```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <queue>
#include <cmath>
#include <unordered_set>

using namespace std;

struct Place {
    int index;
    string name;
    double x, y;
};

struct Edge {
    int from, to;
    double weight;

    bool operator>(const Edge& other) const {
        return weight > other.weight;
    }
};

double calculateDistance(const Place& a, const Place& b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}

vector<string> split(const string &s, char delimiter) {
    vector<string> tokens;
    string token;
    istringstream tokenStream(s);
    while (getline(tokenStream, token, delimiter)) {
        tokens.push_back(token);
    }
    return tokens;
}

vector<Place> readPlacesFromFile(const string& filename) {
    ifstream file(filename);
    vector<Place> places;
    string line;

    int index = 0;
    while (getline(file, line)) {
        vector<string> tokens = split(line, ',');
        if (tokens.size() == 3) {
            Place place;
            place.index = index++;
            place.name = tokens[0];
            place.x = stod(tokens[1]);
            place.y = stod(tokens[2]);
            places.push_back(place);
        }
    }

    return places;
```

```
}

void printPlaces(const vector<Place>& places) {
    for (const auto& place : places) {
        cout << "Index: " << place.index << ", Name: " << place.name << ", X: " <<
place.x << ", Y: " << place.y << endl;
    }
}

//todo
vector<Place> tspBruteForce(const vector<Place>& places) {
    vector<Place> brutePath;
    double minDistance = numeric_limits<double>::max();

    vector<int> indexes(places.size());

    for (size_t i = 0; i < indexes.size(); ++i) {
        indexes[i] = i;
    }
    // Checks all permutations of the cities
    do {
        double currDistance = 0.0;
        for (size_t i = 0; i < places.size() - 1; ++i) {
            currDistance += calculateDistance(places[indexes[i]], places[indexes[i
+ 1]]);
        }
        currDistance += calculateDistance(places[indexes.back()],
places[indexes.front()]);

        // Checks if the current minumum is greater than current distance
        if (currDistance < minDistance) {
            minDistance = currDistance;
            brutePath.clear();
            for (size_t i = 0; i < places.size(); ++i) {
                brutePath.push_back(places[indexes[i]]);
            }
        }
    } while (next_permutation(indexes.begin() + 1, indexes.end()));

    return brutePath;
}

//todo
vector<Place>  tspPrimApproximation(const vector<Place>& places) {
    vector<Place> path;
    priority_queue<Edge, vector<Edge>, greater<Edge>> minHeap;

    unordered_set<int> visited;

    visited.insert(0);

    // Initializing the minHeap with edges starting at the first city
    for (size_t i = 1; i < places.size(); ++i) {
        Edge a{ 0, static_cast<int>(i), calculateDistance(places[0], places[i]) };
        minHeap.push(a);
    }

    while (!minHeap.empty()) {
        Edge currEdge = minHeap.top();
```

```
        minHeap.pop();
        // Checks if the destination city of the currEdge is visited or not
        if (visited.count(currEdge.to) == 0) {
            visited.insert(currEdge.to);
            path.push_back(places[currEdge.to]);

            // Adds edges from the new cities to unvisited cities
            for (size_t i = 0; i < places.size(); ++i) {
                if (visited.count(static_cast<int>(i)) == 0) {
                    Edge a{ currEdge.to, static_cast<int>(i),
calculateDistance(places[currEdge.to], places[i]) };
                    minHeap.push(a);
                }
            }
        }
    }

    return path;
}

int main() {
    string filename;
    vector<Place> places;
    ifstream file;

    cout << "Enter the filename: ";
    cin >> filename;
    file.open(filename);

    if (!file) {
        cout << "File not found. Please try again." << endl;
        return 1;
    }

    string line;
    int index = 0;
    while (getline(file, line)) {
        vector<string> tokens = split(line, ',');
        if (tokens.size() == 4) {
            Place place;
            place.index = index++;
            place.name = tokens[1];
            place.x = stod(tokens[2]);
            place.y = stod(tokens[3]);
            places.push_back(place);
        }
    }
    file.close();

    printPlaces(places); // This will print the index along with each city (will be
useful to you)

    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    int startIndex;
    cout << "Enter the index of the starting city: ";
    cin >> startIndex;

    // Must choose valid start
```

```
    if (startIndex < 0 || startIndex >= places.size()) {
        cout << "Invalid index for starting city." << endl;
        return 1;
    }

    // Ask user for the indices of cities to visit
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    string indicesList;
    cout << "Enter the indices of the cities you wish to visit, separated by commas
(do not include the starting city): ";
    getline(cin, indicesList);
    vector<string> indicesToVisit = split(indicesList, ',');

    unordered_set<int> selectedIndices; // To check for duplicates, don't let user
revisit cities
    vector<Place> selectedPlaces;
    selectedIndices.insert(startIndex); // Add the starting city index after user
has entered their cities to visit

    cout << "Starting City: " << places[startIndex].name << endl;

    for (const auto& indexStr : indicesToVisit) {
        int index = stoi(indexStr);
        if (index == startIndex) {
            cout << "Note: The starting city (Index: " << startIndex << ") was
included in the visit list and will be ignored." << endl;
            continue;
        }
        if (index < 0 || index >= places.size()) {
            cout << "Invalid index: " << index << ". This city will be ignored." <<
endl;
            continue;
        }
        if (selectedIndices.count(index) > 0) {
            cout << "Duplicate index: " << index << ". This city will be ignored."
<< endl;
            continue;
        }
        selectedIndices.insert(index);
        selectedPlaces.push_back(places[index]);
    }

    cout << "Cities to Visit:" << endl;
    for (const auto& place : selectedPlaces) {
        cout << "Index: " << place.index << ", Name: " << place.name << ", X: " <<
place.x << ", Y: " << place.y << endl;
    }

    vector<Place> pathBruteForce = tspBruteForce(selectedPlaces);
    vector<Place> pathPrimApprox = tspPrimApproximation(selectedPlaces);

    // Print paths
    cout << "TSP Brute Force Path:" << endl;
    for (const auto& place : pathBruteForce) {
        cout << "Index: " << place.index << ", Name: " << place.name << ", X: " <<
place.x << ", Y: " << place.y << endl;
    }

    cout << "TSP Prim's Approximation Path:" << endl;
```

```
    for (const auto& place : pathPrimApprox) {
        cout << "Index: " << place.index << ", Name: " << place.name << ", X: " <<
place.x << ", Y: " << place.y << endl;
    }
    return 0;
}
```