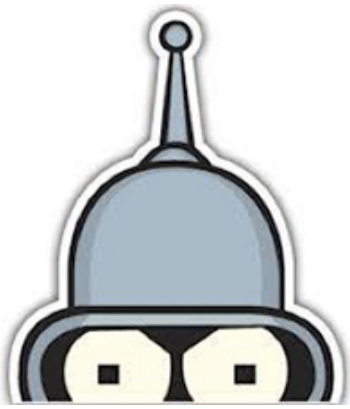
B2 - Elementary Programming in C

B-CPE-201

Lem-in

Ant-based calculation unit



1.0

{EPITECH.}

Lem-in

binary name:	lem_in
repository name:	CPE_lemin_$ACADEMICYEAR
repository rights:	ramassage-tek
language:	C
compilation:	via Makefile, including re, clean and fclean rules

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

For this project, the **only** authorized function are `read, write, malloc, free and getline.`

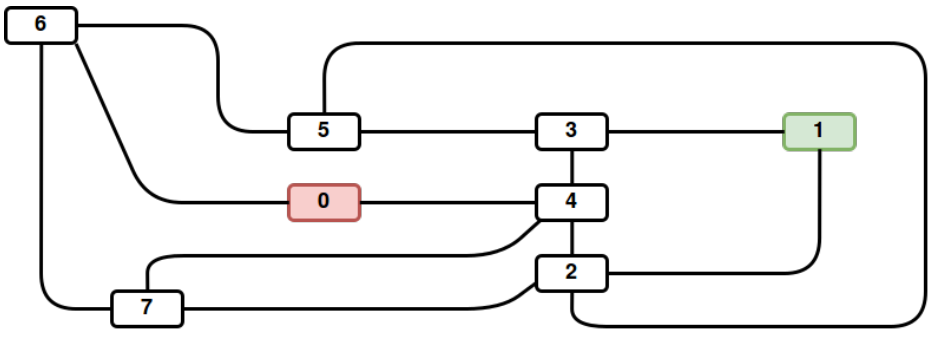
> *Hex is an elaborate, magic-powered and self-building computer (not unlike the 'shamble', a kind of magical device used by the Witches of the Discworld) and is housed in the basement of the High Energy Magic Building at the Unseen University (UU) in the twin city of Ankh-Morpork.*
>
> *Hex is a computer unlike any other. Programmed via 'Softlore', Hex runs and evolves under the watchful eyes of wizard Ponder Stibbons, who becomes the de facto IT manager at UU because he's the only one who understands what he's talking about.*
>
> *Hex has its origins in a device created by some student Wizards in the High Energy Magic building. In this form it was simply a complex network of glass tubes, containing ants. The wizards could then use punched cards to control which tubes the ants could crawl through, enabling it to perform simple mathematical functions.*
>
> *https://en.wikipedia.org/wiki/Hex_(Discworld)*

Let's pay close attention to its calculation unit: an anthill with rooms and tunnels. A room can be connected to an infinite number of other rooms by as many tunnels as needed, but a tunnel can only connect two rooms.
It would be nice to build one of our own, but since we're not really into DIY, let's make a high-tech version: a "Hex' calculation unit" simulator.

On the standard input, your program must receive the anthill description the following way: first, the number of ants, then the rooms and their position and finally the tunnels (defined by the rooms they connect). Here is an example of input and the associated anthill:

Terminal

```
~/B-CPE-201> cat input
34
##start
1 23 3
2 16 7
#comment
3 16 3
4 16 5
5 9 3
6 1 0
7 4 8
##end
0 9 5
0-4
0-6
1-3
4-3
5-2
3-5
#another comment
4-2
2-1
7-6
7-2
7-4
6-5
```



The names of the rooms won't necessarily be numbers in order, and are less likely to be uninterrupted (you could even have rooms named: zdfg, 256, qwerty, ...)

The rooms' coordinates will always be whole numbers. Please note that it is possible to insert comments by using "#" and commands by using "##".
##start indicates the next room is the anthill entrance, and ##end indicates the next room is the anthill exit.
Any unknown commands will be ignored.
The objective of your program is to find the quickest way to make the ants cross over the anthill. To do so, each single ant need to take the shortest route (and not necessarily the easiest), whithout walking on its peers, and avoiding traffic jams.

At the beginning of the game, all of the ants are in the anthill entrance.
The goal is to lead them to the exit room, in a minimum amount of laps.
Each room could contain a single ant at a time (except ##start and ##end, which can contain as many as needed).
With each lap, you can move each ant only once by following a tunnel (if the receiving room is clear).

You must show the result on the standard output, in this order: number_of_ants, rooms, tunnels and then for each lap, a series of **Pn-r** where **n** is the number of the ant, and **r** the name of the room it gets into.
In the output, you must display a comment indicating the part that will follow. These comment must be exactly like in the examples.

The first noncompliant or empty line will lead to the end of the anthill acquisition, as well as its normal treatment with the data that has already been acquired.

It's not as easy as snapping your fingers. Concerning what type of operation students of magic can perform with such a computer, all we know, today, is that electricity is more dependable.

EXAMPLES

Concerning the following trivial anthill:

```
[0] - - - [2] - - - [3] - - - [1]
```

Terminal

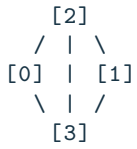
```
~/B-CPE-201> cat anthill
3
##start
0 1 0
##end
1 13 0 #bedroom
2 5 0
# The next room is the kitchen
3 9 0
0-2
2-3
3-1
```

Terminal

```
~/B-CPE-201> ./lem_in < anthill
#number_of_ants
3
#rooms
##start
0 1 0
##end
1 13 0
2 5 0
3 9 0
#tunnels
0-2
2-3
3-1
#moves
P1-2
P1-3 P2-2
P1-1 P2-3 P3-2
P2-1 P3-3
P3-1
```

There you have it, it's over after 5 laps.

Considering the following anthill:



Terminal

```
~/B-CPE-201> cat anthill
3
2 5 0
##start
0 1 2
##end
1 9 2
3 5 4
0-2
0-3
2-1
3-1
2-3
```

Terminal

```
~/B-CPE-201> ./lem_in < anthill
#number_of_ants
3
#rooms
2 5 0
##start
0 1 2
##end
1 9 2
3 5 4
#tunnels
0-2
0-3
2-1
3-1
2-3
#moves
P1-2 P2-3
P1-1 P2-1 P3-2
P3-1
```

It's over after 3 laps.

BONUS

For a bonus, why not code an anthill viewer?
Whether in two dimensions (an overhead view) or from an ant's point of view in a 3D corridor environment (using the Oculus Rift?).
Please note that because the commands and comments are repeated on the output of the program, it is possible to pass specific commands to the viewer (why not colors, levels,...?).
An example of usage may be:

```
▽                          Terminal                          - + x
~/B-CPE-201> ./lem_in < map | ./viewer
```