Part B - BFS
Discovery: Denver -> Kansas City
Discovery: Denver -> Chicago
Discovery: Denver -> Los Angeles
Discovery: Denver -> San Francisco
Discovery: Denver -> Seattle
Discovery: Kansas City -> Dallas
Cross: Kansas City -> Chicago
Cross: Kansas City -> Denver
Discovery: Kansas City -> Atlanta
Discovery: Kansas City -> New York
Cross: Kansas City -> Los Angeles
Cross: Chicago -> Kansas City
Cross: Chicago -> New York
Discovery: Chicago -> Boston
Cross: Chicago -> Denver
Cross: Chicago -> Seattle
Cross: Los Angeles -> San Francisco
Cross: Los Angeles -> Denver
Cross: Los Angeles -> Dallas
Cross: Los Angeles -> Kansas City
Cross: San Francisco -> Los Angeles
Cross: San Francisco -> Seattle
Cross: San Francisco -> Denver
Cross: Seattle -> San Francisco
Cross: Seattle -> Denver
Cross: Seattle -> Chicago
Cross: Dallas -> Kansas City
Cross: Dallas -> Los Angeles
Discovery: Atlanta -> Miami
Discovery: Atlanta -> Houston
Cross: Atlanta -> Kansas City
Cross: Atlanta -> New York
Cross: New York -> Boston
Cross: New York -> Chicago
Cross: New York -> Atlanta
Cross: New York -> Kansas City
Cross: Boston -> New York
Cross: Boston -> Chicago
Cross: Miami -> Atlanta
Cross: Miami -> Houston
Cross: Houston -> Atlanta
Cross: Houston -> Miami

```
#ifndef DATA_H
#define DATA_H

#include <unordered_map>
#include <string>

enum City {
    AT,     // 0
    BO,     // 1
    CHI,    // 2
    DA,     // 3
    DEN,    // 4
    HO,     // 5
    KC,     // 6
    LA,     // 7
    MI,     // 8
    NY,     // 9
    SF,     // 10
    SE      // 11
};

std::unordered_map<int, std::string> cities {
    { AT, "Atlanta" },
    { BO, "Boston" },
    { CHI, "Chicago" },
    { DA, "Dallas" },
    { DEN, "Denver" },
    { HO, "Houston" },
    { KC, "Kansas City" },
    { LA, "Los Angeles" },
    { MI, "Miami" },
    { NY, "New York" },
    { SF, "San Francisco" },
    { SE, "Seattle" }
};

// PRINTS CITY NAME
void printCity (int index) {
    auto it = cities.find(index);

    if (it != cities.end()) {
        std::cout << it->second << " ";
    }
```

```
} // END printCity
#endif
```

```
/*****************************************************************************
 * bfs.h - Part 2
 *---------------------------------------------------------------------------
 * Header file for Breadth-First Search (BFS) algorithm
 *****************************************************************************/
#ifndef BFS_H
#define BFS_H

#include "data.h"

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <unordered_map>

void addEdge(std::vector<std::vector<int>> &mat, int cityA, int cityB, int dis) {
    mat[cityA][cityB] = dis;
    mat[cityB][cityA] = dis;
}

// OUTPUT MATRIX - test to see if worked
// void displayMatrix (std::vector<std::vector<int>> &mat) {
//     int v = mat.size();

//     for (int cityA = 0; cityA < v; cityA++) {
//         for (int cityB = 0; cityB < v; cityB++) {
//             std::cout << mat[cityA][cityB] << " ";
//         }
//         std::cout << std::endl;
//     }
// }


// BFS
void bfs(const std::vector<std::vector<int>> &mat, int start) {
    int v = mat.size();

    if (v == 0) return;
    std::vector<bool> visited(v, false);            // univisited cities
    std::vector<int> parent(v, -1);
    std::queue<int> order;                          // cities in order
    std::unordered_map<int, std::string> edgeType;  // stores city edge type
    int discoverDis = 0;            // distances for discover
```

```
order.push(start);
visited[start] = true;

while (!order.empty()) {
    int ver = order.front();
    order.pop();

    // find neihbors of vertex
    std::vector<std::pair<int,int>> neighbors;
    for (int j = 0; j < v; ++j) {
        int weight = mat[ver][j];
        // treat 0 as "no edge" — change if your sentinel differs
        if (weight != 0) neighbors.emplace_back(weight, j);
    }

    // sort ascending by weight, tie-break by index
    std::sort(neighbors.begin(), neighbors.end(),
              [](const std::pair<int,int>& cityA, const std::pair<int,int>& cityB){
                  if (cityA.first != cityB.first) return cityA.first < cityB.first;
                  return cityA.second < cityB.second;
              });

    // enqueue unvisited neighbors in ascending-weight order
    for (auto &p : neighbors) {
        int weight = p.first;
        int neigh = p.second;
        // discovery
        if (!visited[neigh]) {
            visited[neigh] = true;
            order.push(neigh);

            // save type
            parent[neigh] = ver;
            edgeType[ver] = "Discovery";
            std::cout << "Discovery: ";
            printCity(ver);
            std::cout << " -> ";
            printCity(neigh);
            std::cout << "\n";

            discoverDis += weight;
        }
        // cross
```

```
            else {
                edgeType[ver] = "Cross";
                std::cout << "Cross: ";
                printCity(ver);
                std::cout << " -> ";
                printCity(neigh);
                std::cout << "\n";
            }
        }
    }
    std::cout << "\nTotal Distance Traveled: " << discoverDis << std::endl;
}

#endif // BFS_H
```

```
/*****************************************************************************
 * main.cpp
 *---------------------------------------------------------------------------
 * By Aspen Cristobal and Amy
 *---------------------------------------------------------------------------
 * Main source file for Breadth-First Search (BFS) algorithm
 *****************************************************************************/
#include <iostream>
#include <string>
#include <vector>

#include "bfs.h"
#include "data.h"

//using namespace std;

int main() {
    const int SIZE = cities.size();

    /*****************************************************************************
    * DATA
    *****************************************************************************/


    /*****************************************************************************
    * PART 1 - DFS Implementation
    *****************************************************************************/

    /*****************************************************************************
    * PART 2 - BFS Implementation
    *****************************************************************************/
    std::vector<std::vector<int>> adjMat(SIZE, std::vector<int>(SIZE, 0));

    addEdge(adjMat, SE, SF, 807);
    addEdge(adjMat, SE, DEN, 1331);
    addEdge(adjMat, SE, CHI, 2097);
    addEdge(adjMat, SF, DEN, 1267);
    addEdge(adjMat, SF, LA, 381);
    addEdge(adjMat, DEN, LA, 1015);
    addEdge(adjMat, DEN, KC, 599);
    addEdge(adjMat, DEN, CHI, 1003);
    addEdge(adjMat, CHI, BO, 983);
    addEdge(adjMat, CHI, NY, 787);
    addEdge(adjMat, CHI, KC, 533);
```

```
    addEdge(adjMat, LA, DEN, 1015);
    addEdge(adjMat, LA, KC, 1663);
    addEdge(adjMat, LA, DA, 1435);
    addEdge(adjMat, KC, DA, 496);
    addEdge(adjMat, KC, AT, 864);
    addEdge(adjMat, KC, NY, 1260);
    addEdge(adjMat, NY, BO, 214);
    addEdge(adjMat, NY, AT, 888);
    addEdge(adjMat, AT, MI, 661);
    addEdge(adjMat, AT, HO, 810);
    addEdge(adjMat, HO, MI, 1187);

    std::cout << "Part B - BFS\n";
    bfs(adjMat, DEN);

    return 0;
} // END main
```