Problem Set 3: Robot Simulation

The questions below are due on Friday April 29, 2022; 01:53:00 PM.
Checkoff start: Apr 22 at 09:00AM
Checkoff due: Apr 29 at 09:00PM
Download Files

Pset Buddy

Your buddy for this week is kemily

Go ahead and reach out to your buddy via their kerb/email. Additionally, watch out for an email from your buddy! Please check this box once you have successfully made contact with each other. If your buddy has not responded after 2 days, please start the pset and contact the course staff. This will not affect grading.

✓ I have made contact with my buddy

View Answer **100.00%**

You have infinitely many submissions remaining.

Introduction

In this problem set, you will design a simulation and implement a program that uses classes to simulate robot movement. We recommend testing your code incrementally to see if your code is not working as expected. To test your code, run `test_ps3.py`.

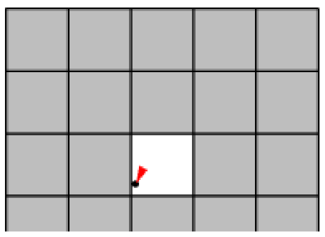
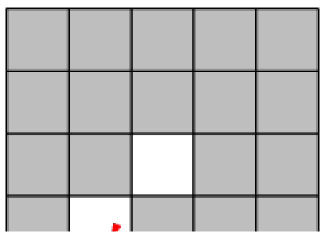
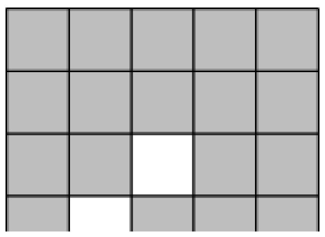
As always, please do not change any given function signatures. Remember to follow the 6.0002 Style Guide.

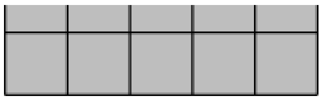
A) Simulation Overview

iRobot is a company (started by MIT alumni and faculty) that sells the Roomba Vacuuming Robot (watch one of the product videos to see these robots in action). Roomba robots move around the floor, cleaning the area they pass over.

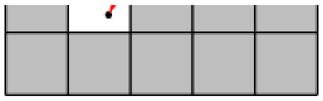
You will code a simulation to compare how much time a group of Roomba-like robots will take to clean the floor of a room. The following simplified model of a single robot moving in a square 5x5 room should give you some intuition about the system we are simulating. A description and sample illustrations are below.

The robot starts out at a random position in the room. Its direction is specified by the angle of motion measured in degrees clockwise from "north." Its position is specified from the lower left corner of the room, which is considered the origin (0.0, 0.0). The illustrations below show the robot's position (indicated by a black dot) as well as its direction (indicated by the direction of the red arrowhead).

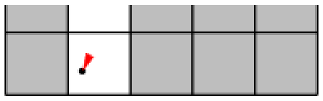






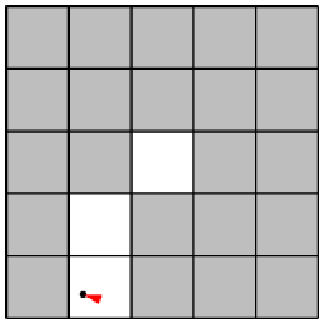
t0
The robot starts at the position (2.1, 2.2) with an angle of 205 degrees (measured clockwise from "north"). The tile that it is on is now clean.



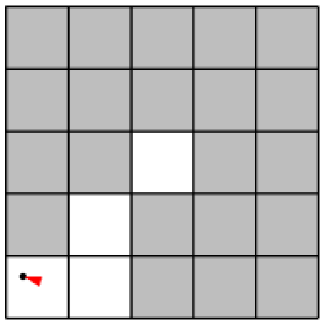
t1
The robot has moved 1 unit in the direction it was facing, to the position (1.7, 1.3), cleaning another tile.



t2
The robot has moved 1 unit in the same direction (205 degrees from north), to the position (1.2, 0.4), cleaning another tile.



t3
The robot could not have moved another unit in the same direction without hitting the wall, so instead it turns to face in a new, random direction, 287 degrees.



t4
The robot moves along its new direction to the position (0.3, 0.7), cleaning another tile.

B) Simulation Components:

Here are the components of the simulation model.

1. **Room**: Rooms are rectangles, divided into square tiles. At the start of the simulation, each tile is covered in some amount of dust, which is the same across all the tiles. You will first implement the class **Room** in Problem 1.
2. **Robot**: Multiple robots can exist in the room. iRobot has invested in technology that allows the robots to exist in the same position as another robot without causing a collision. You will implement the abstract class **Robot** in Problem 1. You will then implement the subclasses **NormalRobot**, **ClumsyRobot** and **SensingRobot** in Problems 2, 3, and 4.

More details about the properties of these components will be described later in the problem set.

C) Helper Code

To test your code, run `test_ps3.py`. You can comment out test suites at the bottom of `test_ps3.py` to test just one part. Note that any test containing the word Simulation (e.g. `ps3_P5_Simple.testSimulation1`) will produce an error until you implement Problem 5 of the pset.

We have also provided an additional file: `ps3_visualize.py`. This Python file contains helper code for visualizing your robot simulation. More information about this visualizer is located at the bottom of this pset.

1) Implementing the Room and Robot classes

Read `ps3.py` carefully before starting, so that you understand the provided code and its capabilities. **Remember to carefully read the docstrings for each function to understand what it should do and what it needs to return.**

The first task is to implement the class `Room` and the abstract class `Robot`.

In the skeleton code provided, the abstract class contains some methods which should only be implemented in the subclasses. **If the comment for the method says "do not change," please do not change it.** You can test your code as you go along by running the provided tests in `test_ps3.py`.

In `ps3.py`, we've provided skeletons for these classes, which you will fill in for Problem 1. We've also provided for you a complete implementation of the class `Position`. Do not change the `Position` class.

Class Descriptions:

- **Position** - Represents a location in x- and y-coordinates. x and y are floats satisfying $0 \leq x < w$ and $0 \leq y < h$, where w and h are the room's width and height.
- **Room** - Represents the space to be cleaned and keeps track of which tiles have been cleaned and how much dust is on each tile.
- **Robot** - Stores and sets the position, direction, and cleaning volume of a robot.

Room Implementation Details:

- Representation:
  - You will need to keep track of which parts of the floor have been cleaned by the robot(s). When a robot's location is anywhere inside a particular tile, we will consider the dust on that entire tile to be reduced by some amount determined by the robot. We consider the tile to be "clean" when the amount of dust on the tile is 0. We will refer to the tiles using ordered pairs of **integers**, $(0,0),(0,1),\ldots,(0,h-1),(1,0),(1,1),\ldots,(w-1,h-1)$, which will correspond to positions occupied by the robot while cleaning.
  - When choosing a data structure to represent the room, be careful as you keep track of the x- and y-coordinates and how they correspond to the room's dimensions.
  - Tiles can **never** have a negative amount of dust.
- Starting Conditions:
  - Initially, the entire floor is uniformly dirty. Each tile should start with an integer amount of dust, specified by `dust_amount`.
- Cleaning Conditions:
  - As the robots clean the room, the amount of dust on each tile changes. The method `clean_tile_at_position` locates the tile corresponding with the robot's position and reduces the amount of dust on the tile by the robot's cleaning volume. Remember that the amount of dust on each tile should be NON-NEGATIVE.
- Other methods:
  - You will also implement methods to return the amount of dust at a particular tile, to check whether a given tile is clean (dust amount is 0), to count the number of total tiles and cleaned tiles, to check if a position is inside the room, and to return a random position in the room.

Robot Implementation Details:

- Representation
  - Each robot has a **position** inside the **room**. We'll represent the position using an instance of the `Position` class. Remember the `Position` coordinates are floats.
  - A robot has a **direction of motion**. We'll represent the direction using a float `direction` satisfying $0 \leq$ direction < 360, which gives an angle in degrees from north.

    - A robot has a **cleaning volume**, `cleaning_volume`, which describes how much dust it can clean from each tile at each time.
    - A robot has a **speed**. We'll represent the speed using a positive float.
- Starting Conditions
    - Each robot should start at a random position in the room (hint: the `Robot's room` attribute has a method you can use) and a random direction that it will move in.
- Movement Strategy
    - A robot moves according to its movement strategy described below and in `ps3.py`, which you will implement in the `update_position_and_clean` methods. You do not need to implement this function here. It will be implemented in the subclasses.

If you find any places above where the specification of the simulation dynamics seems ambiguous, it is up to you to make a reasonable decision about how your program/model will behave, and document that decision in your code.

Complete the Room class and Robot abstract class by implementing their methods according to the specifications in ps3.py. You should now be passing all tests under the `ps3_1A` and `ps3_1B` test suites. Remember that the Robot class will never be instantiated; we will only instantiate its subclasses.

Hints:

- Make sure to think carefully about what kind of data type you want to use to store information about the floor tiles in the `Room` class.
- A majority of the methods should require only one line of code.
- Remember that tiles are represented using ordered pairs of integers $(0,0),(0,1),\ldots,(0,h-1),(1,0),(1,1),\ldots,(w-1,h-1)$. But a robot's `Position` is specified as **floats** (x, y). Be careful converting between the two! We recommend using `math.floor(x)` to always round down when converting to ensure that `Position`s are always in the room.
- Remember to give the robot an initial **random** position and direction. The robot's position should be an instance of the `Position` class and should be a valid position in the room. Note that the class `Room` has a `get_random_position` method that may be useful for this.
- In the final implementation of the `Robot` abstract class, not all methods will be implemented. Not to worry — their subclass(es) will implement them (e.g., `Robot's` subclasses will implement the method `update_position_and_clean`).
- **Your implementation may occasionally fail a few simulation tests. This is okay, and will not be counted against you as long as all of the tests pass most of the time.**

2) NormalRobot and Simulating a Timestep

Each robot must also have some code that tells it how to move about a room, which will go in a method called `update_position_and_clean.`

We have already refactored the robot code for you into two classes: the abstract `Robot` class you completed above (which contains general robot code), and a `NormalRobot` class inheriting from it (which contains its own movement strategy).

The movement strategy for `NormalRobot` is as follows. In each time-step:

- Calculate what the new position for the robot would be if it moved straight in its current direction at its given speed.
- If that is a valid position, move there and then clean the tile corresponding to that position by the robot's cleaning volume. The position is valid if it is in the room. Do not worry about the robot's path in between the old position and the new position.

- Otherwise, rotate the robot to be pointing in a random new direction. **Don't clean the current tile or move to a different tile**.

The `NormalRobot` then repeats this strategy at each time-step.

We have provided the `get_new_position` method of the `Position` class, which you may find helpful in implementing this. It computes and returns the new `Position` from the current `Position` object after a single time-step has passed with the given angle and speed parameters. Read the docstring of this method for more information.

Complete the update_position_and_clean method of NormalRobot to simulate the motion of the robot during a single time-step (as described above in the time-step dynamics). You should now be passing the `ps3_P3` test suite.

Testing Your Code: Before moving on to Problem 3, you can check that your implementation of `NormalRobot` works by uncommenting the line at the bottom of the file within the `if __name__ == "__main__:"` block:
`test_robot_movement(NormalRobot, Room)`

The file will open a visualization of a 5 by 5 room as implemented in `Room` and a robot (the arrowhead) as implemented in `NormalRobot`. Initially, all dirty tiles are marked as black. As the robot visits each tile and cleans the tile according to its given cleaning volume, the color of the tile changes from black to gray to white, with white meaning the tile is completely clean.

Make sure that as your robot moves around the room, the tiles get lighter (from black to gray to white) each time your robot lands on them. The simulation terminates when the robot finishes cleaning the entire room. Make sure your robot doesn't violate any of the simulation specifications (e.g., your robot should never move to a position outside of the room, it should never clean the tile if it also had to choose a new direction, etc.)

Do not worry if it appears your robot is "cutting corners" as it cleans, as long as its final position in each time step is never outside of the room.

---

3) Implementing ClumsyRobot

iRobot's roombas have become quite the hit sensation. And the company wants more people to be able to take advantage of their product. iRobot has made a more affordable version of the roomba, but it doesn't hold the dust as securely as the NormalRobot. Therefore, sometimes, the ClumsyRobot will drop some dust on a tile instead of cleaning it. You wonder how badly this affects the time it takes a robot to clean a room and decide to design a simulation.

Note: Whether the robot drops dust or not is determined separately for each timestep. If your robot drops dust at one timestep, it may or may not drop dust again at the next timestep.

Complete the class ClumsyRobot that inherits from Robot (just as NormalRobot inherits) but factors in the dropped dust behavior. ClumsyRobot should have its own implementation of `update_position_and_clean`.

The behavior for a `ClumsyRobot` is outlined in the docstring.

We have written a method `dropping_dust` inside `ClumsyRobot` for you that you should use in order to determine if your robot drops dust on a tile. The robot drops dust with probability p = 0.05. If the robot does drop dust, the amount of dust that will be dropped is a random decimal value between 0 (inclusive) and 0.5 (exclusive). The random class may be useful. If you are confused how to drop dust on a tile, think about what this means in terms of how 'clean' a tile becomes. As with `NormalRobot`, you may find the provided `get_new_position` method of `Position` helpful.

Testing Your Code Test out your new class. Perform a single trial with the new `ClumsyRobot` implementation and watch the visualization to make sure it behaves as expected by uncommenting this line at the bottom of the file:

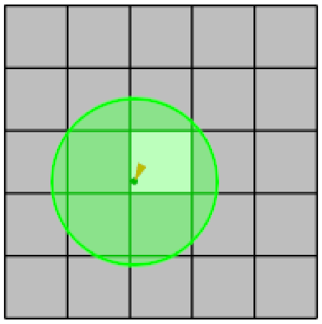
```
test_robot_movement(ClumsyRobot, Room)
```

4) Implementing SensingRobot

iRobot just signed a contract with DustDetector, a startup that makes laser scanners for detecting dust on the floor. They hope that adding this sensor capability to their robots will help clean floors more efficiently.

The way that the sensor works is that it takes one reading every 5 integer angles between 0 and 360, (i.e. `[0, 5, 10, ... 355]`). The sensor is calibrated to a range that is equal to the robot's speed. For each angle, it looks ahead at the position the robot would move to given it's speed and returns the amount of dust on that tile. The sensor then returns a dictionary mapping the angles to the amount of dust it sees at that angle.

NOTE: If the sensor sees a wall in one of it's scans, it will give a reading of -1



With this very helpful information, the robot can then make a decision about how proceed. It will find the angles that have the most amount of dust, and randomly choose one as its next direction.

NOTE: Given the discrete nature of the tiles, it would make sense that many angles will have the same amount of dust! i.e. If all of the surrounding tiles have the same amount of dust, the robot will pick an angle from the dictionary returned by the sensor randomly.

The "sensor" has been implemented for you. Your job is to implement SensingRobot's `update_position_and_clean` function, which does the following in one timestep:

1. Scan the surrounding area (using the provided function, `scan_surrounding_area`)
2. Find the angles with the maximum amount of dust
3. Pick one of the dirtiest angles at random and move in that direction (random.choice() might be useful!)
4. Clean the tile the robot lands on

You must also design a simulation to determine how well this affects the time it takes a robot to clean a room.

HINTS:

- If a robot senses a wall, is it possible for it to move to an invalid position given this algorithm?

Complete the class `SensingRobot` that inherits from `Robot` (just as `NormalRobot` inherits) but implement the new strategy using the sensor information. `SensingRobot` should have its own implementation of `update_position_and_clean`.

Testing Your Code Test out your new class. Perform a single trial with the new `SensingRobot` implementation and watch the visualization to make sure it behaves as expected.

Uncomment the following line at the bottom of the file to watch your SensingRobot Implementation:
`test_robot_movement(SensingRobot, Room)`

---

5) Creating the Simulator

In this problem you will write code that:

1. Simulates the robot(s) cleaning the room up to a specified fraction of the room for various trials; and
2. Returns the mean number of time-steps needed to clean the room.

Once you have written this code, you'll comment on the results of your simulation in Problem 6.

Implement `run_simulation(num_robots, speed, cleaning_volume, width, height, dust_amount, min_coverage, num_trials, robot_type)` according to its specification.

Simulation Starting Conditions:

1. Each trial should begin with a new room and `num_robots` number of robots.
2. Each robot in the trial should start at a random position in that room.
3. Each room should start with a uniform amount of dust on each tile, given by `dust_amount`.

At Each Time Step:

1. Each robot in the room should perform its strategy specified by `update_pos_and_clean`.
2. Once all the robots have performed their actions, move on to the next time step.

A trial **ends** when a specified fraction of the room tiles have been fully cleaned (i.e., the amount of dust on those tiles is 0). The simulation **terminates** once the specified number of trials has been run. The average number of time steps for the simulation is the sum of the time steps across all the trials over the number of trials.

The first six parameters of `run_simulation` should be self-explanatory. If you are confused, check the docstrings. For the time being, you should pass in `NormalRobot` for the `robot_type` parameter, like so: `avg = run_simulation(10, 1.0, 1, 15, 20, 5, 0.8, 30, NormalRobot)`

When writing `run_simulation` you should use `robot_type(…)` instead of `NormalRobot(...)` whenever you wish to instantiate a robot. (This will allow us to easily adapt the simulation to run with different robot implementations, which you'll encounter in Problem 6.) Feel free to write whatever helper functions you wish. You should now be passing all of the test cases.

Simulation Animation: If you want to see a visualization of your simulation, similar to the visualization that pops up when you call `test_robot_movement`, check the end of this pset for instructions!

You should now be passing all tests.

---

6) Running the Simulator

Now, use your simulation to answer some questions about the robots' performance. In order to do this problem, you will be using a Python package called `pylab` (aka `matplotlib`). If you want to learn more about `pylab`, please read this tutorial.

For the questions below, uncomment the function calls provided (at the very end of the problem set) and run the code to generate a plot using pylab, and then answer the corresponding questions underneath the function calls in `ps3.py`.

1. Examine `show_plot_compare_strategies` in `ps3.py`, which takes in the parameters title, `x_label`, and `y_label`. It outputs a plot comparing the performance of all types of robots in a 20x20 `Room` with 3 units of dust on each tile and 80% minimum coverage, with a varying number of robots with speed of 1.0 and cleaning volume of 1. Uncomment the call to `show_plot_compare_strategies`, and answer question #1. Depending on your computer, it may take a few minutes for the plot to show up. Remember to comment this out when submitting your pset.

2. Examine `show_plot_room_shape` in `ps3.py`, which takes in the same parameters as `show_plot_compare_strategies`. This figure compares how long it takes two of each type of robot to clean 80% of `Rooms` with dimensions 10x30, 20x15, 25x12, and 50x6 (notice that the rooms have the same area.) Uncomment the call to `show_plot_room_shape`, and answer question #2. Depending on your computer, it may take a few minutes for the plot to show up. Remember to comment this out when submitting your pset.

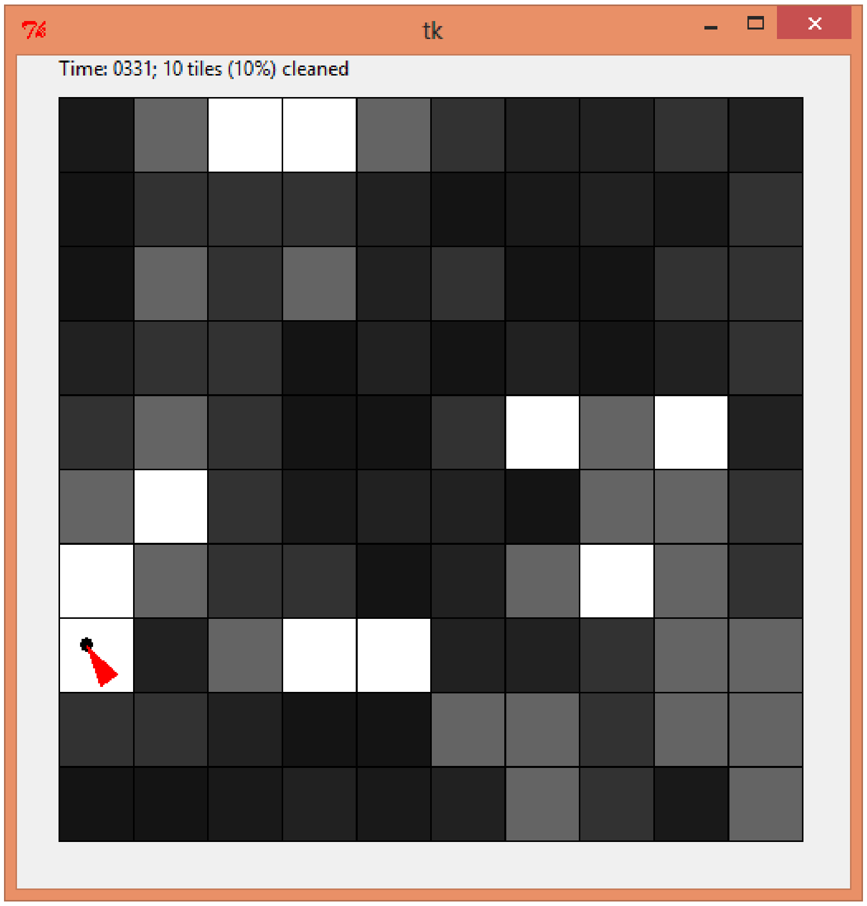
Please save the two generated plots in a PDF, which you will submit at the bottom the this page. You will not be graded on the plots, but they will be referred to during the checkoff, so make sure you can pull the PDF with the plots up before going to get your checkoff. If you do not have the plots available, you will not be able to get a checkoff, since running the simulation may take up checkoff time.

Visualizing Robot Simulation

We've provided some code to generate animations of your robots as they go about cleaning a room. These animations can also help you debug your simulation by helping you to visually determine when things are going wrong.

Running the Visualization:

1. In your simulation, at the beginning of a trial, do the following to start an animation:
   `anim = ps3_visualize.RobotVisualization(num_robots, width, height, delay)`
2. Pass in parameters appropriate to the trial, of course. `delay` is an optional parameter that is discussed below. This will open a new window to display the animation and draw a picture of the room.
3. Then, during each time-step, after the robot(s) move, do the following to draw a new frame of the animation:
   `anim.update(room, robots)` where `room` is a `Room` object and robots is a list of Robot objects representing the current state of the room and the robots in the room.
4. When the trial is over, call the following method:
   `anim.done()`
   The resulting animation will look like this:



Initially, all dirty tiles are marked as black. As the robot cleans each tile by its given cleaning volume, the color of the tile transits from black to gray to white, with white means completely clean.

The visualization code slows down your simulation so that the animation doesn't zip by too fast (by default, it shows 5 time-steps every second). Naturally, you will want to avoid running the animation code if you are trying to run many trials at once.

Delay:
For purposes of debugging your simulation, you can slow down the animation even further. You can do this by changing the call to `RobotVisualization`, as follows:

```
anim = ps3_visualize.RobotVisualization(num_robots, width, height, delay)
```

The parameter delay specifies how many seconds the program should pause between frames. The default is 0.2 (5 frames/second). You can raise this value to make the animation slower.

For problem 6, we will make calls to `run_simulation` to get simulation data and plot it. However, you don't want the visualization getting in the way. If you choose to do this visualization exercise, before you get started on problem 6 and before you turn your problem set in, **make sure to comment out the visualization code out of `run_simulation`.**

---

7) Hand-In Procedure

1. Save

Save your solution as `ps3.py`, and add your two plots from the simulations to a PDF.

2. Test

Run your file to make sure it has no syntax errors. Test your `run_simulation` to make sure that it still works with all of the `NormalRobot`, `ClumsyRobot` and `SensingRobot` classes. Make sure that plots are produced when you run the two simulation functions and verify that the results make sense. Make sure all the tests run.

3. Time and Collaboration Info

At the start of ps3.py, in a comment, write down the names of the people you collaborated with. For example:

```
# Problem Set 3
# Name: Jane Lee
# Collaborators: John Doe
```

Please estimate the number of hours you spent on the Problem Set in the question box below.

10

Thank you!

You have infinitely many submissions remaining.

Note: Passing all of the tests in the local tester does not necessarily mean you will receive a perfect score. The staff will run additional tests on your code to check for correctness.

4. Submission

The tester file contains a subset of the tests that will be run to determine the problem set grade.

You may upload new versions of each file until the 9PM deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.

When you upload a new file with the same name, your old one will be overwritten.

Submitting your `ps3.py` make take up to 40 seconds, so don't worry and just be patient if the submission loads for a bit.

Make sure to also submit the PDF with the two plots, as these will be used during your checkoff.

Download Most Recent Submission

Select File No file selected

Code Submitted Successfully!

You have infinitely many submissions remaining.

Plots PDF Submission

Download Most Recent Submission

Select File No file selected

File Submitted Successfully!

You have infinitely many submissions remaining.