Computer Programming Language

【Fall, 2020】

Homework 5

Program A： Caesar Shift Cipher Decoder (20%)

Cryptography is one of most interesting branches of programming. Studying its algorithms usually begins with the simple method named after famous Roman emperor Julius Caesar who used it for communicating his military secrets. The idea of the Caesar shift cipher algorithm is simple. Each letter of the original text is substituted by another, by the following rule:

(1) find the letter (which should be encrypted) in the alphabet;
(2) move K positions further (down the alphabet);
(3) take the new letter from here;
(4) if "shifting" encountered the end of the algorithm, continue from its start.

For example, if K = 3, then A becomes D, B becomes E, W becomes Z and Z becomes C and so on.

Your task is to write a Caesar shift cipher decoder program to decode the following encrypted sentence given K = 6:

O RUBK IUSVAZKX VXUMXGSSOTM YU SAIN

■ *AUTOLAB Submission Check:*

```
char answer1;    // Store the first character of the decoded sentence
char answer2;    // Store the last character of the decoded sentence
```

Optional Bonus Points (20%):

You are encouraged to challenge a more difficult problem by designing a Caesar shift cipher cracker program to automatically find out the encrypted sentence without knowing the value of K. The encrypted sentence is as follows. Read the Wikipedia article about Caesar shift cipher algorithm to learn more about the approach to crack the encrypted sentence (https://en.wikipedia.org/wiki/Caesar_cipher).

GWC IZM ZMITTG I PIZL EWZSQVO ABCLMVB

■ *AUTOLAB Submission Check:*

```
int answer1;     // The value of K to decode the above encrypted message.
char answer2;    // Store the first character of the decoded sentence
char answer3;    // Store the last character of the decoded sentence
```

Program B： Game of Life (40%)

The game of life is a computer simulation that was created by a Cambridge Mathematician named John Conway. The idea is that in each generation life will populate, survive, or die based on certain rules. Read the Wikipedia article (http://en.wikipedia.org/wiki/Conway's_Game_of_Life) to learn more about this famous simulation.

The universe of the Game of Life is a two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbors, the immediately adjacent cells in orthogonal and diagonal direction. Cells on the border have less than eight neighbors. At each generation, the following transitions occur:

(1) any live cell with fewer than two live neighbors dies, as if by loneliness.
(2) any live cell with more than three live neighbors dies, as if by overcrowding.
(3) any live cell with two or three live neighbors lives, unchanged, to the next generation.
(4) any dead cell with exactly three live neighbors comes to life.

The following figures show the grid examples for the first generation, second generation, and third generation, respectively.

		1		1		
			1			
			1			
		1		1		

First generation

			1			
		1	1	1		
		1	1	1		
			1			

Second generation

		1	1	1		
		1	1	1		

Third generation

Your task is to write a program to simulate the Game of Life for a 20×20 world. The 20×20 world needs to be initialized by reading in which cells are occupied from the user. Your program will generate and display the next generation iteratively based on the set of rules described above. The number of generation for the simulation is also input by the user.

■ *AUTOLAB Submission Check:*

```
int answer1;    // Store the total number of live cells of the first generation
int answer2;    // Store the total number of live cells of the second generation
int answer3;    // Store the total number of live cells of the third generation
```

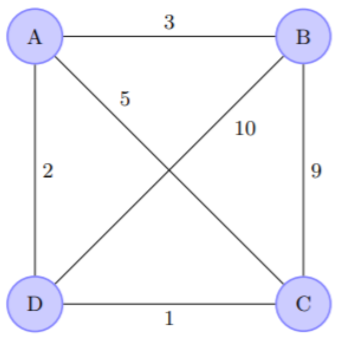
Program C： The Traveling Salesman Problem (40%)

The traveling salesman problem involves a salesman who must make a tour of a number of cities using the shortest path available and visit each city exactly once and return to the original starting city. The Naïve, or brute-force, approach computes and compares all possible permutations of paths to discover the shortest unique solution. Given *N* possible cities, with every city connected by a path to every other city, the number of paths which must be explored is (*N*-1)!.

The brute-force method is to simply generate all possible tours and compute their distances. The shortest tour is thus the optimal tour. We can use the following steps to find the shortest tour:

(1) calculate the total number of tours.
(2) list all the possible tours.
(3) calculate the distance of each tour.
(4) choose the shortest tour as the optimal solution.

Write a program to find the shortest tour of the following 4-city case. The program outputs the traveling sequence and the shortest distance of the optimal tour on the screen.



■ *AUTOLAB Submission Check:*

int answer1; // Store the shortest distance of the tour

Optional Bonus Points (20%):

You are encouraged to challenge a different algorithm using Nearest Neighbor method to find an approximate solution of the shortest tour. The key to this algorithm is to always visit the nearest city, then return to the starting city when all the other cities are visited. The steps of this method is as follows:

(1) select a starting city.
(2) find the nearest unvisited city and go there.
(3) are there any unvisited cities left? If yes, repeat step 2.
(4) return to the first city and sum the distance of this tour.
(5) repeat step 1 to step 4 for all cities, as a starting city, and find the shortest tour among all tours.

■ *AUTOLAB Submission Check:*

```
int answer1;     // Store the shortest distance of the tour
```

Notes:

1. Please submit your programs (source codes) to the AUTOLAB grading system website (http://140.112.183.225) before **Dec. 10** (3:30PM)

2. Late submission will have a penalty of 10% discount per day of your homework total score toward a maximum of 50% discount. No late submission over five days will be accepted.

3. Criteria of grading include: (1) Program functionality; (2) User interface; (3) Structure of the program; (4) Suitable comments; (5) Programming style; (6) Creativity.