CS 214 Homework 1

Jeff Ames

Fall 2022

Introduction

This assignment will give you some initial experience with programming in C. Your task is to write the following C programs.

- Please be careful to follow the I/O formats exactly.
- You can assume that inputs will be correctly formatted.

You can work in small groups (1 – 3 people). Please include a README file with your code that contains all partners' names and netIDs. Only one person from each group should submit the assignment.

factor.c

Write a program that takes a single integer n as a command-line argument and finds its prime factors, excluding 1. The output should consist of a single line listing each prime factor in non-decreasing order, separated by a space. If n is prime, just print n itself. If there are repeat factors, print the factor as many times as it divides n. You can assume that $2 \leq n \leq 2^{31} - 1$ (i.e., n will fit in a signed 32-bit integer).

```
./factor 247
13 19

./factor 7
7

./factor 32
2 2 2 2 2
```

grep.c

Write a program that takes a string as a command-line argument and then reads from stdin until the user closes stdin (with ctrl-d, written `^D` below). It should print every line that contains the given string. It should be case-sensitive by default, but case-insensitive if the command line option `-i` is given before the string. Strings may have arbitrary length.

```
./grep ello
Hello there        # input
Hello there        # output
Hi there           # input, ignored since it doesn't contain "ello"
The sun is yellow  # input
The sun is yellow  # output
YELLOW             # input, ignored since "ello" is case-sensitive
^D                 # close stdin

./grep -i ello
Hello there        # input
Hello there        # output
YELLOW             # input
YELLOW             # output
^D                 # close stdin
```

sort.c

Write a program that reads from stdin and sorts its input. If no command-line option is given, it should use case-insensitive lexicographic sorting. If the `-n` option is given, it should use numeric sorting (in this case, you can assume all inputs will be valid integers). You can assume there won't be any spaces in the input string. There can be an arbitrary number of strings, of arbitrary length.

```
./sort
juice
aPPLES
donuts
Banana
^D    # close stdin
aPPLES
Banana
donuts
juice
```

```
./sort -n
5
-3
2
7
^D    # close stdin
-3
2
5
7

./sort
274
10101
123
10
^D    # close stdin
10
10101
123
274
```

uniq.c

Write a program that reads from stdin and filters out duplicate lines of input. It should read lines from stdin, and then print each unique line along with a count of how many times it appeared. Input is case-sensitive, so "hello" and "Hello" are not duplicates. There can be an arbitrary number of strings, of arbitrary length.

Note that only adjacent duplicates count, so if the input were the lines "hello", "world", and "hello" again, all three would be treated as unique.

```
./uniq
hello
hello
hello
world
world
^D    # close stdin
3 hello
2 world
```

```
./uniq
hello
world
hello
^D    # close stdin
1 hello
1 world
1 hello
```

monster.c

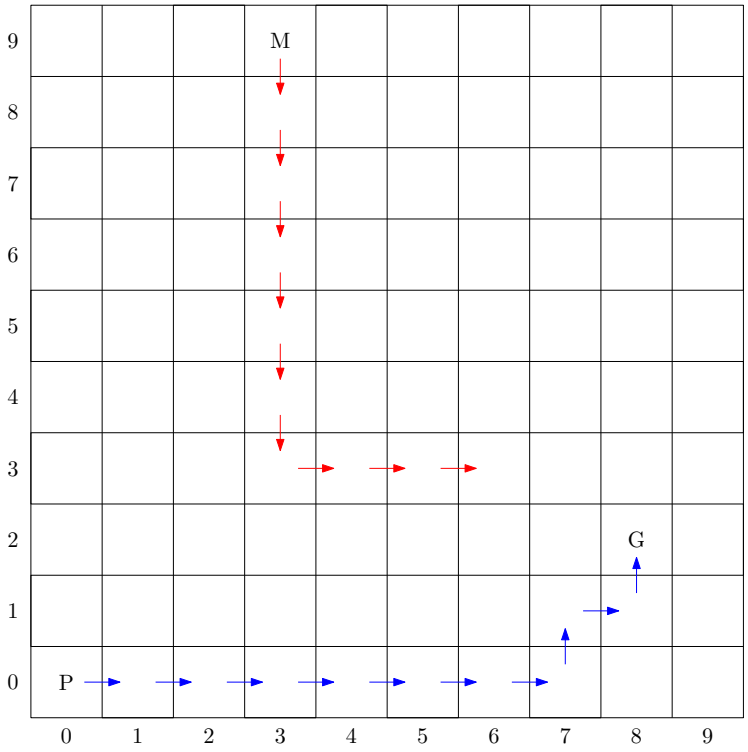
This program is a game of a player and monster moving on a 2D grid. The player has a goal to reach. The monster wants to reach the player.

In each turn, the player moves one square, then the monster moves towards the player. If either reaches their target, the game ends immediately. If the player reaches the goal, it prints "player wins!". If the monster reaches the player, it prints "monster wins!".

The player's moves (N, W, S, or E) are read from stdin, one per line. If the move is invalid (e.g., the player tries to move off the board, print "invalid move" and read a new move). If the player intentionally moves into the monster's square, it counts as the monster winning.

On its turn, the monster considers the horizontal and vertical distances to reach the player and moves to reduce the larger of these two. If they're the same, the monster chooses one of the two at random. Its move should be printed, e.g., "monster moves E". The monster cannot move onto the goal; if the monster's intended move would put it on the goal square, then the monster forfeits its move that turn.

For example, suppose we have a 10×10 grid where the player starts at $(0, 0)$ and chooses to move along the blue path to reach the goal at $(8, 2)$. The monster starts at $(3, 9)$ and its moves are shown in red. The player reaches the goal before the monster reaches the player, so the player wins.



The board is specified by command-line parameters:

```
./monster boardX boardY plrX plrY goalX goalY monX monY
```

`boardX`	Number of squares in x direction
`boardY`	Number of squares in y direction
`plrX`	Player x starting position
`plrY`	Player y starting position
`goalX`	Goal x location
`goalY`	Goal y location
`monX`	Monster x starting position
`monY`	Monster y starting position

At the beginning of the game, and after each round where no one won, the board should be printed as shown in the examples below. Each square should be separated by a space.

```
./monster 5 5 0 0 1 1 3 4
. . . M .
. . . . .
. . . . .
. G . . .
P . . . .
E
monster moves S
. . . . .
. . . M .
. . . . .
. G . . .
. P . . .
N
player wins!

./monster 4 6 0 0 2 4 1 1
. . . .
. . G .
. . . .
. . . .
. M . .
P . . .
E
monster moves S
monster wins!
```

Compiling and testing

You can compile your code simply with `gcc`, but adding some options, as in `gcc -g -Wall -Wvla -fsanitize=address`, may make debugging easier.

For any arrays, they should be give a constant size or, for dynamic arrays, created using `malloc` or similar. You should not use any variable-length arrays (VLAs). The `-Wvla` flag to `gcc` will warn if any VLAs are found.

To test some of these programs, it may be more convenient to put your input in a text file and then pipe it to your program:

`cat test.txt | ./myProgram` or `./myProgram < test.txt`

Submission

If you develop on your local machine, please be sure to test your code on ilab before submitting.

Please submit the assignment on Canvas as a tar file named `hw1.tar`. To create this file, put everything that you are submitting into a directory (folder) named `hw1`. Then, `cd` into the directory containing `hw1` (that is, `hw1`'s parent directory) and run the following command:

```
tar cvf hw1.tar hw1
```

To check that you have correctly created the tar file, you can copy it (`hw1.tar`) into an empty directory and run the following command:

```
tar xvf hw1.tar
```

This will re-expand your tar file and should create a directory named `hw1` in the (previously) empty directory with your code.