Assignment 3

DTU Compute
Technical University of Denmark
02102 Introductory Programming
14/04/2022

Group 34

Polly Clémentine Nielsen Boutet-Livoff - 214424

As I am the only member of the group, all the work was done by me.

1 Prime Factors

After running the program the CLI prompts the user for numbers until the user exits the application.

```
java PrimeFactors
Enter integer greater than 1 (0 to terminate): 1337
The prime factors of 1337 are 7, 191

Enter integer greater than 1 (0 to terminate): 420
The prime factors of 420 are 2, 2, 3, 5, 7

Enter integer greater than 1 (0 to terminate): 9223372036854775807
The prime factors of 9223372036854775807 are 7, 7, 73, 127, 337, 92737, 649657

Enter integer greater than 1 (0 to terminate): -1
I said greater than 1!!
Enter integer greater than 1 (0 to terminate): 0
Goodbye! <3
```

As can be seen in the example input, the program handles large inputs up to 2^32-1. However, larger values result in input parsing errors.

```
Enter integer greater than 1 (0 to terminate): 9223372036854775808
Exception in thread "main" java.util.InputMismatchException: For input
string: "9223372036854775808"
    at java.base/java.util.Scanner.nextLong(Scanner.java:2379)
    at java.base/java.util.Scanner.nextLong(Scanner.java:2328)
    at PrimeFactors.main(PrimeFactors.java:29)
```

In order to factorize a number n we check if any lower number divides n_0. If $a_0|n_0$ then we remove a_0 as a factor through division.

$$n_{i+1} = \frac{n_i}{a_i}$$

We then check for all numbers[1] starting from the lowest prime, 2. Eventually, if $n_i = 1$ we know we've found all prime factors. The program is pretty straight forward, you could extract some of the functionality (namely the factorization part) into a different method but the program is so short that I feel it isn't needed. There's not many other design choices to talk about it, it's pretty straight forward. An improvement you could make is to have better input parsing; Typing "0" to exit the program is very inelegant. Instead, we could parse the strings on our own and check if it was the string "exit", and if not try to parse it as a number. This would also allows us to have more informative errors, like pointing out what part of the input is invalid.

2 Random Walk

Random walk asks the user for a grid size (the width and height are actually twice what the user types in) and then runs a random walk. The program displays the positions the user is in as well as a rendering of the walk using StdDraw.

[1] We can optimise this by only checking using prime numbers

```
$ java RandomWalk
Enter size of grid: 5
Position = (0,0)
Position = (0,1)
Position = (0,0)
Position = (0,-1)
Position = (0,0)
Position = (0,1)
Position = (0,0)
Position = (0,1)
Position = (0,0)
Position = (1,0)
Position = (0,0)
Position = (1,0)
Position = (1,1)
Position = (1,2)
Position = (0,2)
Position = (-1,2)
Position = (-1,1)
Position = (-2,1)
Position = (-2,2)
Position = (-1,2)
Position = (-1,3)
Position = (0,3)
Position = (0,4)
Position = (0,5)
Position = (0,6)

Total number of steps = 24
```

The rendered path for this random walk can be seen in figure 1. For this problem I made the `RandomWalk` class have properties describing the size of the grid and the position in it, along with some methods to update and output those values. It's pretty OOP. The main function is very short, and mostly calls these methods on `RandomWalk`. An improvement could be to gradually change color for every step so we can more easily see where the path lingers. I could also unify the user interface into a GUI instead of using a CLI and a window.

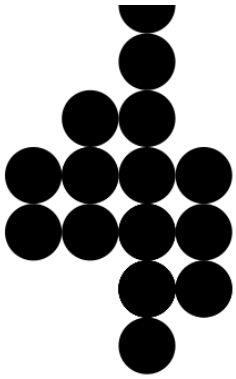

Figure 1: Random walk where $n = 5$

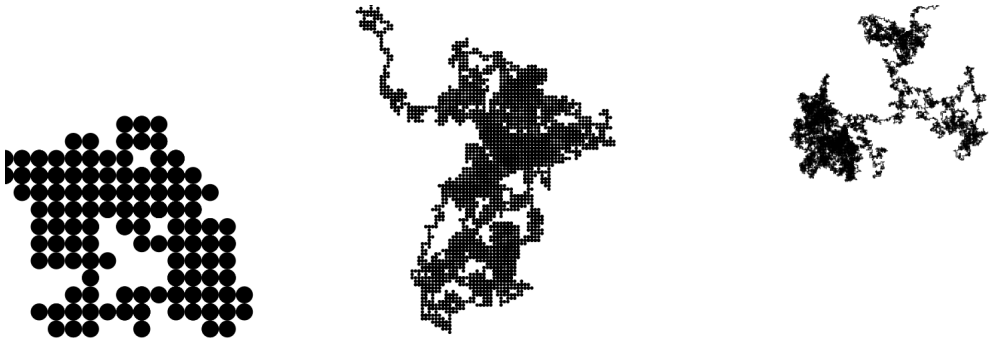

Figure 2: Additional random walks using the size 10, 50 and 100

3 Race Track

The program consists of a race track rendered using StdDraw and a CLI to accept user input.

```
$ java RaceTrack
```

7	8	9
4	5	6
1	2	3

```
Select the direction of acceleration: 6
```

7	8	9
4	5	6
1	2	3

```
Select the direction of acceleration: 3
```

7	8	9
4	5	6
1	2	3

```
Select the direction of acceleration: 2
You crashed! Better luck next time.
```

The `Vector` class is a fleshed out class designed around OOP paradigms. It's a vector that encapsulates its own coordinates and exposes useful methods to access and modify it. I don't use getters, as I find them unnecessary in this case, but if I wanted to go full OOP I would use those. I use the vector class to represent both position, velocity and acceleration for my car. I could have a class for the car and a class for the map but I think the code is easier to read the way it is. My main function does 2 things; first it renders the map using hardcoded values and functions and then it runs a game loop simulating the car. The map code is very boring, simply a bunch of hardcoded functions. The game loop consists of taking user input, computing the new position and velocity of the car, checking for collisions and checking if we crossed the finish line. The collision checks only check each position, not the transition between them. This means that it is possible to travel through the white area if one travels fast enough, especially around corners. The finish line only checks that you go from being on its left side to its right side, which means you can easily cheat by going over it counter-clockwise and then turning around and going clockwise. There are many things I could do to improve the game. I could add more cars (multiplayer, AI) or more levels, which would warrant creating a class for those. I could do more advanced collision checks that include the paths between positions. I could improve the graphics with sprites or pretty backgrounds. As this problem is very broad, the list of improvements is only bounded by

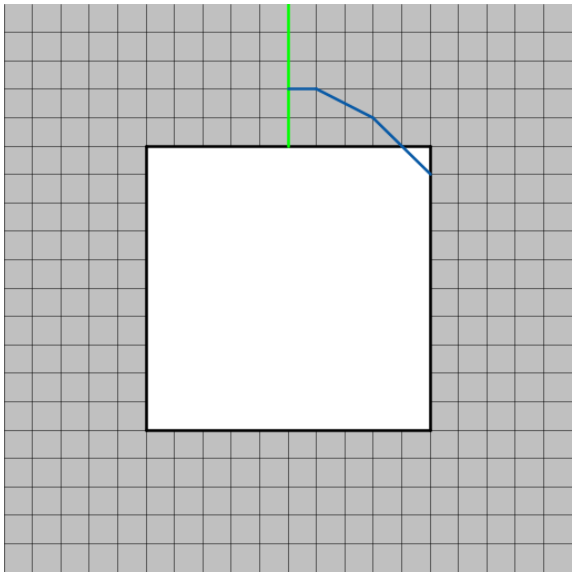

Figure 3: The travelled path when picking 6, 3 and 2 as your inputs.

the programmers imagination.

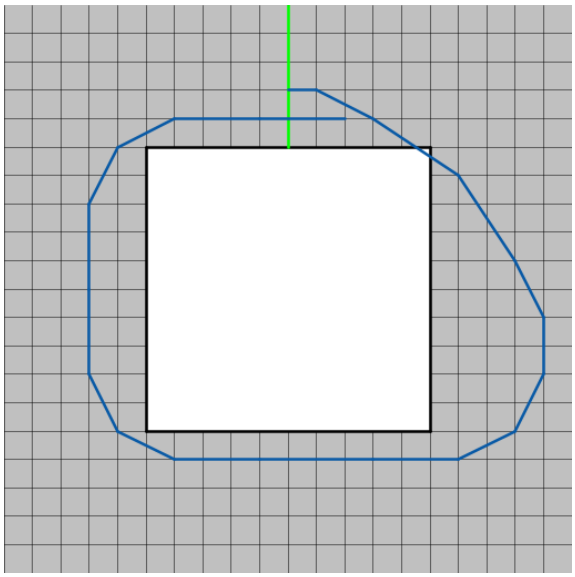

Figure 4: A complete lap in 19 moves using the moves 6331744774699953335.