EE 450

Lab 1

Report

Name: Yirou Qian

Student ID: 4354716429

Part 1. The Whole Code

#include <iostream>

#include <string>

#include <cstring>

#include <fstream>

#include <sstream>

#include <vector>

#include <climits>

#include <regex>

#include <algorithm>

#include <stdio.h>

**using** **namespace** std;

**const** **int** INF = INT\_MAX;

string srcFileName;

*// read in file*

vector<vector<**int**>> readFile(**const** string fileName){

vector<vector<**int**>> graph;

ifstream inFile(fileName,ios::in);

**if**(!inFile.is\_open()){

cout << "File not found: " << fileName << endl;

exit(0);

}

*//get line in file to string*

string line;

**while** (getline(inFile, line)){

**if** (line.length() < 2){

**continue**;

}

stringstream ss(line);

string strIn;

ss >> strIn;

vector<**int**> vertex;

**while** (strIn.length() > 0){

transform(strIn.begin(), strIn.end(), strIn.begin(), ::toupper);

*// store distance value*

**if** (strIn.compare("INF") == 0) {

vertex.push\_back(INF);

}**else**{

*// multi-cost case*

**if** (strIn[0] == '['){

**int** n1, n2, n3;

sscanf(strIn.c\_str(), "[%d,%d,%d]", &n1, &n2, &n3);

vertex.push\_back(n1 + n2 + n3);

}**else**{

vertex.push\_back(stoi(strIn.c\_str()));

}

}

strIn.clear();

ss >> strIn;

}

*// store values into graph*

graph.push\_back(vertex);

}

**return** graph;

}

*// do Bellman-Ford algorithm*

**void** bellmanFord(vector<vector<**int**>> graph){

**int** numNode = (**int**)graph.size();

**int** count = 0;

vector<**int**> distance;

distance.push\_back(0);

*//initialize the distance*

**for** (**int** i = 1; i < numNode; i++){

distance.push\_back(INT\_MAX);

}

*//last node having done relax calcution*

vector<**int**> preNode(numNode, -1);

**bool** distUpdate;

*//do relax and save the shortest distance value*

**for** (**int** i = 0; i < numNode - 1; i++){

distUpdate = **false**;

count++;

**for** (**int** j = 0; j < numNode; j++){

**for** (**int** k = 0; k < graph[j].size(); k ++){

**if** (graph[j][k] != INT\_MAX && distance[j] != INT\_MAX && distance[j] + graph[j][k] < distance[k]){

distance[k] = distance[j] + graph[j][k];

preNode[k] = j;

distUpdate = **true**;

}

}

}

**if** (!distUpdate){

**break**;

}

}

*//detect negative loop*

count++;

**bool** bFlag = **false**;

**for** (**int** j = 0; j < numNode; j++){

**if**(bFlag){

**break**;

}

**for** (**int** k = 0; k < graph[j].size(); k++){

**if** (graph[j][k] != INT\_MAX && distance[j] != INT\_MAX && distance[j] + graph[j][k] < distance[k]){

bFlag = **true**;

**break**;

}

}

}

*// output file with result*

ofstream outFile;

outFile.open("output-" + srcFileName, ios::out);

**if** (bFlag){

outFile << "Negative Loop Detected" << endl;

*//show negative iteration path*

**int** nPre;

string strOut;

stringstream ssOut;

nPre = preNode[0];

ssOut << nPre << "->" << 0;

strOut = ssOut.str();

**while** (nPre){

nPre = preNode[nPre];

stringstream ssOut2;

ssOut2 << nPre << "->" << strOut;

strOut = ssOut2.str();

}

outFile << strOut << endl;

}**else**{

*//show distance to each node*

outFile << distance[0];

**for** (**int** i = 1; i < numNode; i++){

outFile << "," << distance[i];

}

outFile << endl << 0 << endl;

*//show non-negative iteration path*

**int** nPre;

string strOut;

**for** (**int** i = 1; i < numNode; i++){

stringstream ssOut;

nPre = preNode[i];

ssOut << nPre << "->" << i;

strOut = ssOut.str();

**while** (nPre){

nPre = preNode[nPre];

stringstream ssOut2;

ssOut2 << nPre << "->" << strOut;

strOut = ssOut2.str();

}

outFile << strOut << endl;

}

*// times of iteration*

outFile << "Iteration: " << count << endl;

}

outFile.close();

}

**int** main(**int** argc, **char** \*\*argv){

**if** (argc != 2){

**return** 0;

}

srcFileName = argv[1];

vector<vector<**int**>> newGraph = readFile(argv[1]);

bellmanFord(newGraph);

**return** 0;

}

Part 2. Key Aspects of Implementation

1. Read in test files.
2. Because the graph is directed weighted graph, and considering that I need to record the distance between two nodes, I use the vector<vector< >> which seems to be more convenient and useful. vector<vector<**int**>> graph;

ifstream inFile(fileName,ios::in);

**if**(!inFile.is\_open()){

cout << "File not found: " << fileName << endl;

exit(0);

}

1. When reading in files, first I use stringstream to read the lines from the file and turn them to string and record weight word by word, or calculate the multi-cost of the weight; then store the weight values of vertices in the graph.

string line;

**while** (getline(inFile, line)){

**if** (line.length() < 2){

**continue**;

}

stringstream ss(line);

string strIn;

ss >> strIn;

vector<**int**> vertex;

**while** (strIn.length() > 0){

transform(strIn.begin(), strIn.end(), strIn.begin(), ::toupper);

*// store distance value*

**if** (strIn.compare("INF") == 0) {

vertex.push\_back(INF);

}**else**{

*// multi-cost case*

**if** (strIn[0] == '['){

**int** n1, n2, n3;

sscanf(strIn.c\_str(), "[%d,%d,%d]", &n1, &n2, &n3);

vertex.push\_back(n1 + n2 + n3);

}**else**{

vertex.push\_back(stoi(strIn.c\_str()));

}

}

strIn.clear();

ss >> strIn;

}

*// store values into graph*

graph.push\_back(vertex);

1. Do bellman ford algorithm.
2. Initialize all node and all distance. And do iteration function and refresh the value of the shortest path between two nodes in the graph starting from node 0, which is:D(0v)= min{d0(v)+c(u,v)}. It is the most important part in the whole code. And the total iteration times should be no more than numNode-1 times.

**int** numNode = (**int**)graph.size();

**int** count = 0;

vector<**int**> distance;

distance.push\_back(0);

*//initialize the distance*

**for** (**int** i = 1; i < numNode; i++){

distance.push\_back(INT\_MAX);

}

*//last node having done relax calcution*

vector<**int**> preNode(numNode, -1);

**bool** distUpdate;

*//do relax and save the shortest distance value*

**for** (**int** i = 0; i < numNode - 1; i++){

distUpdate = **false**;

count++;

**for** (**int** j = 0; j < numNode; j++){

**for** (**int** k = 0; k < graph[j].size(); k ++){

**if** (graph[j][k] != INT\_MAX && distance[j] != INT\_MAX && distance[j] + graph[j][k] < distance[k]){

distance[k] = distance[j] + graph[j][k];

preNode[k] = j;

distUpdate = **true**;

}

}

}

**if** (!distUpdate){

**break**;

}

}

1. Calculate the total iteration times and detect negative loop to examine if there is shorter path than that of the above function. If the values are the same, then we finish the whole iteration, which is also important in the whole code.

count++;

**bool** bFlag = **false**;

**for** (**int** j = 0; j < numNode; j++){

**if**(bFlag){

**break**;

}

**for** (**int** k = 0; k < graph[j].size(); k++){

**if** (graph[j][k] != INT\_MAX && distance[j] != INT\_MAX && distance[j] + graph[j][k] < distance[k]){

bFlag = **true**;

**break**;

}

}

}

1. Output the results of test files
2. If find the negative loop, then the output should be “Negative Loop Detect” and show the negative cycle path.

**if** (bFlag){

outFile << "Negative Loop Detected" << endl;

*//show negative iteration path*

**int** nPre;

string strOut;

stringstream ssOut;

nPre = preNode[0];

ssOut << nPre << "->" << 0;

strOut = ssOut.str();

**while** (nPre){

nPre = preNode[nPre];

stringstream ssOut2;

ssOut2 << nPre << "->" << strOut;

strOut = ssOut2.str();

}

outFile << strOut << endl;

1. If there is no negative loop, then the first line of the output should be the shortest distance value from node 0 to the other node. Then it shows the iteration paths of the way to go to each node from node 0 with least weight. And the last line should be the total iteration times.

*//show distance to each node*

outFile << distance[0];

**for** (**int** i = 1; i < numNode; i++){

outFile << "," << distance[i];

}

outFile << endl << 0 << endl;

*//show non-negative iteration path*

**int** nPre;

string strOut;

**for** (**int** i = 1; i < numNode; i++){

stringstream ssOut;

nPre = preNode[i];

ssOut << nPre << "->" << i;

strOut = ssOut.str();

**while** (nPre){

nPre = preNode[nPre];

stringstream ssOut2;

ssOut2 << nPre << "->" << strOut;

strOut = ssOut2.str();

}

outFile << strOut << endl;

}

*// times of iteration*

outFile << "Iteration: " << count << endl;

1. Run main function

Compile main to run input and do bellman ford algorithm, and then output the result.

**int** main(**int** argc, **char** \*\*argv){

**if** (argc != 2){

**return** 0;

}

srcFileName = argv[1];

vector<vector<**int**>> newGraph = readFile(argv[1]);

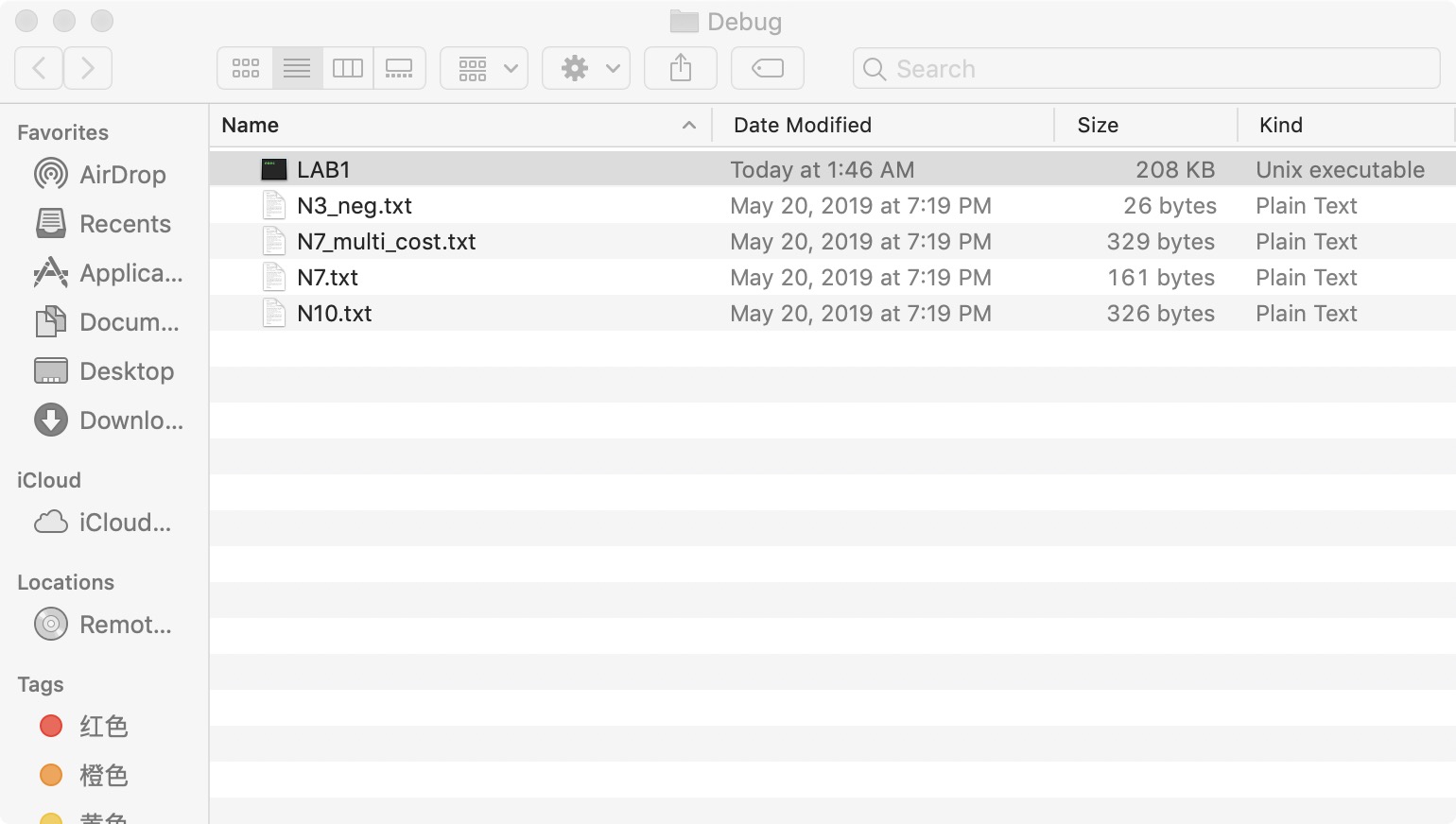
bellmanFord(newGraph);

**return** 0;

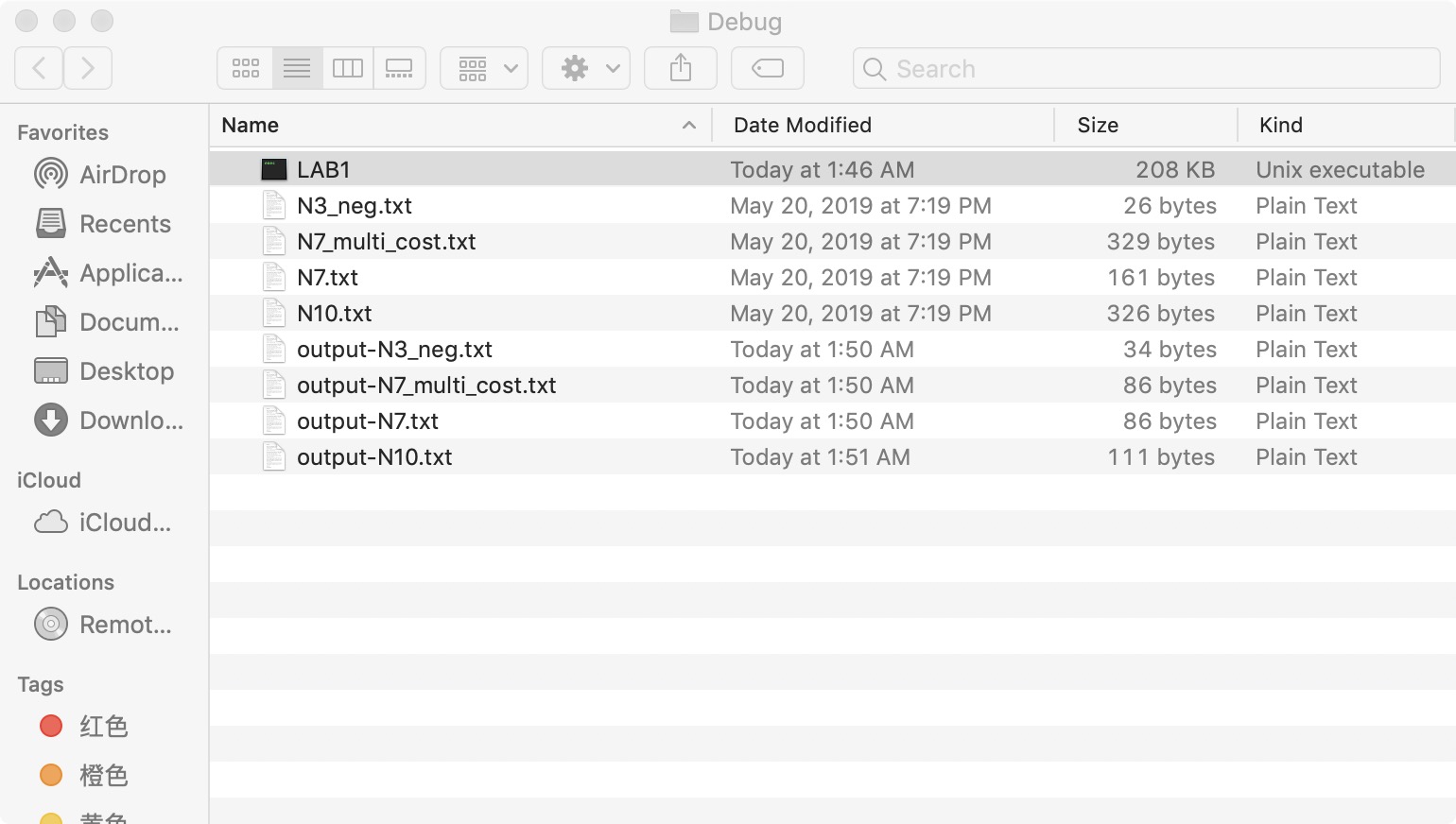
}

Part 3. Experimental Results and The Output File Format

Terminal commands:



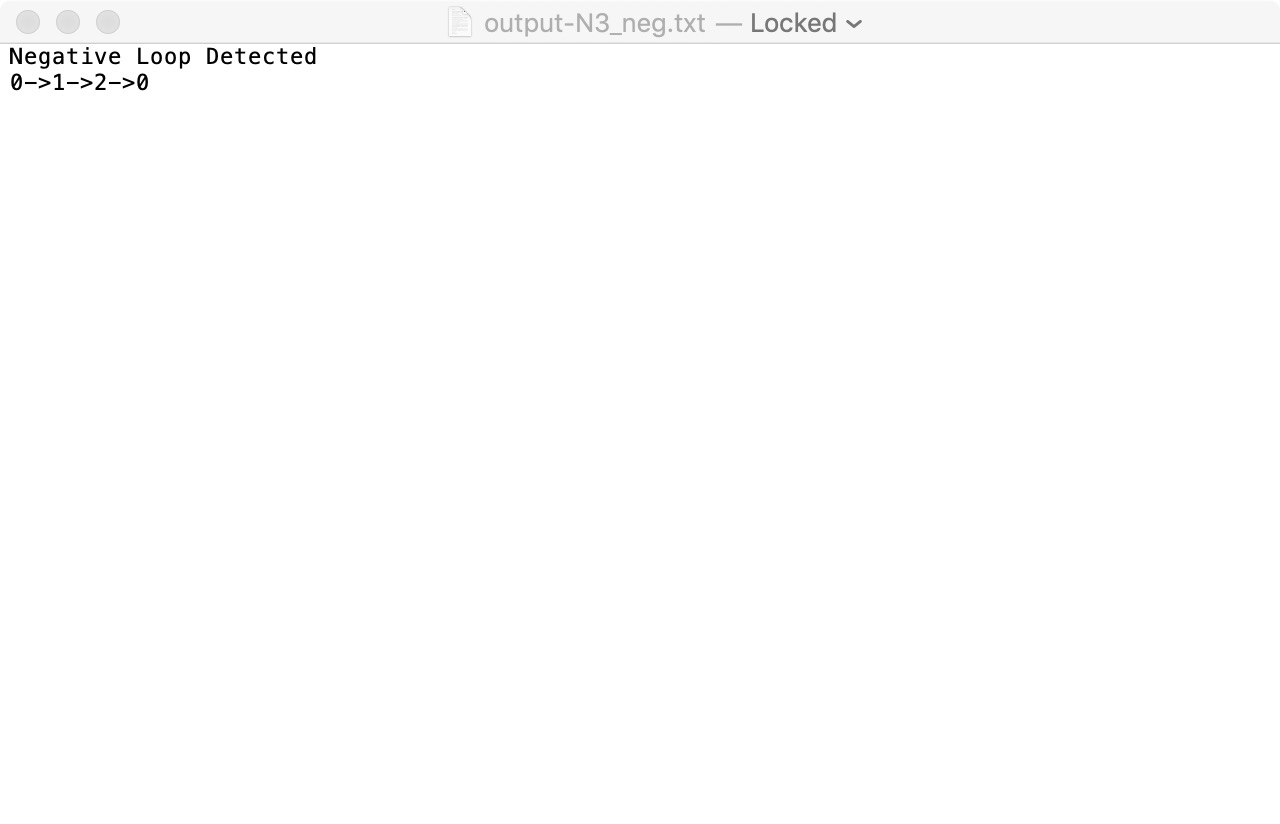




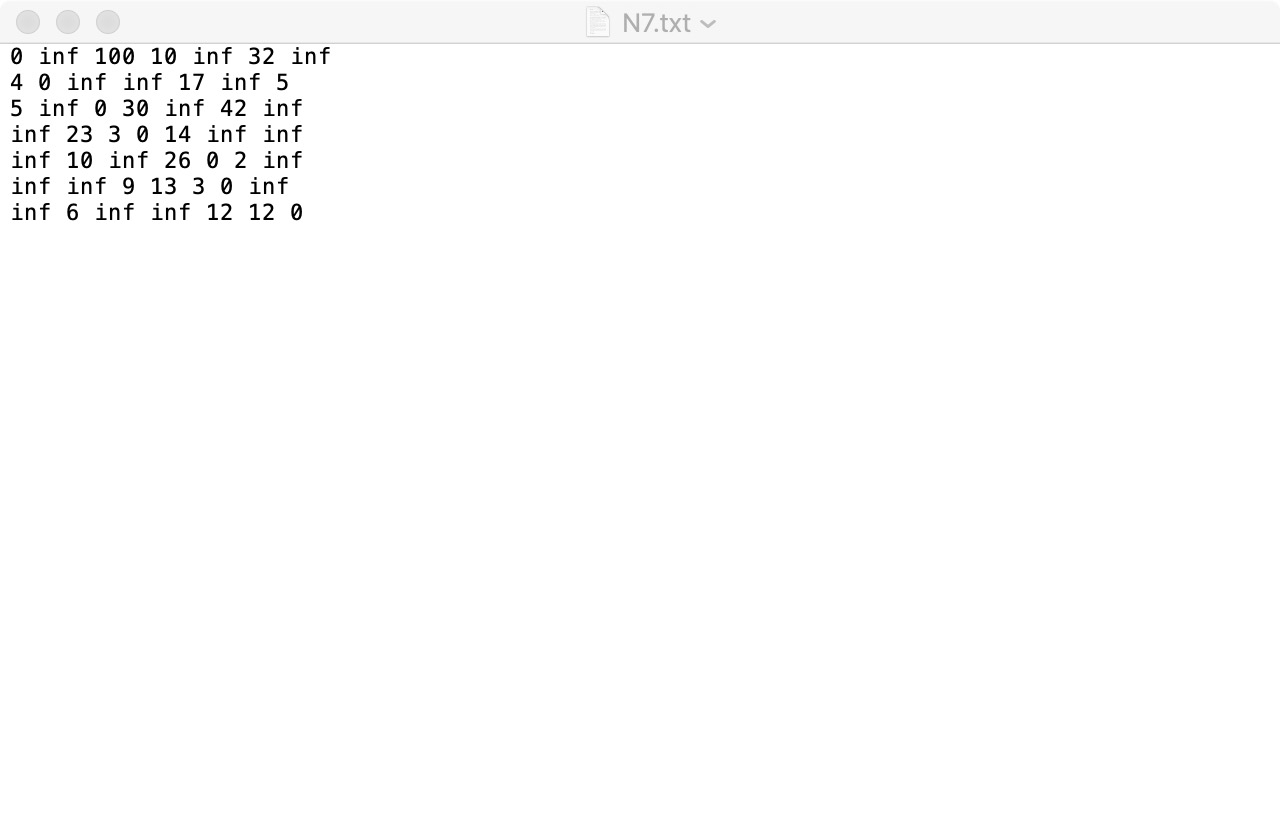
1. N3\_neg.txt
2. Input:



1. Output:



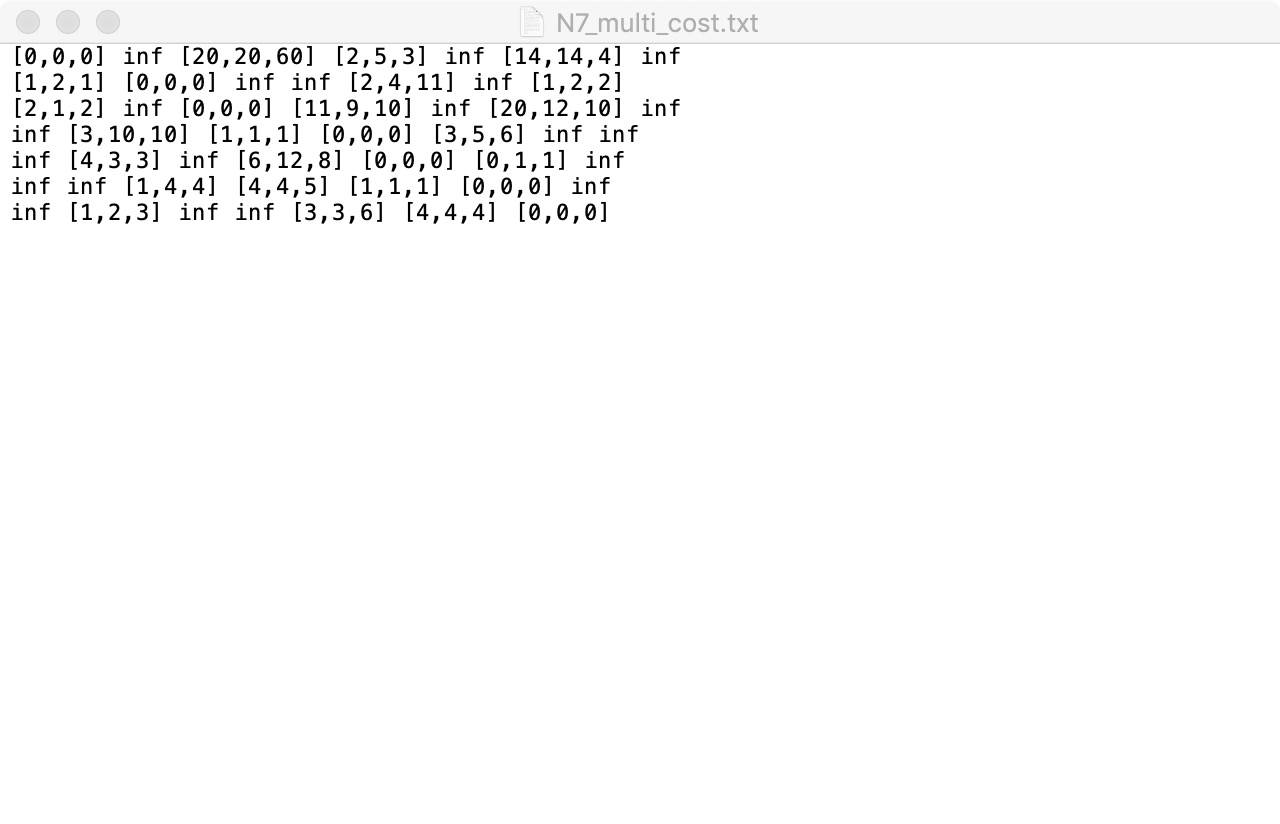
1. N7.txt
2. Input:



1. Output:



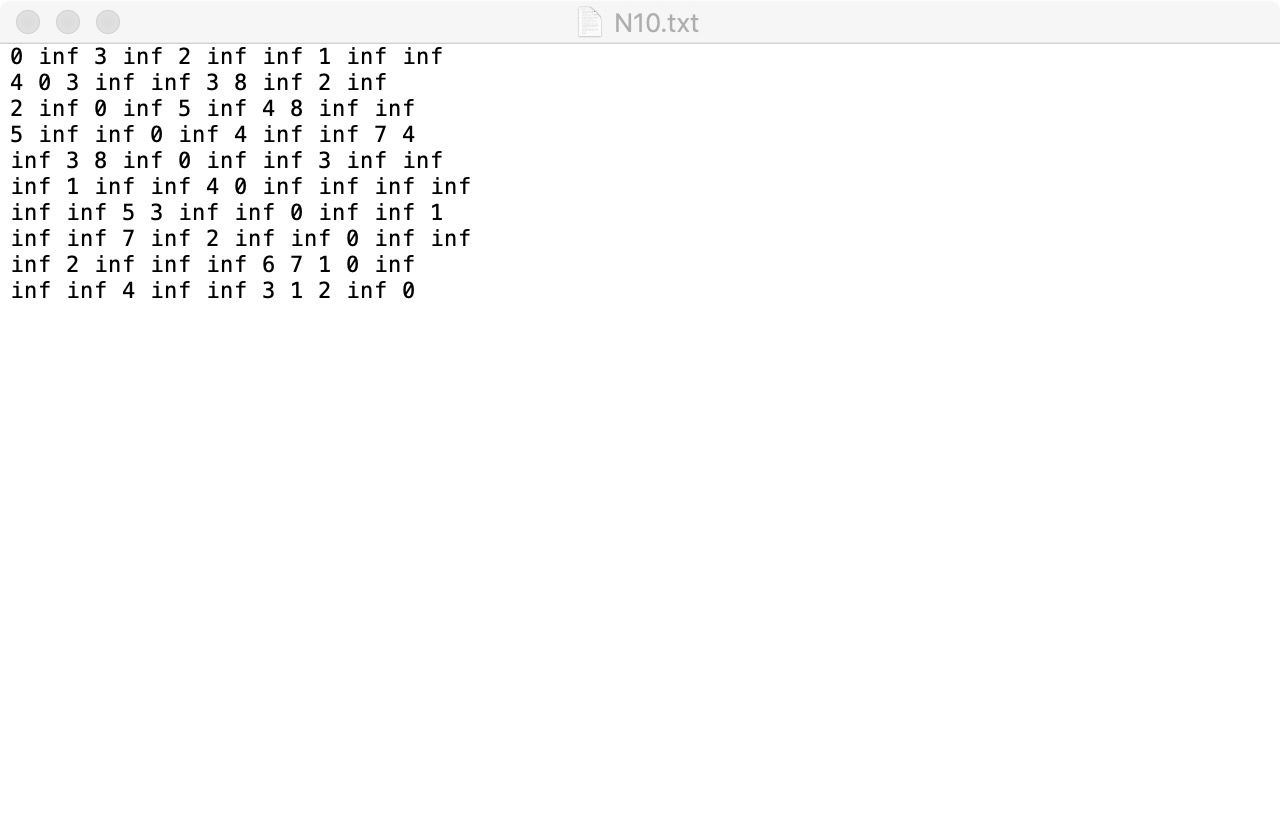
1. N7\_multi\_cost.txt
2. Input:



1. Output:



1. N10.txt
2. Input:



1. Output:

