Title:	Methods and Tools for Software Engineering
Course ID:	ECE 650 Section 001
LEARN:	`https://learn.uwaterloo.ca`
Piazza:	`https://piazza.com/uwaterloo.ca/fall2024/ece650/home`
Lectures:	TTh 5:30 – 6:50 PM EST
Instructor:	Dr. Albert Wasef, `awasef@uwaterloo.ca`
TA:	Josh Sun, `q84sun@uwaterloo.ca`

Office hours by appointment. Begin all email subjects with `[ECE650]`. Use **Piazza** instead of email whenever possible!

Assignment 1 - Due September 23th, 2024

The skeleton for this assignment is available at the master branch of `https://git.uwaterloo.ca/ece650-f24/skeleton` in directory `a1`. Follow the instructions in Assignment 0 to correctly fetch and merge the files from the skeleton!

Get assignment content

If you have not added the skeleton repo to your own repo, you can that by the following command:

```
git remote add upstream https://git.uwaterloo.ca/ece650-f24/skeleton
```

Start by fetching the remote repository

```
git fetch upstream
```

If you have not done so before, create a `master` branch:

```
git checkout -b master
```

Merge the new content into your local `master` branch. You will have to do this every time the skeleton repository is updated for new assignment.

```
git merge upstream/master --allow-unrelated-histories
```

This is the first in a series of assignments that is part of a single large project. The project is to help the local police department with their installation of security cameras at traffic intersections. You will solve a particular kind of optimization problem, called the Vertex Cover problem, in this context. The idea is for the police to be able to minimize the number of cameras they need to install, and still be as effective as possible with their monitoring.

For this assignment, you need to:

- Take as input a series of commands that describe streets.
- Use that input to construct a particular kind of undirected graph.
- Write your code in Python (version 3).
- Ensure that it works on `eceubuntu.uwaterloo.ca`. (You are allowed to use only those Python libraries that are already installed on those machines. You are not allowed to install any new libraries.) (Use `eceterm.uwaterloo.ca` to log-in from off-campus; then follow the instructions to connect to `eceubuntu`.)

Sample Input

The input comprises lines each of which specifies a command. There are 4 kinds of commands. (1) add a street, (2) modify a street, (3) remove a street, and, (4) generate a graph. Here is an example of how your program should work. Visualizing this example using the Cartesian coordinate system may help you understand what's going on.

```
add "Weber Street" (2,-1) (2,2) (5,5) (5,6) (3,8)
add "King Street S" (4,2) (4,8)
add "Davenport Road" (1,4) (5,8)
gg
V = {
  1:  (2,2)
  2:  (4,2)
  3:  (4,4)
  4:  (5,5)
  5:  (1,4)
  6:  (4,7)
  7:  (5,6)
  8:  (5,8)
  9:  (3,8)
  10: (4,8)
}
E = {
  <1,3>,
  <2,3>,
  <3,4>,
  <3,6>,
  <7,6>,
  <6,5>,
  <9,6>,
  <6,8>,
  <6,10>
}
mod "Weber Street" (2,1) (2,2)
gg
V = {
  2:  (4,2)
  5:  (1,4)
  6:  (4,7)
  8:  (5,8)
  10: (4,8)
}
E = {
  <2,6>,
  <6,5>,
  <6,8>,
  <6,10>
}
rm "King Street S"
gg
V = {
}
E = {
}
```

Commands

- `add` is used to add a street. It is specified as: "`add "Street Name" (`x_1`, `y_1`) (`x_2`, `y_2`) ...(`x_n`, `y_n`)`". Each `(`x_i`, `y_i`)` is a GPS coordinate. We interpret the coordinates as a poly-line segment. That is, we draw a line segment from `(`x_i`, `y_i`)` to `(`x_{i+1}`, `y_{i+1}`)`. You are allowed to assume that each x_i and y_i is an integer. (Note, however, that the coordinates of an intersection may not be integers.)
- `mod` is used to modify the specification of a street. Its format is the same as for `add`. It is a new specification for a street you've specified before.
- `rm` is used to remove a street. It is specified as "`rm "Street Name"`".
- `gg` causes the program to output the corresponding graph.

Input and Output

Your program should take input from standard input. Your program should output to the standard output. Error should be output to standard error. You can use exceptions in your code to catch errors.

Errors

The above example is that of a "perfect" user — someone that did not make any mistakes with specifying the input. You should account for errors in the input. If a line in the input is erroneous, you should immediately output an error message. The format of the message is to be the string "`Error:`" followed by a brief descriptive message about the error. For example:

```
Error: `mod' or `rm' specified for a street that does not exist.
```

Your program should recover from the error as well. That is, your program should reject the errorneous line, but continue to accept input. Your program should not crash because of an error. Any erroneous input we try will be of a relatively benign nature that mimics honest mistakes a user makes. We will not try malicious input, such as unduly long lines or weird control characters.

The Output Graph

There is a vertex corresponding to: (a) each intersection, and, (b) the end-point of a line segment of a street that intersects with another street. An example of (a) from above is Vertex 3. An example of (b) is Vertex 1. The identity of a vertex can be any string of letters or integers (but no special characters). For example, `v1xyz` is acceptable as the identity of a vertex, but not `v1 !!#xyz`. (The space is unacceptable, as are '`!`' and '`#`'.

There is an edge between two vertices if: (a) at least one of them is an intersection, (b) both lie on the same street, and, (c) one is reachable from the other without traversing another vertex. An example from above is the edge $\langle 1, 3\rangle$, which connects the end-point of a line segment to an intersection. Another example is the edge $\langle 3, 6\rangle$, which connects two intersections.

Marking

Your output has to perfectly match what is expected. You should also follow the submissions instructions carefully. The reason is that our marking is automated.

- Does not compile/make/crashes: automatic 0
- Your program runs, awaits input and does not crash on input: + 20
- Passes Test Case 1: + 20
- Passes Test Case 2: + 20
- Passes Test Case 3: + 15
- Correctly detects errors: + 25

- Programming style: + 0 for this assignment, but will be > 0 for future assignments.

Get Skeleton Code from Upstream

To get the code, remember to pull from `upstream/master`.

Python Resources

If you're looking for some practice using and understanding Python3 you may wish to use `http://pythontutor.com/`. There is one example one the main page that you can step through line-by-line. More examples can be found on the live online editor page (`http://pythontutor.com/visualize.html`). The generated visualization of the code may be useful to you. The python tutor webpage also has code visualization for C++which you may find useful for the next assignment.

If you have Python code on your computer that you wish to step through in a similar way, either line-by-line or some other way, you can run the code using the Python debugger. The Python debugger (pdb) can be invoked on the command line by providing the flag and argument as follows:

```
python3 -m pdb <YOUR-PYTHON-FILE.py>
```

Documentation on how to use the debugger can be found on `https://docs.python.org/3/library/pdb.html`.

Submission Instructions

Your code can be in multiple Python source files. The main python file should be `a1/a1ece650.py`. The assignment should be submitted as a `master` branch on YOUR GitLab in directory `a1`. Don't forget to enter your student identification in `a1/user.yml` and follow any additional instructions in `a1/README.md`.