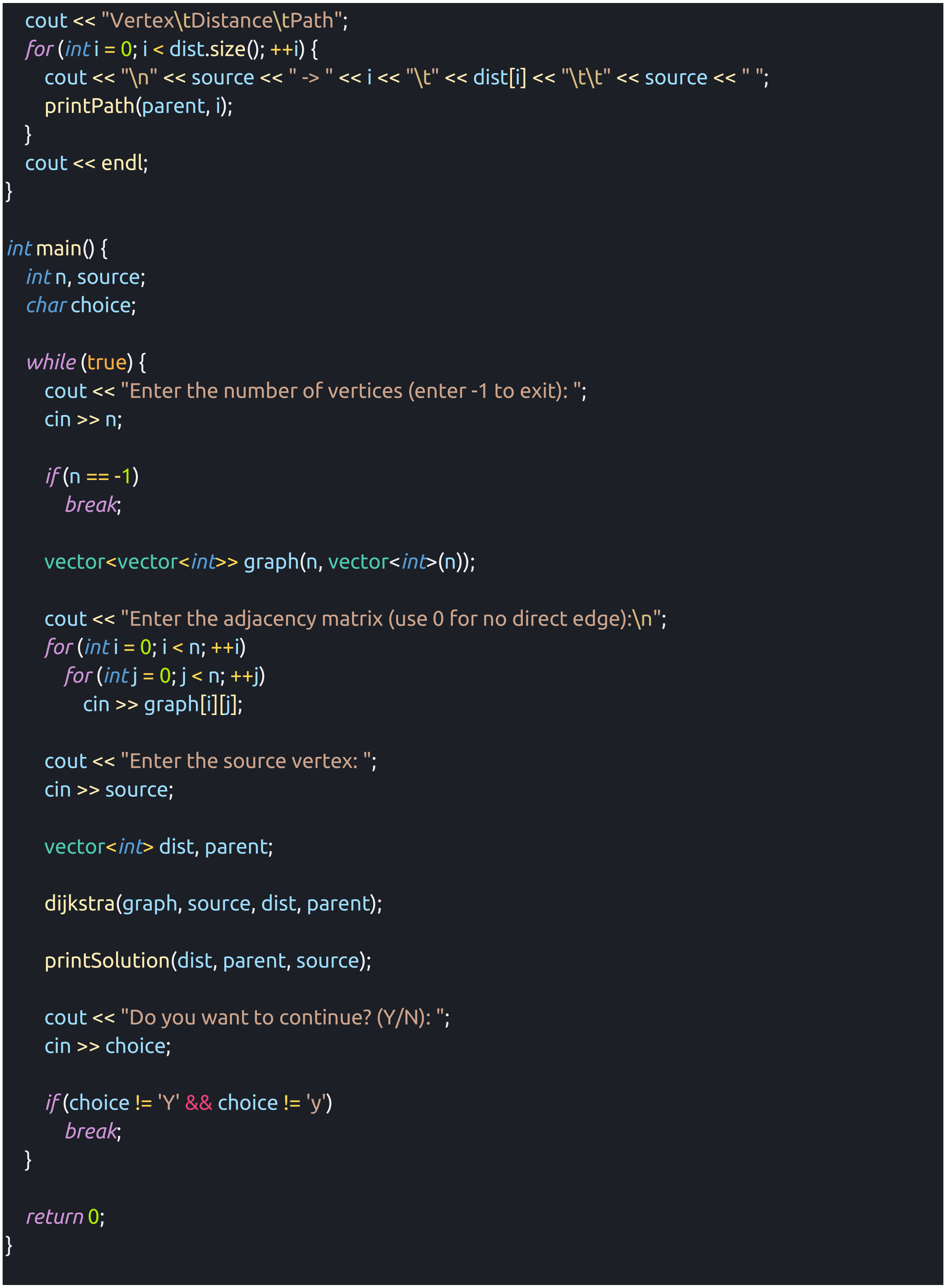
The subarray is: [4, -1, 2, 1]

Dijkstra:

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



```
using namespace std;

const int INF = INT_MAX;

void dijkstra(const vector<vector<int>>& graph, int source, vector<int>& dist, vector<int>& parent)
{
    int n = graph.size();
    dist.assign(n, INF);
    parent.assign(n, -1);
    vector<bool> visited(n, false);

    // Priority queue to store (distance, vertex) pairs, sorting by distance
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    dist[source] = 0;
    pq.push({0, source});

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

        if (visited[u])
            continue;

        visited[u] = true;

        for (int v = 0; v < n; ++v) {
            if (graph[u][v] != 0 && !visited[v] && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
                parent[v] = u;
                pq.push({dist[v], v});
            }
        }
    }
}

void printPath(const vector<int>& parent, int j) {
    if (parent[j] == -1)
        return;
    printPath(parent, parent[j]);
    cout << j << " ";
}

void printSolution(const vector<int>& dist, const vector<int>& parent, int source) {
```



```cpp
    cout << "Vertex\tDistance\tPath";
    for (int i = 0; i < dist.size(); ++i) {
        cout << "\n" << source << " -> " << i << "\t" << dist[i] << "\t\t" << source << " ";
        printPath(parent, i);
    }
    cout << endl;
}

int main() {
    int n, source;
    char choice;

    while (true) {
        cout << "Enter the number of vertices (enter -1 to exit): ";
        cin >> n;

        if (n == -1)
            break;

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

        cout << "Enter the adjacency matrix (use 0 for no direct edge):\n";
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                cin >> graph[i][j];

        cout << "Enter the source vertex: ";
        cin >> source;

        vector<int> dist, parent;

        dijkstra(graph, source, dist, parent);

        printSolution(dist, parent, source);

        cout << "Do you want to continue? (Y/N): ";
        cin >> choice;

        if (choice != 'Y' && choice != 'y')
            break;
    }

    return 0;
}
```

User Input:

Enter the number of vertices: 5

Enter the adjacency matrix (use 0 for no direct edge):

0 10 0 30 100

10 0 50 0 0

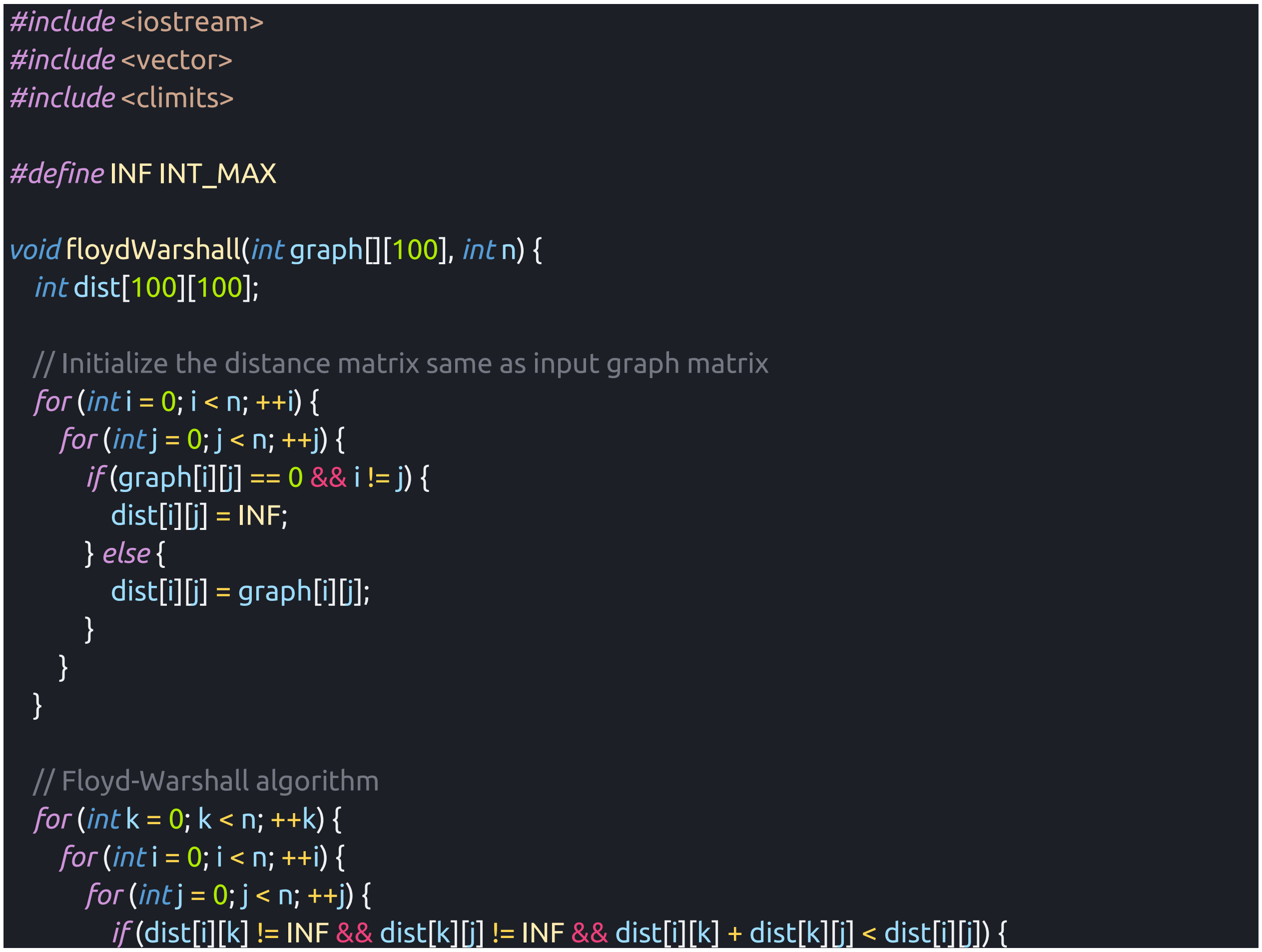
0 50 0 20 10

30 0 20 0 60

100 0 10 60 0

Enter the source vertex: 0

Floyd Warshal:



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

#define INF INT_MAX

void floydWarshall(int graph[][100], int n) {
  int dist[100][100];

  // Initialize the distance matrix same as input graph matrix
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if (graph[i][j] == 0 && i != j) {
        dist[i][j] = INF;
      } else {
        dist[i][j] = graph[i][j];
      }
    }
  }

  // Floyd-Warshall algorithm
  for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
```

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

    // Print the shortest distance matrix
    std::cout << "Shortest distances between every pair of vertices:\n";
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (dist[i][j] == INF) {
                std::cout << "INF ";
            } else {
                std::cout << dist[i][j] << " ";
            }
        }
        std::cout << "\n";
    }
}

int main() {
    int n;
    std::cout << "Enter the number of vertices: ";
    std::cin >> n;

    int graph[100][100];
    std::cout << "Enter the adjacency matrix (use 0 for no edge and input 0 for diagonal
elements):\n";
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cin >> graph[i][j];
        }
    }

    floydWarshall(graph, n);

    return 0;
}
```

User Input:

Enter the number of vertices:

4

Enter the adjacency matrix (use 0 for no edge and input 0 for diagonal elements):

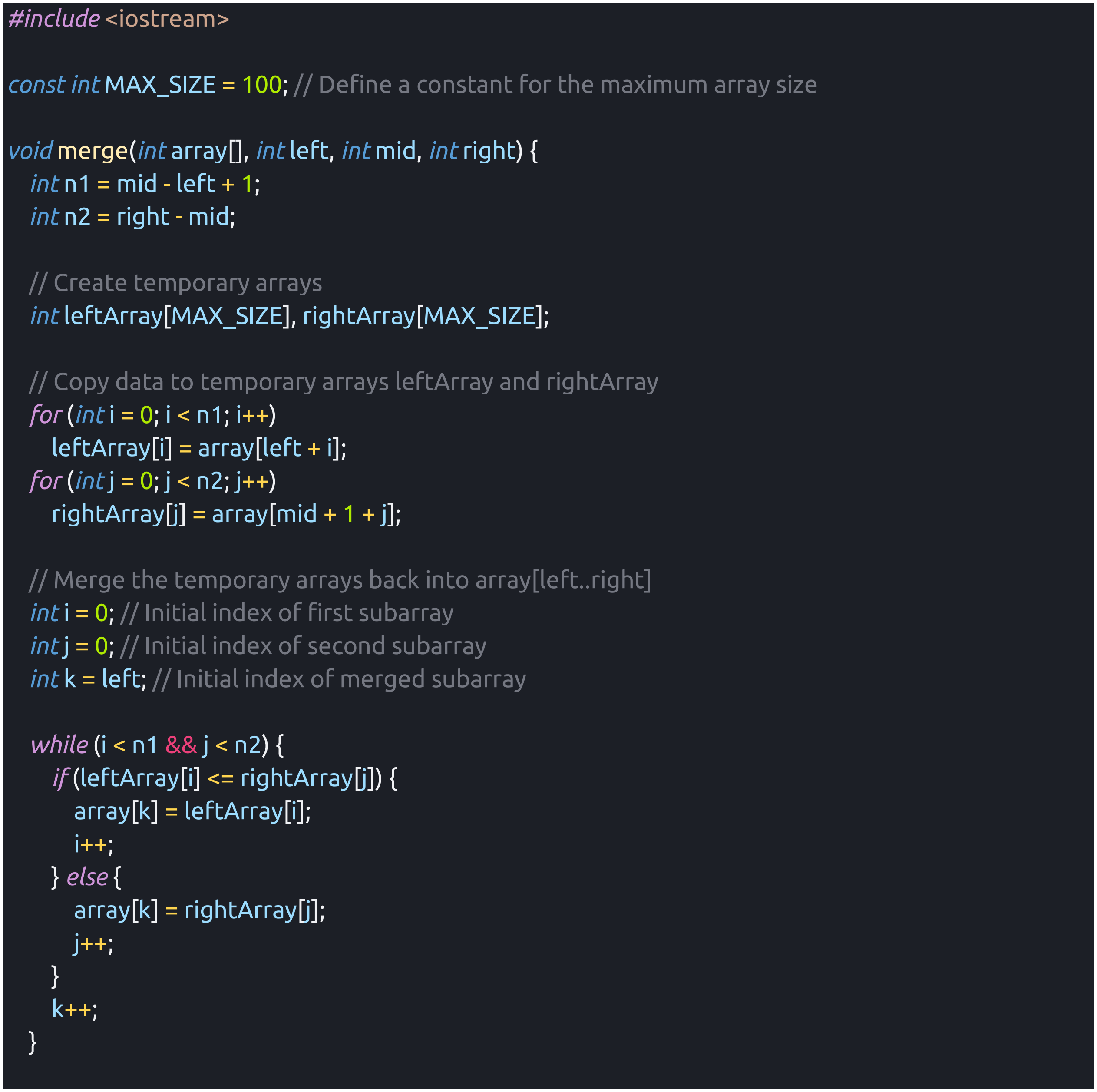
0 3 INF 5

2 0 INF 4

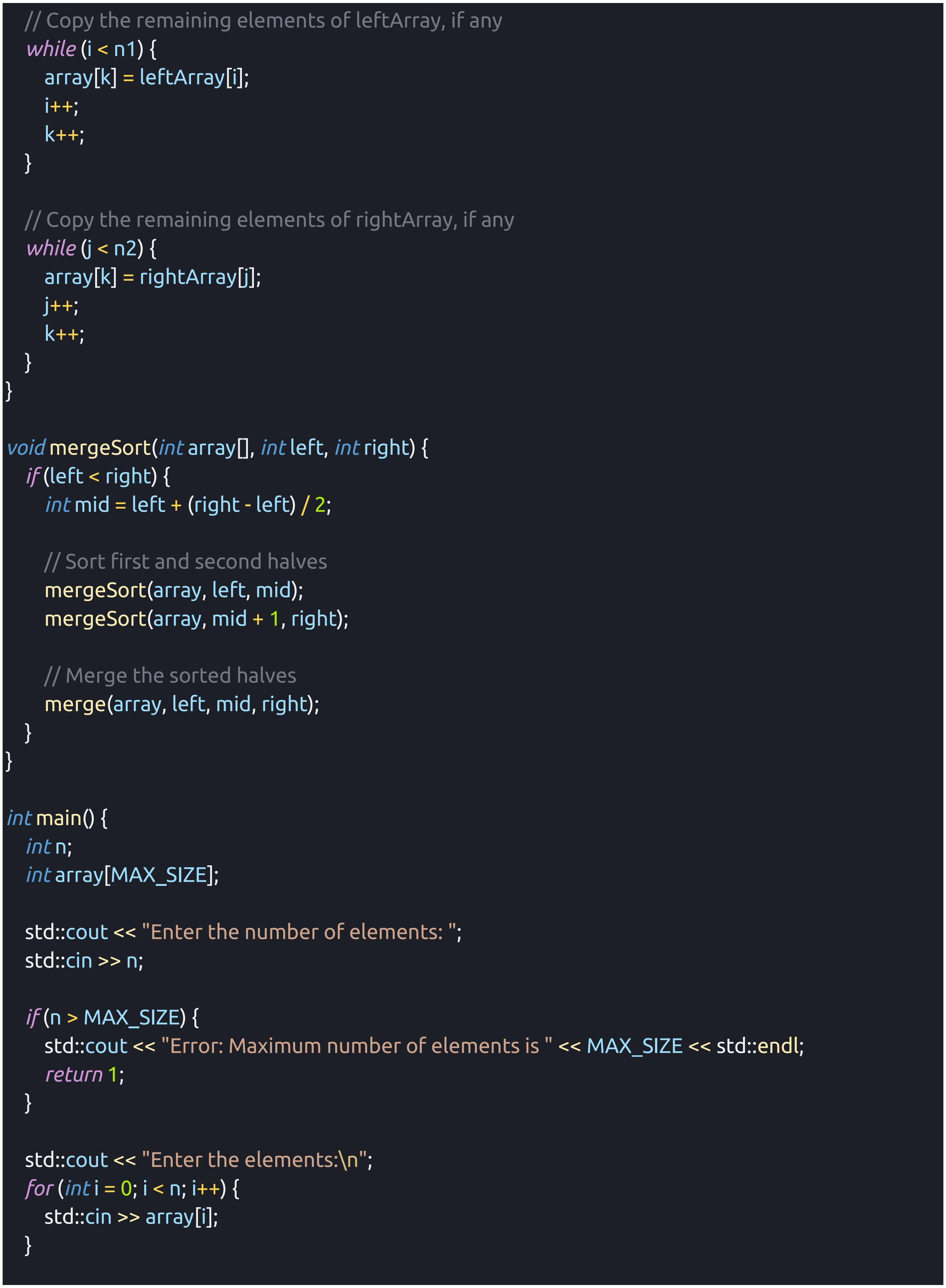
INF 1 0 INF

INF INF 2 0

Merge Sort:



```
#include <iostream>

const int MAX_SIZE = 100; // Define a constant for the maximum array size

void merge(int array[], int left, int mid, int right) {
  int n1 = mid - left + 1;
  int n2 = right - mid;

  // Create temporary arrays
  int leftArray[MAX_SIZE], rightArray[MAX_SIZE];

  // Copy data to temporary arrays leftArray and rightArray
  for (int i = 0; i < n1; i++)
    leftArray[i] = array[left + i];
  for (int j = 0; j < n2; j++)
    rightArray[j] = array[mid + 1 + j];

  // Merge the temporary arrays back into array[left..right]
  int i = 0; // Initial index of first subarray
  int j = 0; // Initial index of second subarray
  int k = left; // Initial index of merged subarray

  while (i < n1 && j < n2) {
    if (leftArray[i] <= rightArray[j]) {
      array[k] = leftArray[i];
      i++;
    } else {
      array[k] = rightArray[j];
      j++;
    }
    k++;
  }
```



```cpp
    // Copy the remaining elements of leftArray, if any
    while (i < n1) {
        array[k] = leftArray[i];
        i++;
        k++;
    }

    // Copy the remaining elements of rightArray, if any
    while (j < n2) {
        array[k] = rightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int array[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(array, left, mid);
        mergeSort(array, mid + 1, right);

        // Merge the sorted halves
        merge(array, left, mid, right);
    }
}

int main() {
    int n;
    int array[MAX_SIZE];

    std::cout << "Enter the number of elements: ";
    std::cin >> n;

    if (n > MAX_SIZE) {
        std::cout << "Error: Maximum number of elements is " << MAX_SIZE << std::endl;
        return 1;
    }

    std::cout << "Enter the elements:\n";
    for (int i = 0; i < n; i++) {
        std::cin >> array[i];
    }
    mergeSort(array, 0, n - 1);

    std::cout << "Sorted array:\n";
    for (int i = 0; i < n; i++) {
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

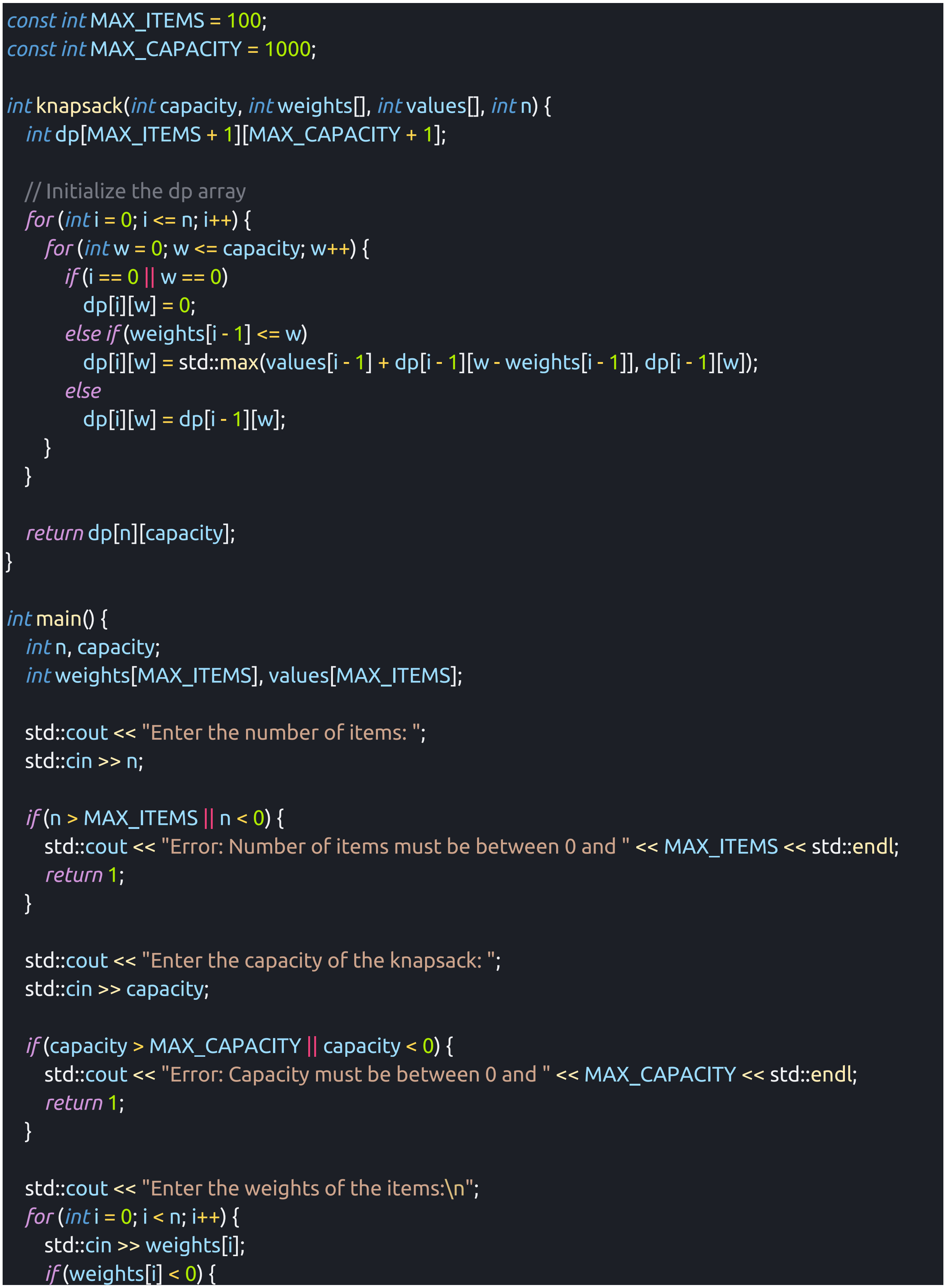
Quick Sort:

```
#include <iostream>

const int MAX_SIZE = 100; // Define a constant for the maximum array size

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int partition(int array[], int low, int high) {
    int pivot = array[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than or equal to pivot
        if (array[j] <= pivot) {
            i++; // increment index of smaller element
            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) {
```

```
    // pi is partitioning index, array[p] is now at right place
    int pi = partition(array, low, high);

    // Separately sort elements before partition and after partition
    quickSort(array, low, pi - 1);
    quickSort(array, pi + 1, high);
  }
}

int main() {
  int n;
  int array[MAX_SIZE];

  std::cout << "Enter the number of elements: ";
  std::cin >> n;

  if (n > MAX_SIZE) {
    std::cout << "Error: Maximum number of elements is " << MAX_SIZE << std::endl;
    return 1;
  }

  std::cout << "Enter the elements:\n";
  for (int i = 0; i < n; i++) {
    std::cin >> array[i];
  }

  quickSort(array, 0, n - 1);

  std::cout << "Sorted array:\n";
  for (int i = 0; i < n; i++) {
    std::cout << array[i] << " ";
  }
  std::cout << std::endl;

  return 0;
}
```

Knapsack:

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



```cpp
const int MAX_ITEMS = 100;
const int MAX_CAPACITY = 1000;

int knapsack(int capacity, int weights[], int values[], int n) {
  int dp[MAX_ITEMS + 1][MAX_CAPACITY + 1];

  // Initialize the dp array
  for (int i = 0; i <= n; i++) {
    for (int w = 0; w <= capacity; w++) {
      if (i == 0 || w == 0)
        dp[i][w] = 0;
      else if (weights[i - 1] <= w)
        dp[i][w] = std::max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
      else
        dp[i][w] = dp[i - 1][w];
    }
  }

  return dp[n][capacity];
}

int main() {
  int n, capacity;
  int weights[MAX_ITEMS], values[MAX_ITEMS];

  std::cout << "Enter the number of items: ";
  std::cin >> n;

  if (n > MAX_ITEMS || n < 0) {
    std::cout << "Error: Number of items must be between 0 and " << MAX_ITEMS << std::endl;
    return 1;
  }

  std::cout << "Enter the capacity of the knapsack: ";
  std::cin >> capacity;

  if (capacity > MAX_CAPACITY || capacity < 0) {
    std::cout << "Error: Capacity must be between 0 and " << MAX_CAPACITY << std::endl;
    return 1;
  }

  std::cout << "Enter the weights of the items:\n";
  for (int i = 0; i < n; i++) {
    std::cin >> weights[i];
    if (weights[i] < 0) {
            std::cout << "Error: Weight must be non-negative.\n";
            return 1;
        }
    }

    std::cout << "Enter the values of the items:\n";
    for (int i = 0; i < n; i++) {
        std::cin >> values[i];
        if (values[i] < 0) {
            std::cout << "Error: Value must be non-negative.\n";
            return 1;
        }
    }

    int max_value = knapsack(capacity, weights, values, n);

    std::cout << "The maximum value that can be put in a knapsack of capacity " << capacity << " is "
<< max_value << std::endl;

    return 0;
}
```

User Input:

Enter the number of items:

4

Enter the capacity of the knapsack:

10

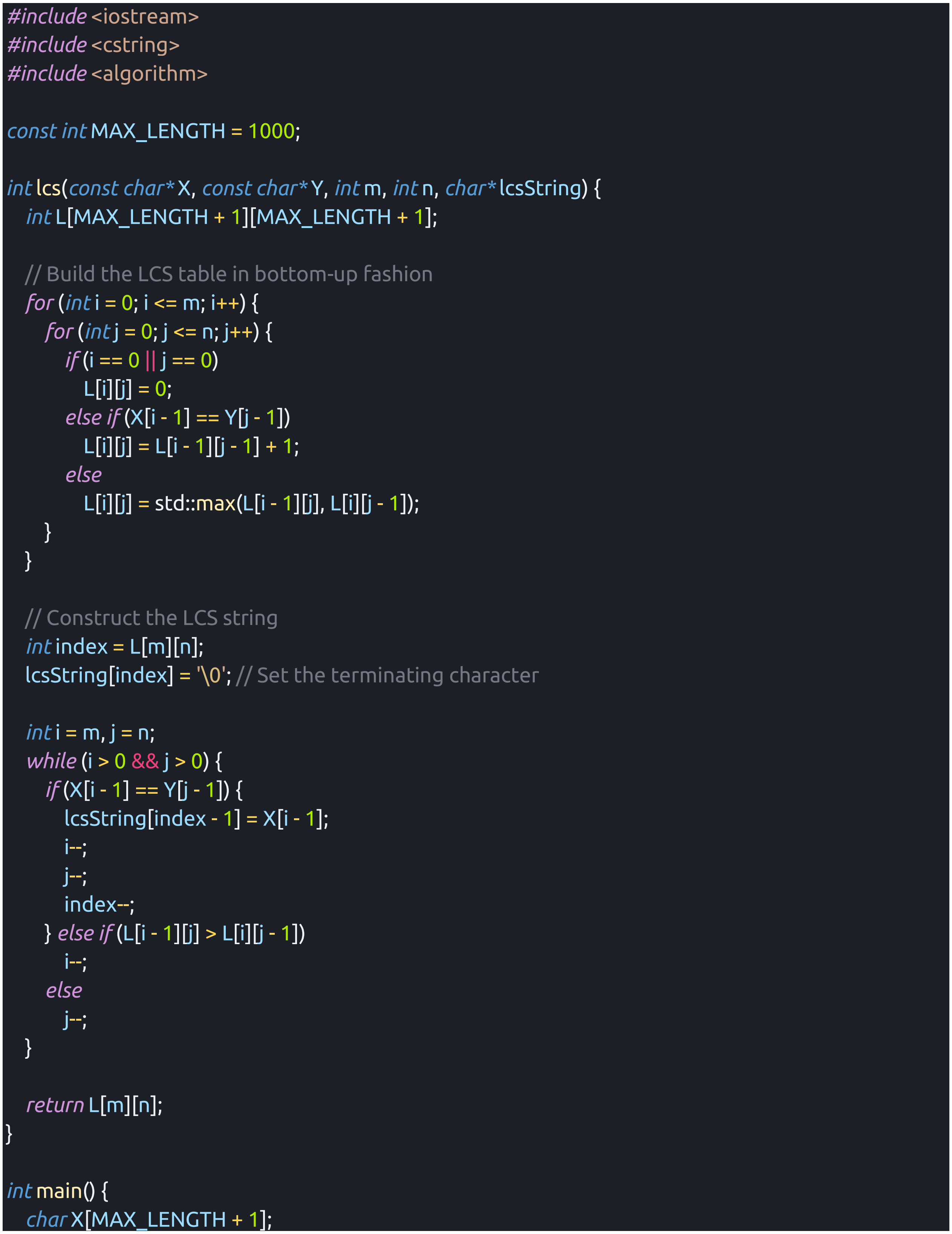
Enter the weights of the items:

2 3 4 5

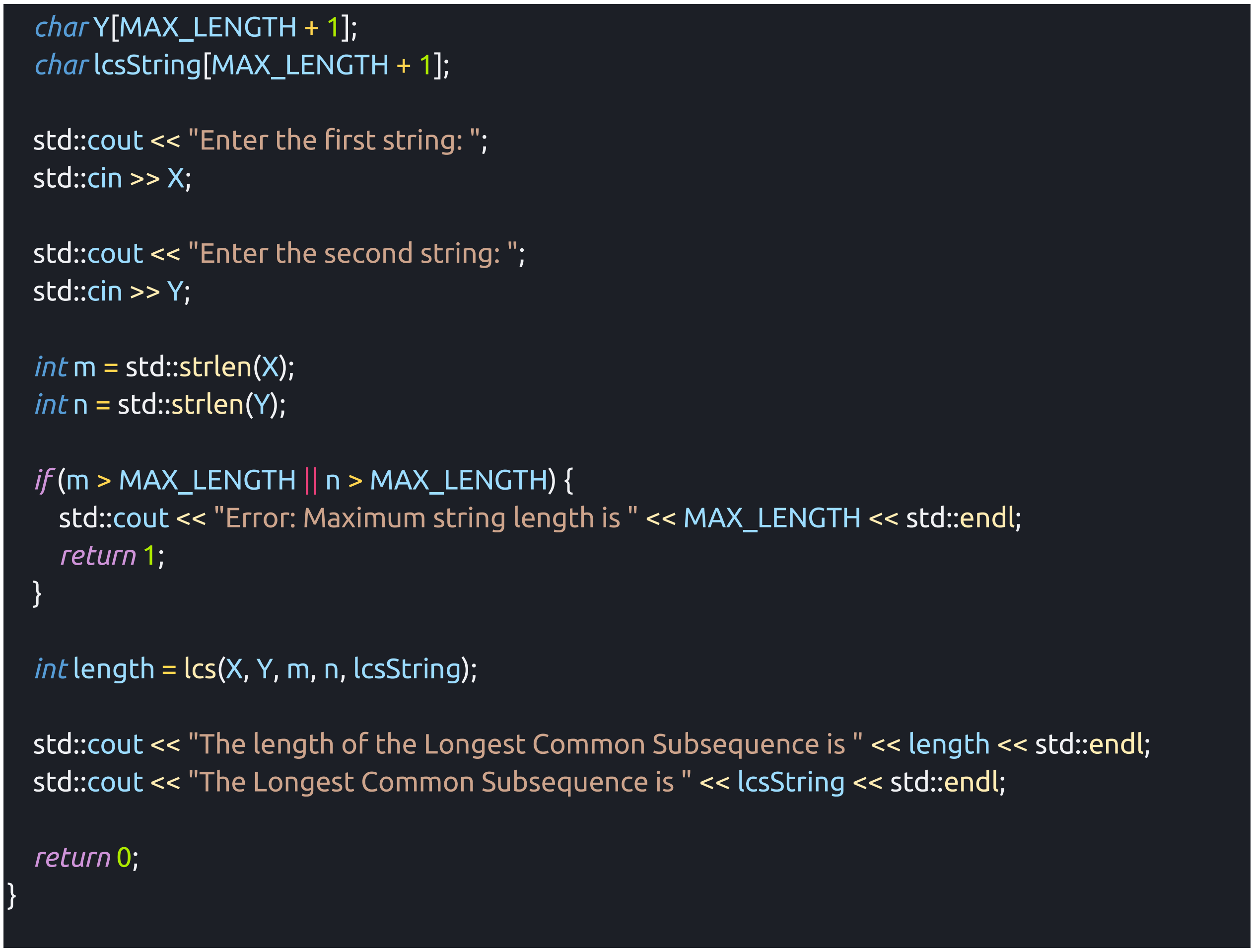
Enter the values of the items:

3 4 5 6

LCS:



```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

const int MAX_LENGTH = 1000;

int lcs(const char* X, const char* Y, int m, int n, char* lcsString) {
   int L[MAX_LENGTH + 1][MAX_LENGTH + 1];

   // Build the LCS table in bottom-up fashion
   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         if (i == 0 || j == 0)
            L[i][j] = 0;
         else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;
         else
            L[i][j] = std::max(L[i - 1][j], L[i][j - 1]);
      }
   }

   // Construct the LCS string
   int index = L[m][n];
   lcsString[index] = '\0'; // Set the terminating character

   int i = m, j = n;
   while (i > 0 && j > 0) {
      if (X[i - 1] == Y[j - 1]) {
         lcsString[index - 1] = X[i - 1];
         i--;
         j--;
         index--;
      } else if (L[i - 1][j] > L[i][j - 1])
         i--;
      else
         j--;
   }

   return L[m][n];
}

int main() {
   char X[MAX_LENGTH + 1];
```



```
    char Y[MAX_LENGTH + 1];
    char lcsString[MAX_LENGTH + 1];

    std::cout << "Enter the first string: ";
    std::cin >> X;

    std::cout << "Enter the second string: ";
    std::cin >> Y;

    int m = std::strlen(X);
    int n = std::strlen(Y);

    if (m > MAX_LENGTH || n > MAX_LENGTH) {
        std::cout << "Error: Maximum string length is " << MAX_LENGTH << std::endl;
        return 1;
    }

    int length = lcs(X, Y, m, n, lcsString);

    std::cout << "The length of the Longest Common Subsequence is " << length << std::endl;
    std::cout << "The Longest Common Subsequence is " << lcsString << std::endl;

    return 0;
}
```

Enter the first string:

AGGTAB

Enter the second string:

GXTXAYB