#include <iostream>

#include <vector>

#include <queue>

#include <utility>

#include <cstring>

#include <climits>

using namespace std;

// `N` is the total number of total nodes on the graph or cities on the map

#define N 5

// Sentinel value for representing `INFINITY`

#define INF INT\_MAX

// State Space Tree nodes

struct Node

{

    // stores edges of the state-space tree

    // help in tracing the path when the answer is found

    vector<pair<int, int>> path;

    // stores the reduced matrix

    int reducedMatrix[N][N];

    // stores the lower bound

    int cost;

    // stores the current city number

    int vertex;

    // stores the total number of cities visited so far

    int level;

};

// Function to allocate a new node `(i, j)` corresponds to visiting city `j`

// from city `i`

Node\* newNode(int parentMatrix[N][N], vector<pair<int, int>> const &path, int level, int i, int j)

{

    Node\* node = new Node;

    // stores ancestors edges of the state-space tree

    node->path = path;

    // skip for the root node

    if (level != 0)

    {

        // add a current edge to the path

        node->path.push\_back(make\_pair(i, j));

    }

    // copy data from the parent node to the current node

    memcpy(node->reducedMatrix, parentMatrix, sizeof node->reducedMatrix);

    // Change all entries of row `i` and column `j` to `INFINITY`

    // skip for the root node

    for (int k = 0; level != 0 && k < N; k++)

    {

        // set outgoing edges for the city `i` to `INFINITY`

        node->reducedMatrix[i][k] = INF;

        // set incoming edges to city `j` to `INFINITY`

        node->reducedMatrix[k][j] = INF;

    }

    // Set `(j, 0)` to `INFINITY`

    // here start node is 0

    node->reducedMatrix[j][0] = INF;

    // set number of cities visited so far

    node->level = level;

    // assign current city number

    node->vertex = j;

    // return node

    return node;

}

// Function to reduce each row so that there must be at least one zero in each row

int rowReduction(int reducedMatrix[N][N], int row[N])

{

    // initialize row array to `INFINITY`

    fill\_n(row, N, INF);

    // `row[i]` contains minimum in row `i`

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (reducedMatrix[i][j] < row[i]) {

                row[i] = reducedMatrix[i][j];

            }

        }

    }

    // reduce the minimum value from each element in each row

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (reducedMatrix[i][j] != INF && row[i] != INF) {

                reducedMatrix[i][j] -= row[i];

            }

        }

    }

}

// Function to reduce each column so that there must be at least one zero

// in each column

int columnReduction(int reducedMatrix[N][N], int col[N])

{

    // initialize all elements of array `col` with `INFINITY`

    fill\_n(col, N, INF);

    // `col[j]` contains minimum in col `j`

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (reducedMatrix[i][j] < col[j]) {

                col[j] = reducedMatrix[i][j];

            }

        }

    }

    // reduce the minimum value from each element in each column

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (reducedMatrix[i][j] != INF && col[j] != INF) {

                reducedMatrix[i][j] -= col[j];

            }

        }

    }

}

// Function to get the lower bound on the path starting at the current minimum node

int calculateCost(int reducedMatrix[N][N])

{

    // initialize cost to 0

    int cost = 0;

    // Row Reduction

    int row[N];

    rowReduction(reducedMatrix, row);

    // Column Reduction

    int col[N];

    columnReduction(reducedMatrix, col);

    // the total expected cost is the sum of all reductions

    for (int i = 0; i < N; i++)

    {

        cost += (row[i] != INT\_MAX) ? row[i] : 0,

            cost += (col[i] != INT\_MAX) ? col[i] : 0;

    }

    return cost;

}

// Function to print list of cities visited following least cost

void printPath(vector<pair<int, int>> const &list)

{

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

        cout << list[i].first + 1 << " —> " << list[i].second + 1 << endl;

    }

}

// Comparison object to be used to order the heap

struct comp

{

    bool operator()(const Node\* lhs, const Node\* rhs) const {

        return lhs->cost > rhs->cost;

    }

};

// Function to solve the traveling salesman problem using Branch and Bound

int solve(int costMatrix[N][N])

{

    // Create a priority queue to store live nodes of the search tree

    priority\_queue<Node\*, vector<Node\*>, comp> pq;

    vector<pair<int, int>> v;

    // create a root node and calculate its cost.

    // The TSP starts from the first city, i.e., node 0

    Node\* root = newNode(costMatrix, v, 0, -1, 0);

    // get the lower bound of the path starting at node 0

    root->cost = calculateCost(root->reducedMatrix);

    // Add root to the list of live nodes

    pq.push(root);

    // Finds a live node with the least cost, adds its children to the list of

    // live nodes, and finally deletes it from the list

    while (!pq.empty())

    {

        // Find a live node with the least estimated cost

        Node\* min = pq.top();

        // The found node is deleted from the list of live nodes

        pq.pop();

        // `i` stores the current city number

        int i = min->vertex;

        // if all cities are visited

        if (min->level == N - 1)

        {

            // return to starting city

            min->path.push\_back(make\_pair(i, 0));

            // print list of cities visited

            printPath(min->path);

            // return optimal cost

            return min->cost;

        }

        // do for each child of min

        // `(i, j)` forms an edge in a space tree

        for (int j = 0; j < N; j++)

        {

            if (min->reducedMatrix[i][j] != INF)

            {

                // create a child node and calculate its cost

                Node\* child = newNode(min->reducedMatrix, min->path,

                    min->level + 1, i, j);

                /\* Cost of the child =

                    cost of the parent node +

                    cost of the edge(i, j) +

                    lower bound of the path starting at node j

                \*/

                child->cost = min->cost + min->reducedMatrix[i][j]

                            + calculateCost(child->reducedMatrix);

                // Add a child to the list of live nodes

                pq.push(child);

            }

        }

        // free node as we have already stored edges `(i, j)` in vector.

        // So no need for a parent node while printing the solution.

        delete min;

    }

}

int main()

{

    // cost matrix for traveling salesman problem.

    /\*

    int costMatrix[N][N] =

    {

        {INF, 5, INF, 6, 5, 4},

        {5, INF, 2, 4, 3, INF},

        {INF, 2, INF, 1, INF, INF},

        {6, 4, 1, INF, 7, INF},

        {5, 3, INF, 7, INF, 3},

        {4, INF, INF, INF, 3, INF}

    };

    \*/

    // cost 34

    int costMatrix[N][N] =

    {

        { INF, 10, 8, 9, 7 },

        { 10, INF, 10, 5, 6 },

        { 8, 10, INF, 8, 9 },

        { 9, 5, 8, INF, 6 },

        { 7, 6, 9, 6, INF }

    };

    /\*

    // cost 16

    int costMatrix[N][N] =

    {

        {INF, 3, 1, 5, 8},

        {3, INF, 6, 7, 9},

        {1, 6, INF, 4, 2},

        {5, 7, 4, INF, 3},

        {8, 9, 2, 3, INF}

    };

    \*/

    /\*

    // cost 8

    int costMatrix[N][N] =

    {

        {INF, 2, 1, INF},

        {2, INF, 4, 3},

        {1, 4, INF, 2},

        {INF, 3, 2, INF}

    };

    \*/

    /\*

    // cost 12

    int costMatrix[N][N] =

    {

        {INF, 5, 4, 3},

        {3, INF, 8, 2},

        {5, 3, INF, 9},

        {6, 4, 3, INF}

    };

    \*/

    cout << "\n\nTotal cost is " << solve(costMatrix);

    return 0;

}