CS1010E: Programming Methodology

Assessed Lab 6 (Make-Up): Struct [12%]

19 Apr 2017

Instructions

Please read all the instructions very carefully!

1. This is an **Open Book** assessment:
   - You are allowed to bring any printed materials and calculator
   - You are NOT allowed to use other electronic devices besides the lab's computer
   - You are NOT allowed to talk with your friends, to talk with invigilators please raise your hand
   - You are NOT allowed to access the internet except to the `plab` server via `SSH terminal`
2. This lab assessment consists of **one (1)** problems with several tasks:
   - The tasks are intended to guide you in solving the problem
   - Each task should have **its own separate file** where the task number is written at the back: `task3.c` is used for task 3
   - To proceed to the next level (*e.g., from task 2 to task 3*), copy your program using the command `cp task2.c task3.c`
   - Fill in your **Name**, **Matric** (*starts with* `A`), and **NUSNET ID** (*starts with either* `A` *or* `E`)
3. Numerical and precision guides:
   - **Two (2)** types of *input* numbers: `real` (*may have decimal point*) and `integer` (*no decimal point*)
   - `integer` may contain leading *zeroes*: always use `scanf("%d")` to ensure *decimal* representation
   - `integer` has a range of -2^{31} to $+2^{31}-1$, `unsigned integer` has a range of 0 to $+2^{32}-1$
   - Always use `double` for `real` number input for high precision, but numbers that differs by less than `0.001` are considered *equal*
4. Starting the tests:
   - Use the program `SSH Secure Shell Client`
   - Login to `plab` server using the given `username` and `password`
5. Testing and debugging guides:
   - You may open **two (2)** or more `SSH Terminal`: 1 for *coding* and 1 for *compilation + testing*
   - Assumption stated in the task is considered to always hold and no checking is necessary
   - Assumption NOT stated in the task will be tested in hidden input: *always think of worst case*
   - Test case outputs are organized by task number and test case number:
     - Task number `T` on test case number `C` have output file `testT_C.out`
     - *For example*: task number 2 with test case number 3 have output file `test2_3.out`
   - Test case inputs are the same for all tasks: *e.g.,* `test2.in`
6. Marking:
   - Grading is done *automatically* using CodeCrunch: only the largest correct task is considered
   - For instance: Task 1 is *empty (i.e., not done at all)*, Task 2 is *correct*, Task 3 is *incorrect*
     ↦ mark for Task 2 is taken
   - The mark for each task is given on the right side, it is a *cumulative* mark
7. Time management suggestion: [Total Time: **1 hour 30 minutes**]:
   - *Coding*: approx. **1 hour** (±**30 minutes** for debugging)
   - *Ending*: approx. last **5 minutes** ensures that you save the filename correctly

Hey Carrol [12 %]

Problem Description

Carrol is a robot that lives in a town where the houses are arranged in nice blocks similar to Manhattan, New York. The aerial view of the town can be simplified to look like Diagram 1. The road going from north-south (or south-north, depending on your point-of-view) is called Street and the road going from east-west is called Avenue. Every Street and Avenue are numbered starting from 0 with Carrol living at a house at the corner of Street #0 and Avenue #0.

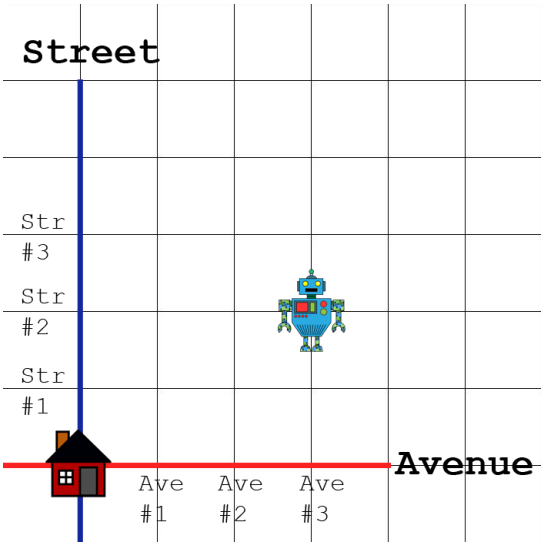

Diagram 1: Example town with position of Carrol at Street #2 and Avenue #3 (*i.e., Point (2, 3)*).

Carrol wants to know how many path are there from her current position to her house *assuming she can only move left and down*. In Diagram 1, there are **ten (10)** ways for Carrol to go from her current position at Point (2, 3) to her house at Point (0, 0). However, due to heavy construction, certain points and roads in the town are off-limit. Diagram 2 shows a presence of a rock at Point (1, 1). Since Carrol cannot move diagonally, she cannot go from Point (2,1) to Point (0,1) or any other path that passes through Point (1, 1). Therefore, there are only **four (4)** ways for Carrol to reach her house.

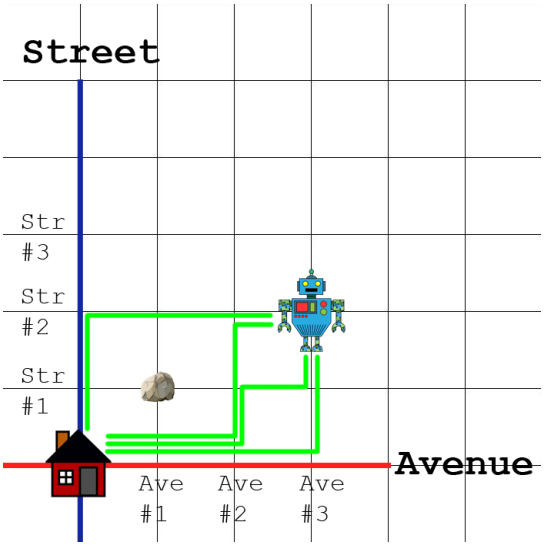

Diagram 2: Example town with position of Carrol at Point (2, 3) with obstacle at Point (1, 1).

In Diagram 3, the path between two points is blocked instead. In this case, the path between Point (1, 2) and Point (1, 1) is blocked. Therefore, from Point (1, 2), she cannot go to Point (1, 1). Thus, there are only **six (6)** ways for Carrol to reach her house.

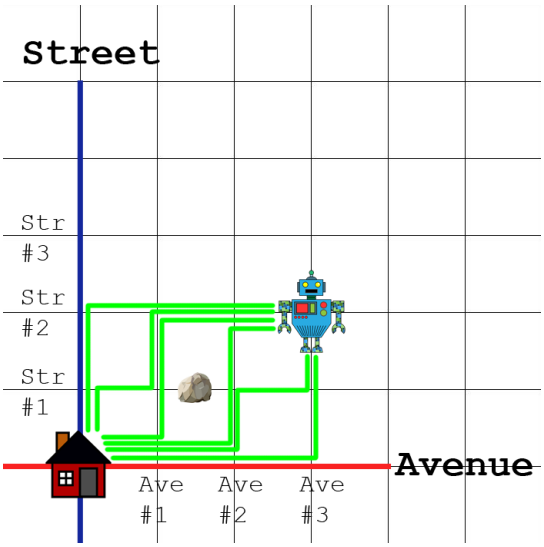

Diagram 3: Example town with position of Carrol at Point (2, 3) with obstacle between Point (1, 1) and Point (1, 2).

The blockade can be represented as a **struct** containing at least the following information:

- **Type:** the type of blockade, either point blockade or path blockade
- **Coordinate #1:** if type is point blockade, this represents the Point (x, y) where the blockade is located
- **Coordinate #2:** if type is path blockade, together with *Coordinate #1* this represents that the path between *Coordinate #1* and *Coordinate #2* is blocked

However, you may use *other representation* that better suits you. The above **struct** representation is merely an incomplete suggestion to simplify your problem.

Given the **struct** above, the map can be represented as an array of such **struct**. This, together with the coordinate of Carrol, can be used to find the number of ways from Carrol's current position to her house at Point (0, 0).

Final Objective

Given Carrol's current position at index (x, y) and a list of blockades, find the number of ways to her house.

Example

Consider Diagram 1. The input to this problem can be given as:

```
2 3 | Carrol position at Point (2, 3)
0   | Number of blockades
```

The output of this problem is printed as:

```
10↩
```

Consider Diagram 2. The input to this problem can be given as:

```
2 3   | Carrol position at Point (2, 3)
1     | NUmber of blockades
1 1 1 | #1: type 1 = point blockade. location at Point (1, 1)
```

The output of this problem is printed as:

```
4↩
```

Consider Diagram 3. The input to this problem can be given as **either**:

```
2 3       | Carrol position at Point (2, 3)
1         | Number of blockades
2 1 1 1 2 | #1: type 2 = path blockade. from Point (1, 1) to Point (1, 2)
```

OR

```
2 3       | Carrol position at Point (2, 3)
1         | Number of blockades
2 1 2 1 1 | #1: type 2 = path blockade. from Point (1, 2) to Point (1, 1)
```

The output of this problem is printed as:

```
6↩
```

NOTE: the test cases are arranged in such a way that the first test case (*i.e.,* `test1.in`) is from the first sample run, the second test case (*i.e.,* `test2.in`) has no blockade, the third test case (*i.e.,* `test3.in`) uses only Point blockade, and the fourth test case (*i.e.,* `test4.in`) uses only Path blockade. You may use this knowledge for a more thorough checks and debugs.

Additionally, you are given a template file. You can follow the template and add your code inside or rewrite all and erase the parts you do not need.

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ $0 \leq$ `x` ≤ 100 (*the Street number of Carrol's position*)
- ▷ $0 \leq$ `y` ≤ 100 (*the Avenue number of Carrol's position*)
- ▷ $0 \leq$ `N` ≤ 1000 (*the number of obstacle in town*)
- ▷ $1 \leq$ `type` ≤ 2 (*the type of blockade*)
- ▷ $0 \leq$ `block_str` ≤ 100 (*the Street number of a blockade*)
- ▷ $0 \leq$ `block_ave` ≤ 100 (*the Avenue number of a blockade*)
- ▷ There is no rock at Point (0, 0) and Carrol's initial position
- ▷ Carrol's initial position will not be at Point (0, 0)
- ▷ Path blockade will give adjacent points, it will not block multiple paths

Tasks

The problem is split into 5 tasks with 4 number of testcases given. In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.
- If the test(s) give(s) **NO** message(s), it means your program is correct.

Task 1/5: ***[Input/Output]*** **[1%]**

Write a program that reads Carrol's position and the position of the blockades and prints the position of the blockade line-by-line. The input is given as multiple lines.

- The first line is **two (2)** `integer` value `S` and `A` indicating Carrol's position at Point (`S`, `A`)
- The next line is **one (1)** `integer` `N` indicating the number of obstacles
- The next `N` lines are of the following format:
  - The first `integer` number `type` indicates the type of blockade
  - If `type` is `1`: a Point blockade
    - The next **two (2)** `integer` numbers indicates the position of the blockade
  - If `type` is `2`: a Path blockade
    - The next **two (2)** `integer` numbers indicates the position X
    - The next **two (2)** `integer` numbers indicates the position Y
    - Such that there can be no movement from X to Y **AND** from Y to X

Sample Run:

Inputs:

```
2 3          | Carrol's position
2            | Number of blockade
2 1 2 1 3    | #1: type 2 (1, 2)--(1, 3)
1 1 1        | #2: type 1 at (1, 1)
```

Outputs:

```
1 2 1 3↩ | type 2 between (1, 2)--(1,3)
1 1↩     | type 1 at (1, 1)
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
1            | Number of blockade
1 2 2        | #1: type 1 at (2, 2)
```

Outputs:

```
2 2↩
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
1            | Number of blockade
2 2 3 1 3    | #1: type 2 (2, 3)--(1, 3)
```

Outputs:

```
2 3 1 3↩
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
2            | Number of blockade
1 2 2        | #1: type 1 at (2, 2)
2 2 3 1 3    | #2: type 2 (2, 3)--(1, 3)
```

Outputs:

```
2 2↩
2 3 1 3↩
```

Save your program in the file named `carrol1.c`. No submission is necessary.
Test your program using the following command(s):

```
./a.out < test1.in | diff - test1_1.out
./a.out < test2.in | diff - test1_2.out
./a.out < test3.in | diff - test1_3.out
./a.out < test4.in | diff - test1_4.out
```

To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp carrol1.c carrol2.c
```

Task 2/5: ***[Baseline]*** **[3%]**

Write a program that reads Carrol's position and the position of the blockades and prints the number of ways for Carrol to go home without considering the obstacles. The input is given as multiple lines.

- The first line is **two (2)** `integer` value `S` and `A` indicating Carrol's position at Point (`S`, `A`)
- The next line is **one (1)** `integer` `N` indicating the number of obstacles
- The next `N` lines are of the following format:
  - The first `integer` number `type` indicates the type of blockade
  - If `type` is `1`: a Point blockade
    - The next **two (2)** `integer` numbers indicates the position of the blockade
  - If `type` is `2`: a Path blockade
    - The next **two (2)** `integer` numbers indicates the position X
    - The next **two (2)** `integer` numbers indicates the position Y
    - Such that there can be no movement from X to Y **AND** from Y to X

Sample Run:

Inputs:

```
2 3         | Carrol's position
2           | Number of blockade
2 1 2 1 3   | #1: type 2 (1, 2)--(1, 3)
1 1 1       | #2: type 1 at (1, 1)
```

Outputs:

```
10↩
```

Sample Run:

Inputs:

```
2 3         | Carrol's position
1           | Number of blockade
1 2 2       | #1: type 1 at (2, 2)
```

Outputs:

```
10↩
```

Sample Run:

Inputs:

```
2 3         | Carrol's position
1           | Number of blockade
2 2 3 1 3   | #1: type 2 (2, 3)--(1, 3)
```

Outputs:

```
10↩
```

Sample Run:

Inputs:

```
2 3         | Carrol's position
2           | Number of blockade
1 2 2       | #1: type 1 at (2, 2)
2 2 3 1 3   | #2: type 2 (2, 3)--(1, 3)
```

Outputs:

```
10↩
```

Save your program in the file named `carrol2.c`. No submission is necessary.
Test your program using the following command(s):

```
./a.out < test1.in | diff - test2_1.out
./a.out < test2.in | diff - test2_2.out
./a.out < test3.in | diff - test2_3.out
./a.out < test4.in | diff - test2_4.out
```

To proceed to the next task (*e.g., task 3*), copy your program using the following command:

```
cp carrol2.c carrol3.c
```

Task 3/5: ***[Checks]*** **[7%]**

Write a program that reads Carrol's position and the position of the blockades and prints if Carrol can move *left*, *down*, *both*, or *none*. The input is given as multiple lines.

- The first line is **two (2)** `integer` value `S` and `A` indicating Carrol's position at Point (`S`, `A`)
- The next line is **one (1)** `integer` `N` indicating the number of obstacles
- The next `N` lines are of the following format:
  - The first `integer` number `type` indicates the type of blockade
  - If `type` is `1`: a Point blockade
    - The next **two (2)** `integer` numbers indicates the position of the blockade
  - If `type` is `2`: a Path blockade
    - The next **two (2)** `integer` numbers indicates the position X
    - The next **two (2)** `integer` numbers indicates the position Y
    - Such that there can be no movement from X to Y **AND** from Y to X

Sample Run:

Inputs:

```
2 3          | Carrol's position
2            | Number of blockade
2 1 2 1 3    | #1: type 2 (1, 2)--(1, 3)
1 1 1        | #2: type 1 at (1, 1)
```

Outputs:

```
both↩ | Carrol can move both ways
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
1            | Number of blockade
1 2 2        | #1: type 1 at (2, 2)
```

Outputs:

```
down↩ | Carrol can only move down
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
1            | Number of blockade
2 2 3 1 3    | #1: type 2 (2, 3)--(1, 3)
```

Outputs:

```
left↩ | Carrol can only move left
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
2            | Number of blockade
1 2 2        | #1: type 1 at (2, 2)
2 2 3 1 3    | #2: type 2 (2, 3)--(1, 3)
```

Outputs:

```
none↩ | Carrol cannot move at all
```

Save your program in the file named `carrol3.c`. No submission is necessary.
Test your program using the following command(s):

```
./a.out < test1.in | diff - test3_1.out
./a.out < test2.in | diff - test3_2.out
./a.out < test3.in | diff - test3_3.out
./a.out < test4.in | diff - test3_4.out
```

To proceed to the next task (*e.g., task 4*), copy your program using the following command:

```
cp carrol3.c carrol4.c
```

Task 4/5: ***[Point Blockade]*** **[10%]**

Write a program that reads Carrol's position and the position of the blockades and prints the number of ways for Carrol to go home while considering **ONLY** Point blockades and ignoring Path blockades. The input is given as multiple lines.

- The first line is **two (2)** `integer` value `S` and `A` indicating Carrol's position at Point (`S`, `A`)
- The next line is **one (1)** `integer` `N` indicating the number of obstacles
- The next `N` lines are of the following format:
  - The first `integer` number `type` indicates the type of blockade
  - If `type` is `1`: a Point blockade
    - The next **two (2)** `integer` numbers indicates the position of the blockade
  - If `type` is `2`: a Path blockade
    - The next **two (2)** `integer` numbers indicates the position X
    - The next **two (2)** `integer` numbers indicates the position Y
    - Such that there can be no movement from X to Y **AND** from Y to X

HINT:

- Check if the Point you are going to go is blocked
- ***What is the actual base case?***

Sample Run:

Inputs:

```
2 3         | Carrol's position
2           | Number of blockade
2 1 2 1 3   | #1: type 2 (1, 2)--(1, 3)
1 1 1       | #2: type 1 at (1, 1)
```

Outputs:

```
4↩
```

Sample Run:

Inputs:

```
2 3         | Carrol's position
1           | Number of blockade
1 2 2       | #1: type 1 at (2, 2)
```

Outputs:

```
4↩
```

Sample Run:

Inputs:

```
2 3         | Carrol's position
1           | Number of blockade
2 2 3 1 3   | #1: type 2 (2, 3)--(1, 3)
```

Outputs:

```
10↩
```

Sample Run:

Inputs:

```
2 3         | Carrol's position
2           | Number of blockade
1 2 2       | #1: type 1 at (2, 2)
2 2 3 1 3   | #2: type 2 (2, 3)--(1, 3)
```

Outputs:

```
4↩
```

Save your program in the file named `carrol4.c`. No submission is necessary.

Test your program using the following command(s):

```
./a.out < test1.in | diff - test4_1.out
./a.out < test2.in | diff - test4_2.out
./a.out < test3.in | diff - test4_3.out
./a.out < test4.in | diff - test4_4.out
```

To proceed to the next task (*e.g., task 5*), copy your program using the following command:

```
cp carrol4.c carrol5.c
```

Task 5/5: ***[Path Blockade]*** **[12%]**

Write a program that reads Carrol's position and the position of the blockades and prints the number of ways for Carrol to go home while considering **BOTH** Point and Path blockades. The input is given as multiple lines.

- The first line is **two (2)** `integer` value `S` and `A` indicating Carrol's position at Point (`S`, `A`)
- The next line is **one (1)** `integer` `N` indicating the number of obstacles
- The next `N` lines are of the following format:
  - The first `integer` number `type` indicates the type of blockade
  - If `type` is `1`: a Point blockade
    - The next **two (2)** `integer` numbers indicates the position of the blockade
  - If `type` is `2`: a Path blockade
    - The next **two (2)** `integer` numbers indicates the position X
    - The next **two (2)** `integer` numbers indicates the position Y
    - Such that there can be no movement from X to Y **AND** from Y to X

HINT:

- Check if the Point you are going to go is blocked, from the current Point
- ***What is the actual base case?***

Sample Run:

Inputs:

```
2 3          | Carrol's position
2            | Number of blockade
2 1 2 1 3    | #1: type 2 (1, 2)--(1, 3)
1 1 1        | #2: type 1 at (1, 1)
```

Outputs:

```
3
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
1            | Number of blockade
1 2 2        | #1: type 1 at (2, 2)
```

Outputs:

```
4
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
1            | Number of blockade
2 2 3 1 3    | #1: type 2 (2, 3)--(1, 3)
```

Outputs:

```
6
```

Sample Run:

Inputs:

```
2 3          | Carrol's position
2            | Number of blockade
1 2 2        | #1: type 1 at (2, 2)
2 2 3 1 3    | #2: type 2 (2, 3)--(1, 3)
```

Outputs:

```
0 | Carrol cannot move at all
```

Save your program in the file named `carrol5.c`. No submission is necessary.

Test your program using the following command:

```
./a.out < test1.in | diff - test5_1.out
./a.out < test2.in | diff - test5_2.out
./a.out < test3.in | diff - test5_3.out
./a.out < test4.in | diff - test5_4.out
```

Useful VIM and SSH Terminal Commands

- **VIM Mode Switch:**
  - **i** i nsert (*from* Command)
  - **esc** esc ape to Command
- **Basic VIM Commands:** [mode=Command]
  - **:w** w rite file
  - **:q** q uit file
  - **:q!** q uit file (*forced*: *without saving*)
  - **:wq** w rite and q uit
- **Advanced VIM Commands:** [mode=Command]
  - **/text** find text
  - **n** find n ext text
  - **shift + n** find previous text
  - **gg=G** auto-indentation all lines
- **VIM Text Edit Commands:** [mode=Command]
  - **dd** d elete line at cursor (*cut*)
  - **yy** y ank line at cursor (*copy*)
  - **p** p aste after current cursor
  - **u** u ndo one change
  - **x** cut one character at cursor
  - **:red** red o undone changes
  - *N* **dd** d elete N lines down (N is number)
  - *N* **yy** y ank N lines down (N is number)
- **VIM Auto-Completion:** [mode=Insert]
  - **ctrl + n** complete word
  - **ctrl + x** complete line
- **Basic SSH Terminal Commands:**
  - **cd dir** open folder dir
  - **cd ..** open *parent* folder
  - **rm file** remove file file
  - **rm -r dir** remove folder dir
  - **vim file** open file in VIM
  - **ls** list files in folder
  - **ls -all** list ALL files in folder
  - **cat file** open *small* text file
  - **less -e file** open *large* text file
  - **cp f1 f2** copy f1 to f2
  - **mv f1 f2** move f1 to f2
    (*in effect, rename if in same folder*)
- **Execute Your Program in SSH Terminal :**
  - **gcc -Wall file** compile file
  - **gcc -Wall -lm file**
    compile file with math library (i.e. #define <math.h>) included
  - **./a.out** run program
  - **gcc -Wall file -o f1**
    compile file and rename executable into f1 (run using ./f1)
- **Advanced Program Execution Commands in SSH Terminal :**
  - **./a.out < f_in**
    run program with input redirection from file located at f_in
    (e.g. **./a.out < test1.in**)
  - **./a.out < f_in > f_out**
    run program with input redirection from file located at f_in and redirect the output to write into (*non-existing*) file called f_out
    (e.g. **./a.out < test1.in > output1**)
  - **diff f1 f2**
    compares the two files (f1 compared with f2) line by line (*note: no news is good news*)
    (e.g. **diff output1 test1_1.out**)
  - **./a.out < f_in | diff - f_out**
    run program with input from f_in immediately compare output with f_out
    (e.g. **./a.out < test1.in | diff - test3_1.out**)
- **SSH Terminal Emergency Commands:**
  - *Infinite loop* press **ctrl + c**
  - *End input* press **ctrl + d**
  (*better way is to use input redirection*)
- **VIM DO NOT DO LIST**
  - **ctrl + z** move to background
    (if done, type **fg** into **SSH Terminal**)
  - **ctrl + s** suspend
    (if done, press ctrl+q)
  - *Close without using* **:q**
    * on reopen, .swp file created
    * open file, choose Recover & exit VIM
    * open file again & choose Delete
- **GCC DO NOT DO LIST**
  - **gcc file -o file**
    compile file and rename into file (now, file is no longer a C program file)
    * ***pray hard...***
    * look for .file.history by typing **ls -all**
    * copy to windows using **SSH File Transfer**
    * ***hope*** latest code is at *end of file*