Name: Lahoucine Chikry

ID: 137611

**The Report**

*The beginning of the algorithm*

After I read the problem, I had an initial idea about how the algorithm would look like. Since the robot always starts from an initial position that is (0, 0) and arrives at a position (***x-final****,* ***y-final***), that means I only need one function that takes three inputs: ***x-final***, ***y-final***, and a string of instructions. After that, the function will declare two variables and initialize them to zero, which will be the starting position. Then, I will do a *while* loop that increments or decrements the ***x-initial*** and ***y-initial*** by 1 according to each instruction using a *switch statement*. (see the figure in the next page)

The condition of the *while* loop was “while ***x-initial***and ***y-initial***are less than ***x-final*** and ***y-final***”. However, this condition doesn’t work in all cases. It just works in case 1 and case 3 from the technical challenge document, or any case that has a positive ***x-final***and ***y-final*** except if this positive case is an oscillating case (we will talk about this case later).

* Case 1 Input:
  + 3
  + 2
  + URR
* Output: Possible
* Case 3 Input:
  + 1
  + 2
  + RU
* Output: Impossible

Figure 1
As a result, I realized that this problem is all about knowing *how* to know *when* to stop implementing the instructions and declare the impossibility of reaching the final position if we haven’t reached it yet.

*The development of the algorithm*

By realizing the previous fact, I started to think about a condition that takes into account all the possible cases, positive and negative. The first thing I thought of is to know the sign of ***x-final*** and ***y-final*** separately first, then adjust the condition of the loop according to the signs of each component of the final position using *if-statements*. Hence, we will have 4 different loops depending on the signs of ***x-final*** and ***y-final****.*

Nevertheless, just after I said this thought to myself, I said that this would increase the complexity of the code and would reduce its readability since there would be a bunch of nested *if-statements* and different *loops* with different conditions. Furthermore, this algorithm will not work if we have the oscillation case I mentioned before, which I will talk about later, or the following case scenario.

* Input:
  + -2
  + 4
  + URR
* Output: Impossible

The reason is that the condition of the loop will be in this case scenario in my code “while ***x-initial***is greater than ***x-final*** && ***y-initial***is less than ***y-final***” and this will cause an infinite loop because although we have “-2”, the instructions are not going towards the final position.

So, after thinking passively about the algorithm, I had the idea that I should calculate the maximum number of steps needed to reach the final position, and that would be the condition that would end the loop if we hadn’t reached yet the final position.

*The final solution*

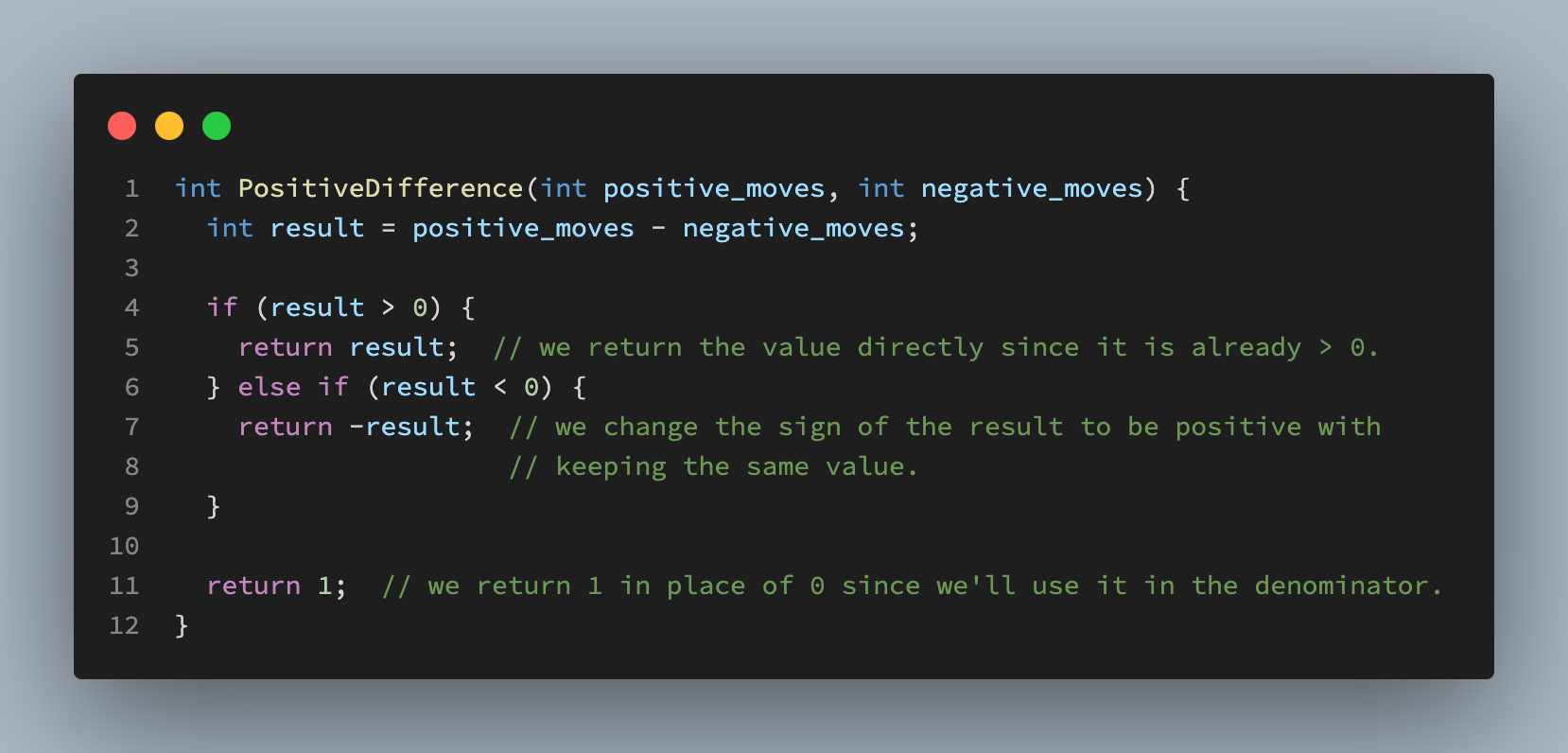
Therefore, I started to try to find a way to calculate the maximum number of times I needed to execute the set of given instructions, which represents the maximum steps needed to reach the final position. After pondering it for a while, I find it difficult to calculate it.

Thus, I switched from thinking about the problem as a whole to breaking it up into smaller problems. First, I said since the final position can be represented as a point in a two-dimensional plane (Cartesian coordinate system), we can first find how many steps are needed to reach the ***x-final***, then find how many steps are needed to reach the ***y-final****,* then sum them up. Therefore, I came up with the following equation that estimates the maximum number of steps needed to reach the final position in one axis, either the x-axis or the y-axis. (this equation is taken from the actual code)

result = axis / positive\_divider;

result += (axis % (positive\_divider));

The *axis* represents the final position in one axis (*axis* = either ***x-final*** or ***y-final***), which also represents how many steps the final position is away from the start point which is (0, 0). The positive divider is how many steps we do toward (or backward) the final position when we finish executing the set of instructions. However, if we are not progressing at the end of all instructions (meaning that positive divider = 0), we put 1.

The positive divider is calculated by getting the difference between the positive moves and the negative moves that we do in one specific axis. We calculate the difference because one may hold back the other since they have different directions. Hence, more steps are needed to reach the final position. Positive moves can be either U or R and negative moves are either L or D. However, if the difference is negative, we change its sign and return it as a positive number since we want a positive number of how many times we need to execute all the instructions to get into the final position. Also, when the difference is equal to zero, we return 1 as mentioned before since it will be used in the denominator.

Therefore, if we need to calculate how many steps we need to reach the final position, we take how many steps the target is far away from the starting point (axis) divided by how many steps we are making each time (positive\_divider). As a result, we’ll get how many times we need to execute all the set of given instructions to get to the target if it is reachable.

However, the division operation may give a double or float\_type answer, which indicates that we will reach the final position after executing one of the middle instructions or the first instruction. So, we need to convert this decimal part to an integer number because it is hard to determine which instruction is meant by the decimal part. Therefore, we convert the decimal part into an integer with the modular operator and add it to the original result that is saved without the decimal part. I didn’t simply add 1 instead of doing the modular operator because in most cases the modular operator is equal to 0, so there is no need to do an extra loop for nothing.

And voila, we find an estimation of how many steps are needed to reach the final position in one axis. The same equation will apply to the final position on the second axis, and then sum up the two results as I said before. That will give us the final estimation of how many steps are needed to reach the final position (***x-final***, ***y-final***).

*Enhancements to the code.*

The previous algorithm was initially written in only two big functions, but the two functions violated the single responsibility principle of clean code that says each function should do one thing to make the program readable and maintainable. Therefore, I break them down into eight functions that do different things, but in the end, all of them contribute to efficiently solving the problem.

The second enhancement was done to the efficiency of the algorithm. After designing this algorithm, I realized that there is a special case where the robot will only go back and forth between two values (Oscillating). If you remember before, I said that there is an oscillating case that the previous algorithm couldn’t solve. This is the case I was talking about. To explain more this case, see the following example:

* Input:
  + 4
  + 4
  + URRDLL
* Output: Impossible

In this example, the robot will always return to the initial position after executing the set of given instructions. Therefore, it will never reach or exceed the final position. Consequently, the first two algorithms that I thought of will fail in this case scenario. However, the latter algorithm (the third one) will work fine since the condition of ending the loop is either the estimated number of the maximum times the instructions should be executed or when we reach the final position. But, in the case we have a scenario as follows:

* Input:
  + 1000000000
  + 1000000000
  + URRDLL
* Output: Impossible

We will execute all the set of instructions 2000000000 times although it is clear we’ll not reach the final position since as far as we can go is (2,1). Therefore, I came up with a function that checks if this Oscillation case happens or not. If yes, do both ***x-final*** and ***y-final*** exist in the oscillation range or not. If yes, then do the estimation, but if no, just return impossible. For more details, you can read the comments in the function definition of the function that is under the name “isOscillationFarAway”.

A computer screen shot of a program code

Description automatically generated

One more thing, depending on when you check if we reached the final position or not while executing each instruction, this final algorithm may or may not work if the final position is (0, 0). Therefore, I went with the option of checking if the final position is (0, 0) or not in the first beginning of the algorithm, so we don’t go through all the steps of the algorithm, and then found out that we are already in the final position, which is inefficient at all.