CRC

PROGRAMMING COMPETITION

PROBLEM BOOKLET BLOCK B

A FEW NOTES

- The complete rules are in section 4 of the rulebook.
- You have until **15h59** to submit your solutions.
- Feel free to ask organizers questions and discuss the problem with your team members. It is not an exam!
- **We are giving you quick and easy-to-use template files. Please use them!**

USING THE TEMPLATE FILE

- All the files contain a function called solve() where you have to put your code. **DO NOT CHANGE THE NAME OF THAT FUNCTION!**
- To run the tests you will have to open a terminal (we strongly recommend to use vscode) and run the command `pytest.` If the command is ran in the folder containing all the problems, all tests are going to be ran
- To run a specific test, run it in the folder containing the test you want. You can do the same for sections, just navigate in the folder before executing the command `pytest`

STRUCTURE

Every problem contains a small introduction like this about the basics of the problem and what is required to solve it. Points distribution is also given here for the preliminary problems.

Input and output specification:

In these two sections, we specify what the inputs will be and what form they will take, and we also say what outputs are required for the code to produce and in what format they shall be.

Sample input and output:

In these two sections, you will find a sample input (that often has multiple entries itself) in the sample input and what your program should produce for such an input in the sample output.

First output explanation:

Sometimes, the problem might still be hard to understand after those sections, which is why there will also be a usually brief explanation of the logic that was used to reach the first output from the first input.

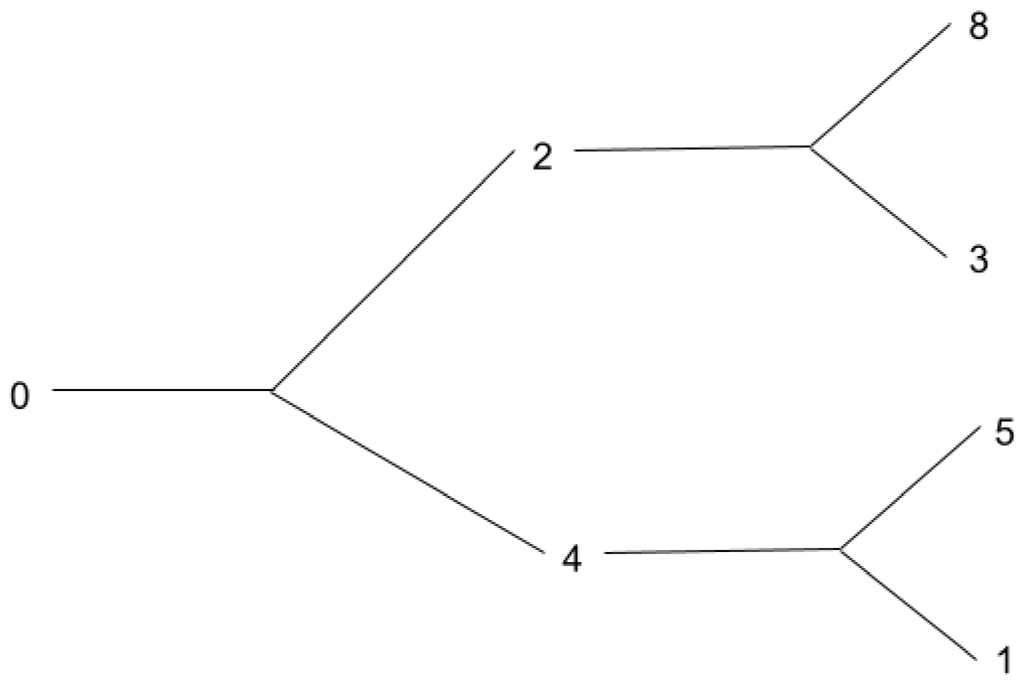
Table of content

MODES OF TRANSPORTATION (MT)

MT1: Choose your path! (30 points)

You are a train operator, and you want to please as many passengers as possible in a single trip. Your path choices consist of a series of branches where you can go either left or right. This is your only decision—whether to go left or right—and these branches will determine which stations you pass through. Therefore, you must find the path that brings the maximum number of passengers to their destination. You will receive the number of passengers getting off at each station in a list representing a tree. Here is a quick representation of the choices you can make and the associated passenger list.

[0, 2, 4, 8, 3, 5, 1]



You will need to find the path that brings the highest number of passengers to their destination. In this case, the best path is left followed by left again to bring 10 passengers to their destination. The output would be [‘L’, ‘L’]. The directions are L for left and R for right.

Input Specification:

You will receive:

- ***passengers:*** the number of passengers for each destination

Output Specification:

As an output, you will provide the directions the train should take.

Sample input:

[0, 2, 4, 8, 3, 5, 1]
[0, 2, 4, 8, 3, 5, 1, 1, 2, 2, 3, 9, 8, 7, 5]
[0, 6, 4, 2, 5, 8, 1]

Sample output:

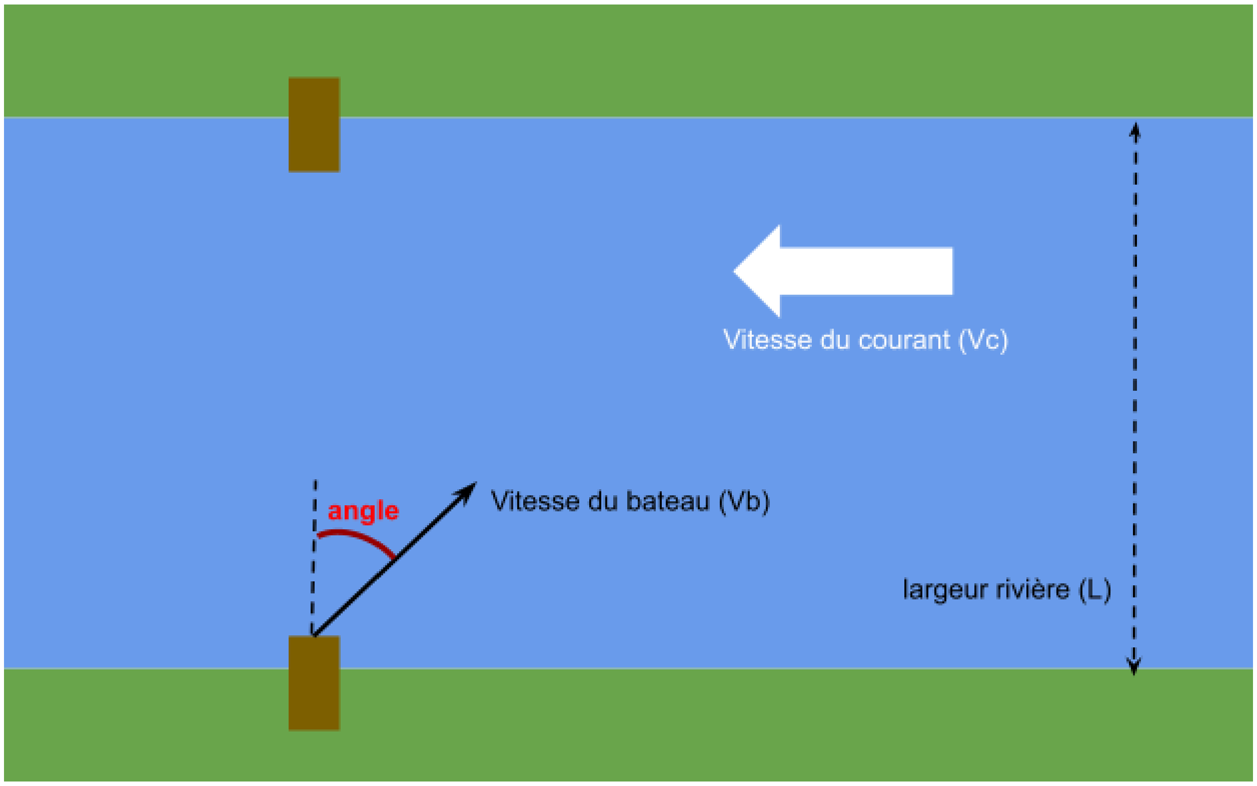
```
['L', 'L']
['R', 'L', 'L']
['R', 'L']
```

First output explanation:

See the explanation in the description of the problem.

MT2: Crossing the river (15 points)

You are operating a small boat that ferries tourists across the St. Lawrence River. To reach the dock on the opposite shore directly in front of you, you must account for the river's current. We will provide you with the speed of the boat and the speed of the current, and you will need to calculate the time required to complete the crossing.



Here are the formulas you will need to calculate the crossing time. First, since the two docks are aligned, you will need to determine the angle of the boat to counter the current. This will allow you to find the speed component that opposes the current and, therefore, the angle the boat must take.

$$angle = arcsin(\frac{V_c}{V_b})$$

We can now determine the speed component in the direction of the crossing.

$$V_{\perp} = V_b \cdot cos(angle)$$

And now, with the speed and distance, we can calculate the time required for the crossing.

$$t = \frac{L}{V_{\perp}}$$

Make sure to provide the crossing time in minutes, with the number of seconds rounded to the nearest second.

Input Specification:

You will receive:

- ***Vb:*** a *float* of the speed of the boat in m/s
- ***Vc***: a *float* of the speed of the courant in m/s
- **L**: an *int* of the width of the river in meters

Output specification:

You will need to output:

- a *string* of the time to cross the river using the format XXmXXs

Sample input:

```
Vb = 4.6 Vc = 3.9 L = 1350
Vb = 5.1 Vc = 4.7 L = 2250
Vb = 3.1 Vc = 1.7 L = 800
```

Sample output:

```
09m13s
18m56s
05m09s
```

First output explanation:

For the first example with Vb=4.6 (boat speed), Vc=3.9 (current speed), and L=1350 (distance), we can start by calculating the angle the boat must take to counter the current.

$$angle = arcsin(\frac{V_c}{V_b})$$

$$angle = 1.0118721573923892 radians$$

With the angle, we can find the component of the speed going toward the crossing of the river.

$$V_\perp = V_b \cdot cos(angle)$$

$$V_\perp = 4.6 m/s \cdot cos(1.0118721573923892 radians)$$

$$V_\perp = 2.4392621835300927 m/s$$

Now we know the effective speed toward the other side of the rive, we can find the time to cross the river.

$$t = \frac{1350m}{2.4392621835300927 m/s}$$

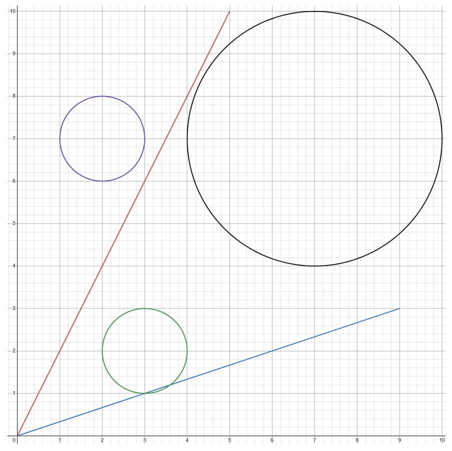
$$t = 553.4460416412819 s$$

t = 09m13s

MT3: We go to hyperespace! (90 points)

You are in the midst of a battle to liberate the galaxy when a retreat is called in the face of an overwhelming defeat. Transporting a passenger of the utmost importance, you must jump to hyperspace to escape. However, your navigation computer, which normally helps access secure routes (i.e., routes that avoid collisions with celestial bodies), is broken. Therefore, you must transmit a backup algorithm to calculate how to reach a specific point in hyperspace quickly enough.

Your ship and destination will be placed on a Cartesian plane ranging from 0 to 10 in both the x and y directions. The origin point (0,0) is located at the bottom left of the plane, and the point (10,10) is at the very top right. **Your ship can move between any combination of two points in a straight line, provided it does not collide with any celestial body.** For example,



In the plane above, the red path between points (0,0) and (5,10) is allowed as it is a straight line and does not intersect with any celestial body, whereas the blue path from (0,0) to (9,3) is not allowed as it collides with the green celestial body. A line perfectly tangent to one of the circles (i.e., it only intersects the celestial body at one point, for example, the line from (2,1) to (4,1) is tangent to the green circle) is considered invalid and collides with the celestial body.

The celestial bodies will all be circles, and will be given in the form (center point, radius). For example, the green circle is described by ((3,2),1). To form a circle (create a series of points!) from this data, you simply use the equation of a circle. Using the center point (a,b) and the radius r, the equation becomes:

$$(x-a)^2+(y-b)^2=r^2$$

You will need to provide a series of points starting from the coordinates of the ship and ending at the coordinates of the destination, containing all the points where the ship changes trajectory in between. In other words, the ship travels in a straight line between one point and the next in this list, then the next point becomes the starting point for the next straight line path to the 3rd point in the list, and so on, until the destination. **You will need to find a valid route that is within 5% of the optimal distance,** meaning if the best path is 8 units long, your solution cannot exceed 8.4 units. It is useful to remember the distance formula:

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

Where Δ represents the difference between two coordinates, in this case, the differences between the x and y coordinates. It's possible that the optimal path only involves the starting point of the ship and its destination if it's a valid straight line. The optimal path will never require more than 5 movements (or changes in trajectory), so your series of points will never need to be longer than 6 points, but it will still be valid with more points as long as the distance and collision constraints are respected.

Final note: It is possible that, due to an error on our part, you may find a better path than the one we used to calculate the optimal distance for an example. Your route will obviously be correct, but we recommend quickly checking that it does not intersect with any celestial bodies on paper or using Desmos to ensure that it is valid.

Input Specification:

You will receive:

- ***vaisseau:*** a tuple of 2 coordinates that are the start point of the vessel
- ***destination:*** a tuple of 2 *int* that are the coordinates of the destination
- ***astres:*** a *tuple* of coordinates in 3D of the celestial bodies to avoid

Output Specifications:

You will need to output:

- an *array* of *tuples* containing ***int*****,** that are the coordinates to follow the itinerary

Sample input:

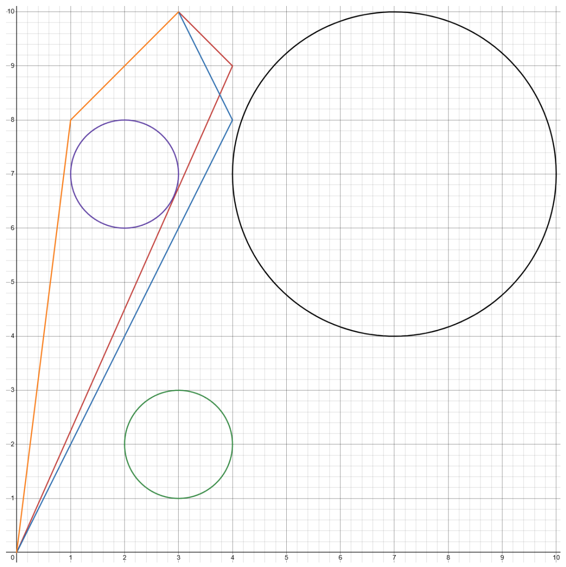
```
(0,0) (3,10) (((3,2),1),((2,7),1),((7,7),3))
(2,0) (9,4) (((6,2),2),((3,7),3),((8,6),1))
(6,9) (9,6) (((4,5),4),((9,8),1))
(2,9) (10,1) (((7,2),2),((4,7),2),((7,7),1),((9,6),1),((2,2),2))
```

Sample output:

```
[(0,0),(1,8),(3,10)] (distance = ~10.89)
[(2,0),(4,3),(6,5),(9,4)] (distance = ~9.60)
[(6,9),(9,6)] (distance = ~4.24)
[(2,9),(1,5),(8,4),(9,3),(10,1)] (distance = ~14.84)
```

First output explanation:

Since the path from (0,0) to (3,10) passes almost directly through the center of the purple celestial body, the options for the best path are to barely avoid it without touching it. Here, only 3 very good paths are shown, but there are of course many more possible.



The blue path covers a distance of ~11.18 units, the red path covers ~11.26 units, and the yellow path covers ~10.89 units. Therefore, the yellow path is the shortest, and the best solution is [(0,0),(1,8),(3,10)]. The other two paths would have been accepted, however, since their distance is less than 10.89*1.05 = ~11.43 units.

MT4: How can I get out of the parking lot? (55 points)

You live in Montreal and unfortunately need to drive every day to work in the downtown area. You have a parking space where you can park your car, but getting your car out is always a major challenge, so you want to code a program to help you exit the parking lot without hitting other vehicles.

You will have a grid in which your vehicle will be marked by @, and the other vehicles will be marked with the letters A, B, C, and so on. The vehicles are at least 2 units long and always move in the direction of their length. Therefore, some vehicles can only move left (L) to right (R), while others can only move up (U) to down (D). To indicate a move, you give the name of the vehicle and its direction (e.g., BU), and each move is one square in the chosen direction.

To exit the parking lot, your vehicle's front portion must be between the two walls of the exit. No vehicle can move onto a square occupied by another vehicle, otherwise, there will be a collision. You need to find a sequence of moves to get your vehicle out that does not exceed 100 operations.

P.S. As many solutions are valid, your code will be directly analyzed

Input Specification:

You will receive:

- ***grid:*** an *array* of *string* that represent the parking lot with all the vehicles

Output Specification:

You will need to output:

- an *array* of *string* of the maneuvers to get out of the parking lot

Sample input:

```
'########',
'#AA   B#',
'#C  D B#',
'#C@@D B ',
'#C  D  #',
'#E   FF#',
'#E GGG #',
'########'
```

```
'########',
'#A  BBB#',
'#A  C D#',
'#@@ CED ',
'#FFF ED#',
'#  G HH#',
'#IIGJJ #',
'########'

'########',
'#      #',
'#      #',
'# @@A   ',
'# BBA C#',
'# D A C#',
'# DEE C#',
'########'
```

<u>Sample output:</u>

```
There are other valid answers than the ones shown
['AR', 'CU', 'EU', 'GL', 'GL', 'FL', 'FL', 'FL', 'DD', 'DD',
'BD', 'BD', 'BD', '@R', '@R', '@R', '@R']

['@R', 'FR', 'AD', 'AD', 'AD', '@L', 'BL', 'BL', 'BL', 'EU',
'DU', 'FR', 'FR', 'GU', 'GU', 'GU', 'IR', 'AD', 'HL', 'HL',
'HL', 'FL', 'FL', 'FL', 'CD', 'CD', 'ED', 'ED', 'DD', 'DD',
'DD', 'BR', 'BR', 'BR', 'GU', '@R', '@R', '@R', '@R', '@R']

['AU', 'AU', 'CU', 'CU', 'CU', 'ER', 'ER', 'BR', 'BR', 'BR',
'AD', 'AD', 'AD', '@R', 'DU', 'DU', 'DU', 'DU', '@L', 'AU',
'AU', 'AU', 'BL', 'BL', 'BL', 'EL', 'EL', 'EL', 'AD', 'AD',
'AD', 'CD', 'CD', 'CD', '@R', '@R', '@R', '@R']
```

<u>First output explanation:</u>

In the first grid, we start by moving the car at the top (A) to make room for the two cars on the left to move each by one square (C and E). This way, we can move the car at the bottom (G) all the way to the left and also shift the car above it (F) to the left. Now, we can move car D down to the bottom and also move car B down to clear the path so that our car can exit the parking lot! **Other solutions may be valid besides the ones shown in the examples.**

MT5: Plane landing (25 points)

You are creating a security system to assist with airplane landings. You calculate the altitude and approach angle to ensure the aircraft stays on the correct trajectory. The ideal approach angle is 3 degrees relative to the touchdown point on the runway. We will provide the projected ground distance between the airplane and the landing point, as well as the speed.

Unfortunately, the real world is not perfect, so after a few uninterrupted seconds of descent, a gust of wind will push the airplane upward. You must determine whether the new trajectory remains perfect (between 3 and 4 degrees), acceptable (between 4 and 5 degrees), or dangerous (more than 5 degrees).

If the angle after the wind gust is:

- Between 3 and 4 degrees, return: **"Perfect"**
- Between 4 and 5 degrees, return: **"Okay"**
- More than 5 degrees, return: **"Danger"**

Input Specification:

You will receive:

- ***distance:*** the distance in meters as an *int* of the distance projected on the floor.
- ***speed:*** The speed in meters per second of the plane towards the landing spot as an *int*
- ***time:*** an *int* of the number of seconds between the start of the descent and the turbulence
- ***offset:*** an *int* of the number of meters the plane is lifted by during the turbulence

Output Specification:

You will need to output:

- a *string* of the status for landing

Sample input:

```
distance=850, speed=60, time=8, offset=10
distance=720, speed=65, time=9, offset=5
distance=880, speed=55, time=4, offset=11
```

Sample output:

```
"Okay"
"Danger"
"Perfect"
```

First output explanation:

For the first example, we have a ground distance of 850 meters, which gives us a direct air distance of 851.1664940982328 meters with an angle of 3 degrees. We can begin the descent for 8 seconds at a speed of 60 m/s, bringing us to a flight distance of 371.1664940982328 meters. This direct distance can also be expressed as 370.65782331780446 meters in ground distance and 19.42535339397198 meters in altitude, because before the turbulence, we maintain a 3-degree approach.

With the turbulence, we rise an additional 10 meters, bringing our angle to 4.539011506371929 degrees, which is considered “Okay.”

4 ELEMENTS (EL)

EL1: Wind gust (45 points)

You are responsible for a wind farm and need to ensure that none of the turbines are stuck. Normally, when the wind blows on the turbine blades, they should rotate in the direction the wind pushes them. To simplify the task, we will represent the wind turbines as 90-degree corners on an n by n grid, and the gusts of wind given will only come from the four cardinal directions (up, right, down, or left). At the start of the day, all wind turbines have their blades pointing to the right and downward. Their rotation point will be fixed at Cartesian coordinates with integer values (enter image for clarification).

When a gust of wind passes, the turbine must rotate 90 degrees if one of its blades is in the path of the wind. The gusts will be given in the form +/- n x/y. The sign and the axis determine the wind direction (+x = to the right, +y = upward), while "n" represents the row or column where the wind blows. If the wind blows along the X-axis, the gust will pass between row "n" and "n-1". If the wind blows along the Y-axis, the gust will pass between column "n" and "n-1."

(Corner (0, 0) is at the bottom-left).

These are the ascii values to use for the turbines: ASCII 191, 192, 217, 218

Input Specification:

You will receive:

- ***n:*** the number of turbine per row
- ***gusts:*** an *array* of *strings* of the wind gusts

Output Specification:

You will need to output:

- an *array* of *strings* of the final state of all the turbines

Sample input:

```
6, (3x, -2y, -5x, 0y)

8, (0x, -0y, 8x, 3y)

4, (-2y, 3x, -2y)
```

Sample output:

```
┘┘┐┐┐┐
[┐[[[[
 ┌
⊓ ┌┌┌┌
⊓ ┌┌┌┌
⊓ ┌┌┌┌

┌┌└┌┌┌┌┌
┌┌└┌┌┌┌┌
┌┌└┌┌┌┌┌
┌┌└┌┌┌┌┌
┌┌└┌┌┌┌┌
┌┌└┌┌┌┌┌
[└┘[[[[[

└┐ └└
⊓ ┌┌
⊓ ┌┌
⊓ ┌┌
```

First output explanation:

```
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
```

`bourrasque 5x`

```
[[[[[[[[[
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
```

```
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
```

`bourrasque 3y`

```
┌ ┌ └ ┌ ┌ ┌ ┌ ┌ ┌
┌ ┌ └ ┌ ┌ ┌ ┌ ┌ ┌
┌ ┌ └ ┌ ┌ ┌ ┌ ┌ ┌
┌ ┌ └ ┌ ┌ ┌ ┌ ┌ ┌
┌ ┌ └ ┌ ┌ ┌ ┌ ┌ ┌
┌ ┌ └ ┌ ┌ ┌ ┌ ┌ ┌
┌ ┌ └ ┌ ┌ ┌ ┌ ┌ ┌
```

```
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
```

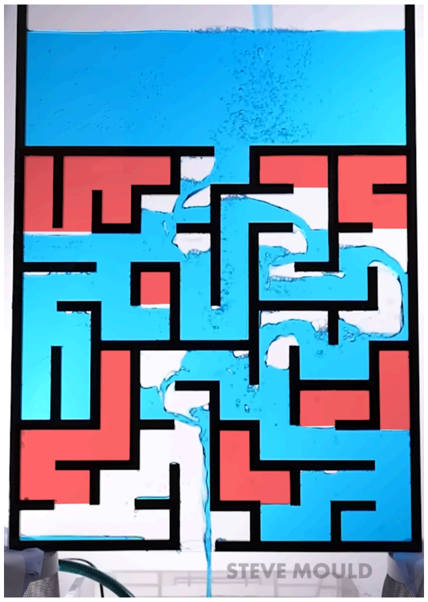
bourrasque -3y

$$\longrightarrow$$

```
┌┌┐ ┌┌┌┌┌┌
┌┌┐ ┌┌┌┌┌┌
┌┌┐ ┌┌┌┌┌┌
┌┌┐ ┌┌┌┌┌┌
┌┌┐ ┌┌┌┌┌┌
┌┌┐ ┌┌┌┌┌┌
┌┌┐ ┌┌┌┌┌┌
```

EL2: Water (80 points)

You watched Steve Mould's video on YouTube (https://youtu.be/81ebWToAnvA?si=VSnPgD3V1bsac7XE&t=252) where he talks about solving a maze with water. However, unlike simulations without air, the real simulation shows a different phenomenon. Although the maze is solved, several pockets of air are not filled with water! This is quite an interesting phenomenon; we notice that the water cannot dislodge the air trapped in these raised sections unless they lead to the exit or the air can escape.



Therefore, you need to take a maze as input and identify all the places where water could collect. These are all the spots that are NOT red in the simulation. The maze walls are represented by #, and you need to place dots (.) in all the locations that can hold water.

Input Specification:

You will receive:

- ***labyrinth:*** an *array* of *strings* that are the labyrinth

Output Specification:

You will need to output:

- an *array* of *strings* that correspond to the labyrinth with all the spot that may contain water to be filled with dots

Sample input:

```
'########### #########',
'# #     #   #   #   #',
'# # # ### ##### # ###',
'# # # #   #     #   #',
'# # ### ### ### ### #',
'#         # #   #   #',
'##### ### # # ### # #',
'#     # # # #     # #',
'# ### # # # ####### #',
'#   #   #   #   #   #',
'# # ######### # # ###',
'# # # #     # #   # #',
'# # # # ### # ##### #',
'# # # #   #   #   # #',
'##### ### ### # ### #',
'#     #   # #       #',
'# ##### # # ####### #',
'#   #   # #   #     #',
'### # ### ### # ### #',
'#       # ###   #   #',
'######### ###########'
```

```
'########## ########',
'# ### ##   ##   ###',
'# # # ## #### # ###',
'#             #   #',
'################# #'
```

```
'### #######',
'#   #   # #',
'# ### # # #',
'#     #   #',
'######### #',
'#    #### #',
'# ##      #',
'# #########'
```

Sample output:

```
'###########.#########',
'# #     #...#   #   #',
'# # # ###.##### # ###',
'# # # #...#.....#   #',
'# # ###.###.###.### #',
'#.........#.#...#...#',
'#####.###.#.#.###.#.#',
'#.....# #.#.#.....#.#',
'#.###.# #.#.#######.#',
'#...#...#...#...#...#',
'#.#.#########.#.#.###',
'#.#.# #.....#.#...# #',
'#.#.# #.###.#.##### #',
'#.#.# #...#...#   # #',
'##### ###.###.# ### #',
'#     #...# #.......#',
'# #####.#.# #######.#',
'#   #...#.#   #.....#',
'### #.###.### #.###.#',
'#.......#.###...#...#',
'#########.###########'
```

```
'##########.########',
'# ### ##...##...###',
'# # # ##.####.#.###',
'#.............#...#',
'#################.#'
```

```
'###.#######',
'#...#...# #',
'#.###.#.# #',
'#.....#...#',
'#########.#',
'#....####.#',
'#.##......#',
'#.#########'
```

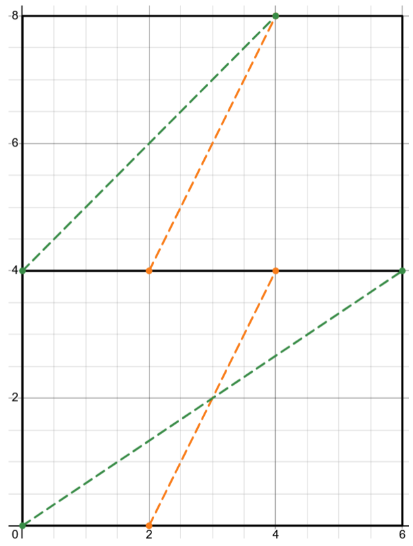
First output explanation:

The first labyrinth is the one from the video and picture!

EL3: Earth is cracking (70 points)

On very dry terrain, the ground tends to crack. One day, while walking on such terrain, Elrik wonders how many different pieces are created by the cracks in a given area.

We will provide you with the coordinates of the bottom-left and top-right corners of a rectangle on a Cartesian plane representing the area to be divided. The cracks will be approximated by straight line segments connecting two points placed on the perimeter of the rectangle. A crack passing through a section, no matter where, splits it into two smaller sections. The first crack will therefore split the area into two pieces. The following crack will "create" a new piece if it does not intersect the previous crack, or 2 pieces if it crosses the previous one.



In the upper case, the second crack (in green) does not intersect with the first one (**inside the rectangle**). It only "creates" one more piece, bringing the total to 3.

In the lower case, the two lines intersect within the rectangle, "creating" 2 new pieces, bringing the total to 4.

The formula for the number of pieces generated by a new crack is as follows:
new pieces = #intersections + 1

It is important to note that the number of intersections does not always correspond to the number of lines crossed. If a crack crosses a point that is already an intersection of two or more other cracks, it only counts as one intersection, even if multiple cracks are crossed at that point. You will need to account for this in your code. A visual representation of this special case will be shown in the explanation of the first output.

P.S. you are allowed external libraries to check for line intersection

Input Specification:

You will receive:

- ***coins:*** a *tuple of tuple of int* that are the coordinates of the corners of the rectangle.
- ***craques:*** an *array* of 2D *tuples* where each tuple is a pair of coordinates of the start and end point of each crack.

Output Specification:

You will need to output:

- an *int* corresponding to the number of distinct zones in the rectangles after the cracks.

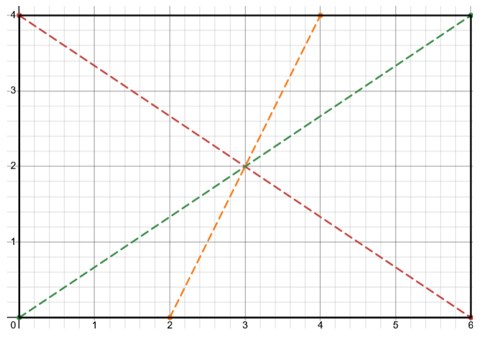
Sample input:

```
((0,0),(6,4))
[((0,0),(6,4)),((2,0),(4,4)),((0,4),(6,0)),((0,3),(6,3))]

((0,0),(2,4))
[((0,4),(2,3)),((1,0),(2,3)),((0,4),(1,0)),((0,2),(1,4)),((0,3),
(2,0)),((0,1),(2,1))]

((0,0),(4,3))
[((4,1),(2,3)),((1,0),(2,3)),((0,3),(1,0)),((0,2),(4,1)),((0,3),
(2,0)),((0,1),(4,2)),((2,0),(4,2)),((1,0),(4,2))]
```

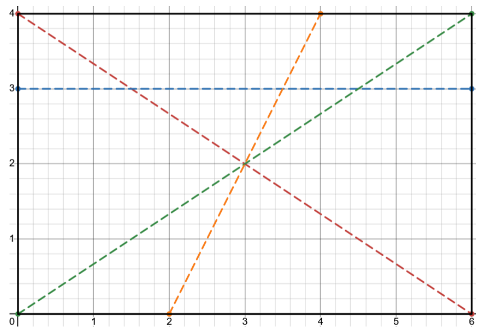
Sample output:

```
10
14
23
```

First output explanation:



Continuing with the example from the problem description, where we have 4 regions after the first two cracks: The third crack (in red) introduces only one more intersection and thus adds two more regions.



The final crack (in blue) makes three different intersections and adds 4 more regions, bringing the total to 10.

EL4: Forest fire (40 points)

To better understand how to stop forest fires, the Ministry of the Environment has entrusted you with the task of modeling the propagation of forest fires to better predict how to stop such a fire. You will be placed in a 20x20 square grid representing different sections of a forest where a specific square has caught fire. A square can have several states:

- Virgin: This square contains a part of the forest, and this part is in good condition.
- Lit: This square has just been ignited by a neighboring fire.
- Flaming: This square is now sufficiently on fire to ignite the adjacent virgin squares.
- Burned: Everything in this square is burnt, and it no longer propagates fire.

You will iterate on the forest after each hour. After each hour, the fire spreads: the lit squares become flaming, and the flaming squares become burned. The propagation rules between squares when the hour changes are as follows:

- If a virgin square is orthogonally adjacent (i.e., shares a side with this square) to exactly one flaming square, it becomes lit in the next hour.
- If a virgin or lit square is orthogonally adjacent to 2 or more flaming squares, it instantly becomes flaming in the next hour.

You will need to determine when there will be no more fire, meaning all the remaining squares are either virgin or burned. A start date and time will be given, and you will need to provide the end time of the fire in the same format. You will never need to change months in the date, only adjust the hour and day based on your result.

Input Specification:

You will receive:

- ***feux:*** an *array* of *tuples* containing pairs of x, y coordinates of the initial fires. The (0, 0) coordinate is at the bottom left and the one next to the right of it (1, 0)
- ***début:*** a *string* containing the date and start time of the fire

Output Specification:

You will need to output:

- a *string* with the same format as the start time that correspond to the end time of the fire

Sample input:

```
[(8,2),(3,3)] "20 janvier, 8:57"
[(0,19)] "5 mars, 11:11"
[(9,9)] "7 décembre, 17:00"
```

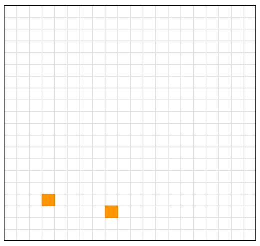
```
[(17,2),(8,11),(13,7)] "22 août, 9:23"
```

Sample output:

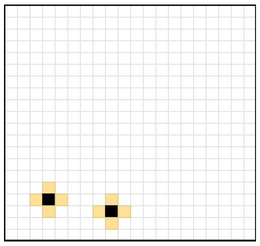
```
"22 janvier, 6:57"
"7 mars, 21:11"
"9 décembre, 0:00"
"23 août, 13:23"
```

First output :

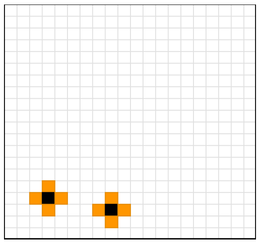
Let's show the first iterations of the forest fire. After placing the two initial fires, the forest looks like this:



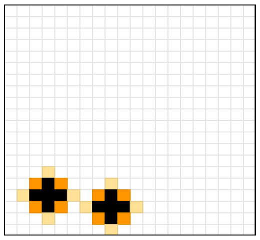
We will use the color orange to signify a flaming square, yellow for a lit square, and black for a burned square. These two fires cause each of their adjacent virgin squares to light up. These squares will become burned in the next hour.



After 1 hour:
There are no flaming squares, so no propagation between squares is possible. The lit squares become flaming.



After 2 hours:
The virgin corners of the 3x3 square centered on the initial fire have two flaming squares as neighbors, which causes them to become flaming in turn. Each flaming square has a neighboring virgin square opposite to the burned square, which only has one flaming neighbor. These squares become lit. Finally, the flaming squares become burned.



After 3 hours:
We repeat the fire propagation pattern until the end. In this case, it continues until the entire forest is burned after 46 hours. 46 hours after January 20 at 8:57 AM gives January 22 at 6:57 AM.

Ragnarök (RK)

RK: The battle of the 6 armies (100 points)

Ragnarök is approaching, and war is brewing with the upcoming clashes of the 6 powerful armies. On one side are the armies of Odin, Thor, and Freya, against the armies of Loki, Hel, and Fenrir. You will be in charge of simulating the epic battles and the great final battle to determine the outcome of Ragnarök. You will need to simulate turn-by-turn conflicts between two opposing armies to determine the winner and their remaining units for the grand final battle.

Thus, you will simulate the initial battles, and all the units that survive these battles will face off in the great battle at the end of humanity. All units with their statistics are directly provided in the template, and the armies are already instantiated for you! You just need to implement the rules for combat and simulate the battles, including the final battle.

Combat rules:

- The army that starts the fight, according to the given matchups, is **the one that appears first** in the combat tuple. Example matchups: [("Odin", "Fenrir"), ("Thor", "Hel"), ("Loki", "Freya")]. This means that in the battle between Odin's army and Fenrir's, Odin's army attacks first.
- In the final battle, the army that attacks first is **the one with the least remaining health** (by adding up the health of all units).
- When an army attacks, **all units with the highest priority score** that are still alive attack one after the other. Each unit of an army attacks one after another. The target of the attack is the unit with the highest priority score. If there are multiple units alive with the same priority score, the one with the most health is attacked.
- Each unit of an army attacks **one after another**. The target of the attack is the unit with the **highest priority score**. If there are multiple units alive with the same priority score, **the one with the most health is attacked**.
- To calculate the damage of a unit, we subtract the defense_value from the attack_value, and the resulting damage is deducted from the target's health.
- The attack turns alternate until one army has no units left alive.
- All the surviving units, along with their remaining health, will be taken to the final battle.
- In the final confrontation, the remaining units from **Odin, Thor, and Freya**'s armies will face off against the surviving units from **Loki, Hel, and Fenrir**'s armies.

Input Specification:

You will receive:

- ***combats:*** An *array* of *tuples* of the initial combats having the first army to attack as the first on of the tuple.
- ***armies***: A *dict* containing a *list* of all the units from each army

Output Specification:

You will need to output:

- an *array* of *strings* of the names of the gods that have survived the final battle

Sample input:

```
[("Odin", "Fenrir"), ("Thor", "Hel"), ("Loki", "Freya")]
```

Sample output:

```
You need to find the output yourself but this is the output
formatting
["Odin", "Freya"]
```

First output explanation:

There is no further explanation, but if you have any questions, go ask CRC.Frank