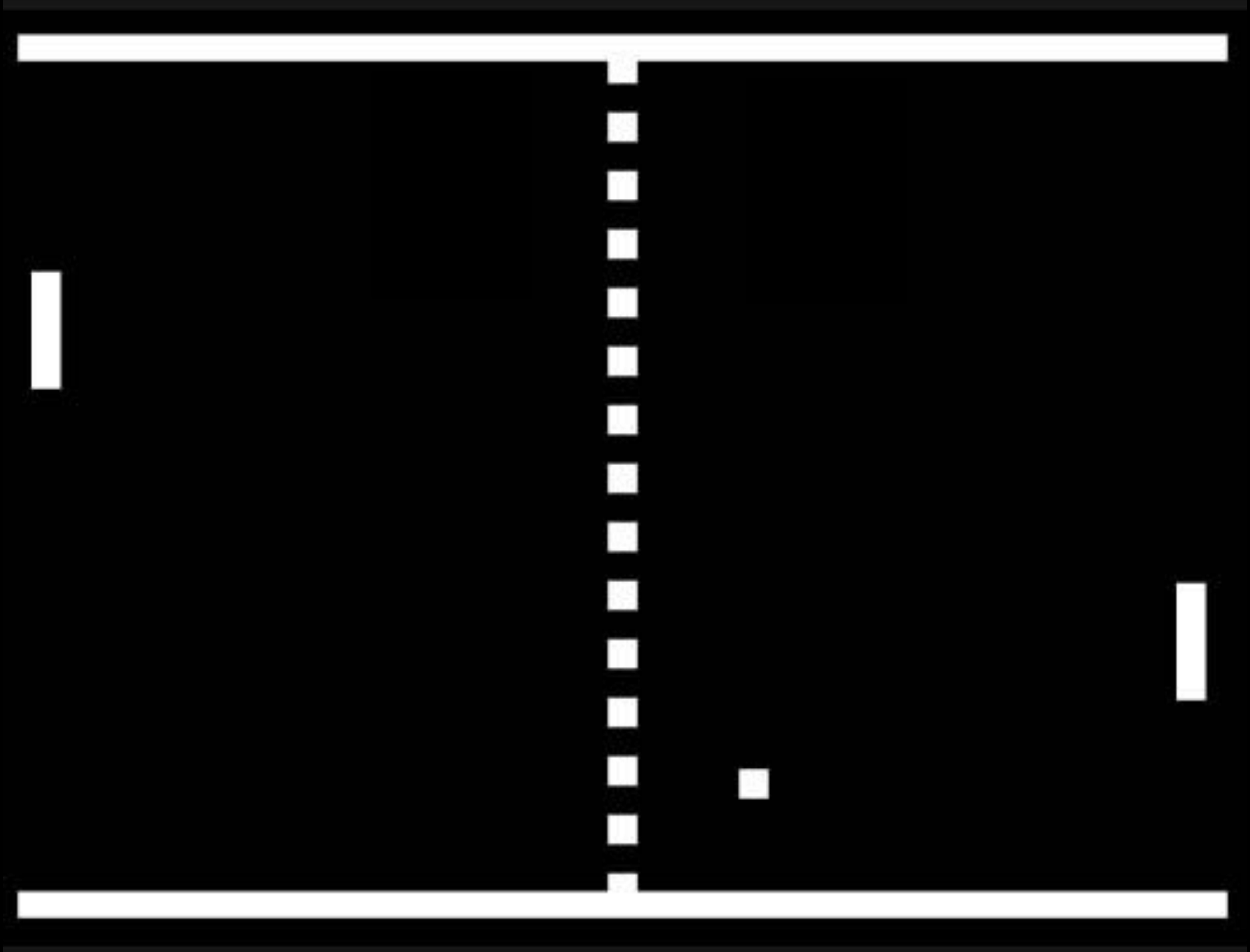
Basic gameplay programming.

PONG
10
15
PLAYER 1
PLAYER 2
ATARI



Movement.

In **setup**

```
float lastFrameTicks = 0.0f;
```

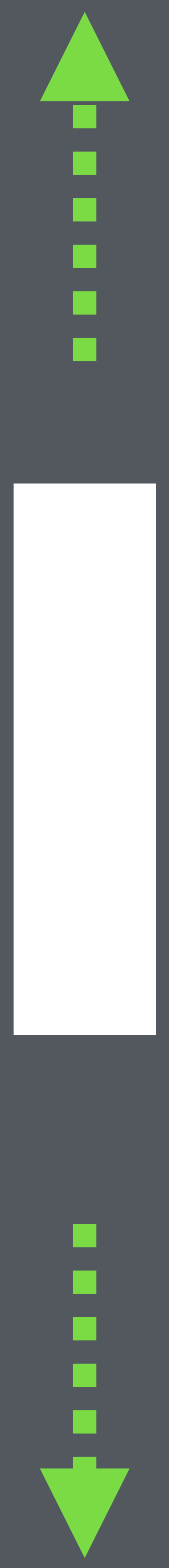
In **game loop**

```
float ticks = (float)SDL_GetTicks()/1000.0f;
float elapsed = ticks - lastFrameTicks;
lastFrameTicks = ticks;
```

elapsed is how many seconds **elapsed since last frame**.

We will use this value to **move everything** in our game.

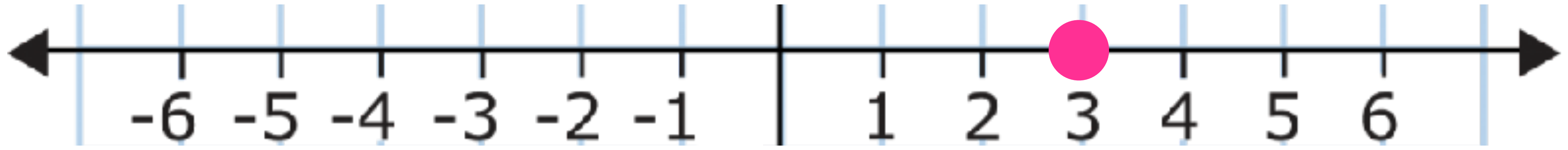
Linear motion.



```
y_position += elapsed *
distance_to_travel_in_one_second
```

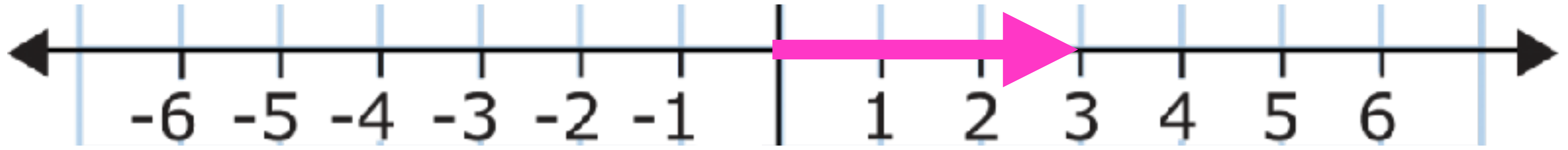
Directional motion.

Vectors.



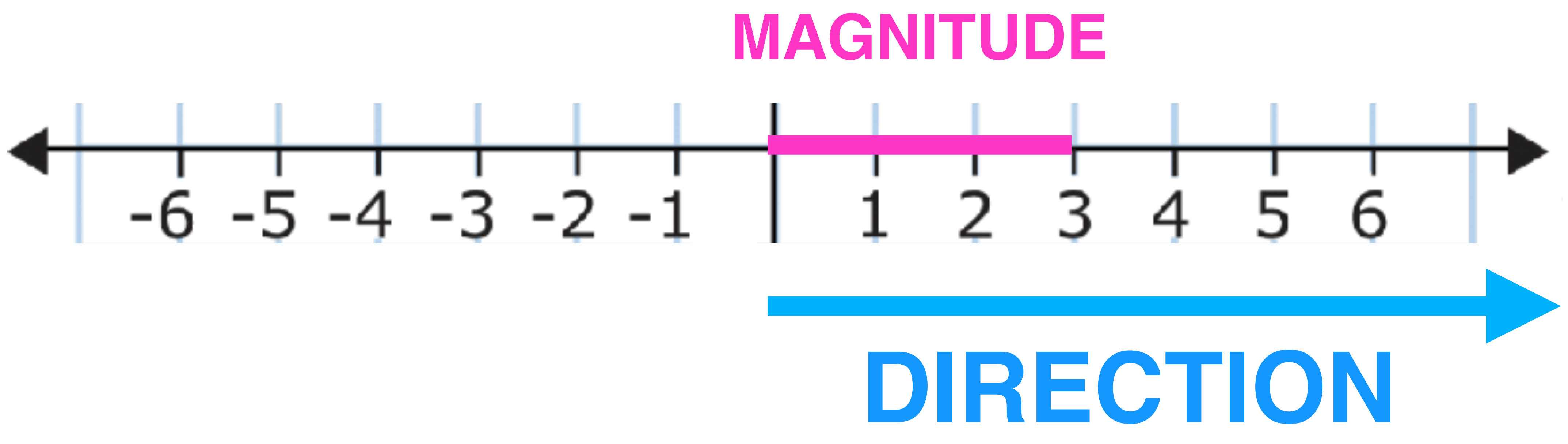
A vector is like a number...

but it has a **magnitude** and a **direction**!



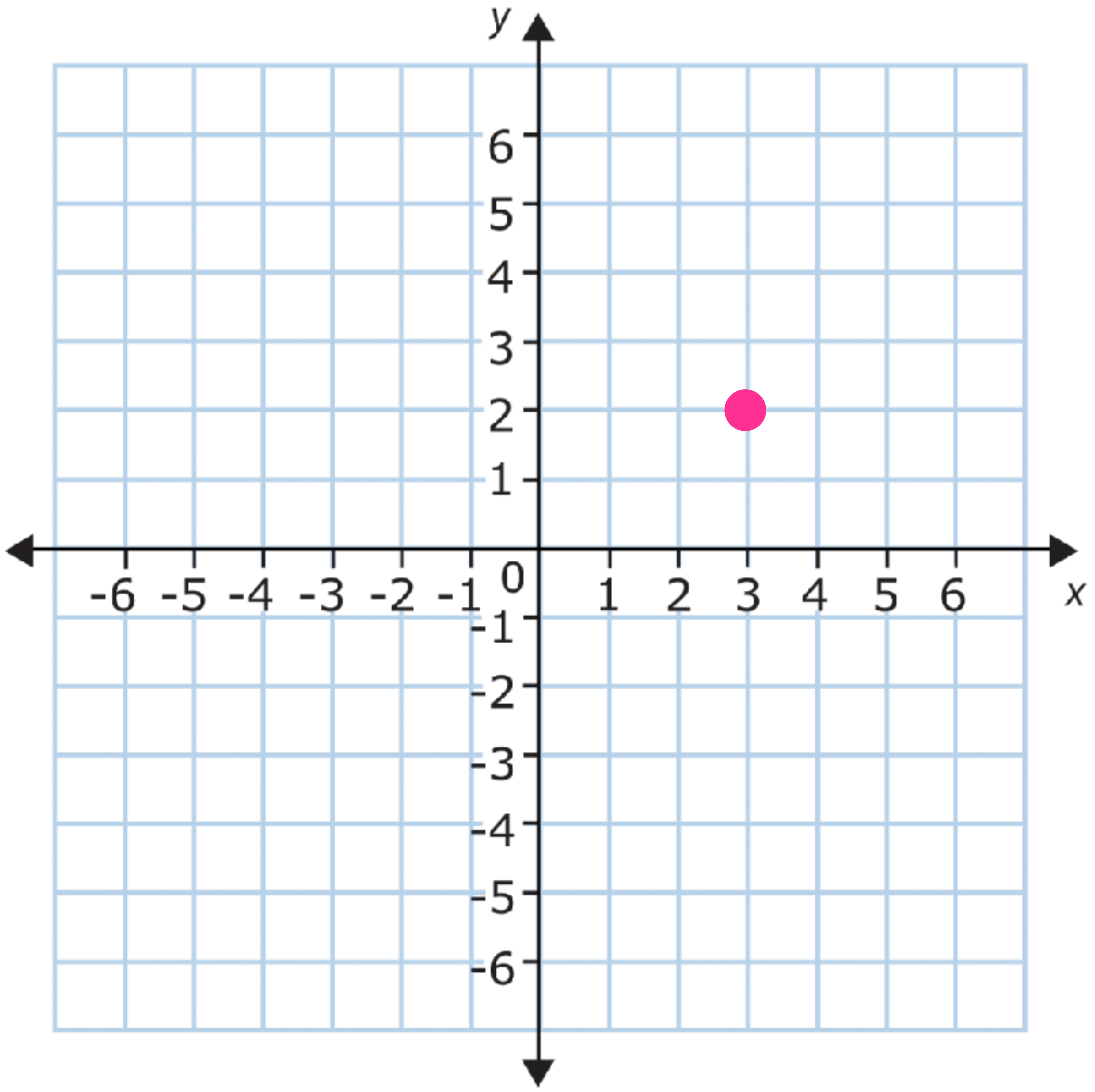
A vector is like a number...

but it has a **magnitude** and a **direction**!

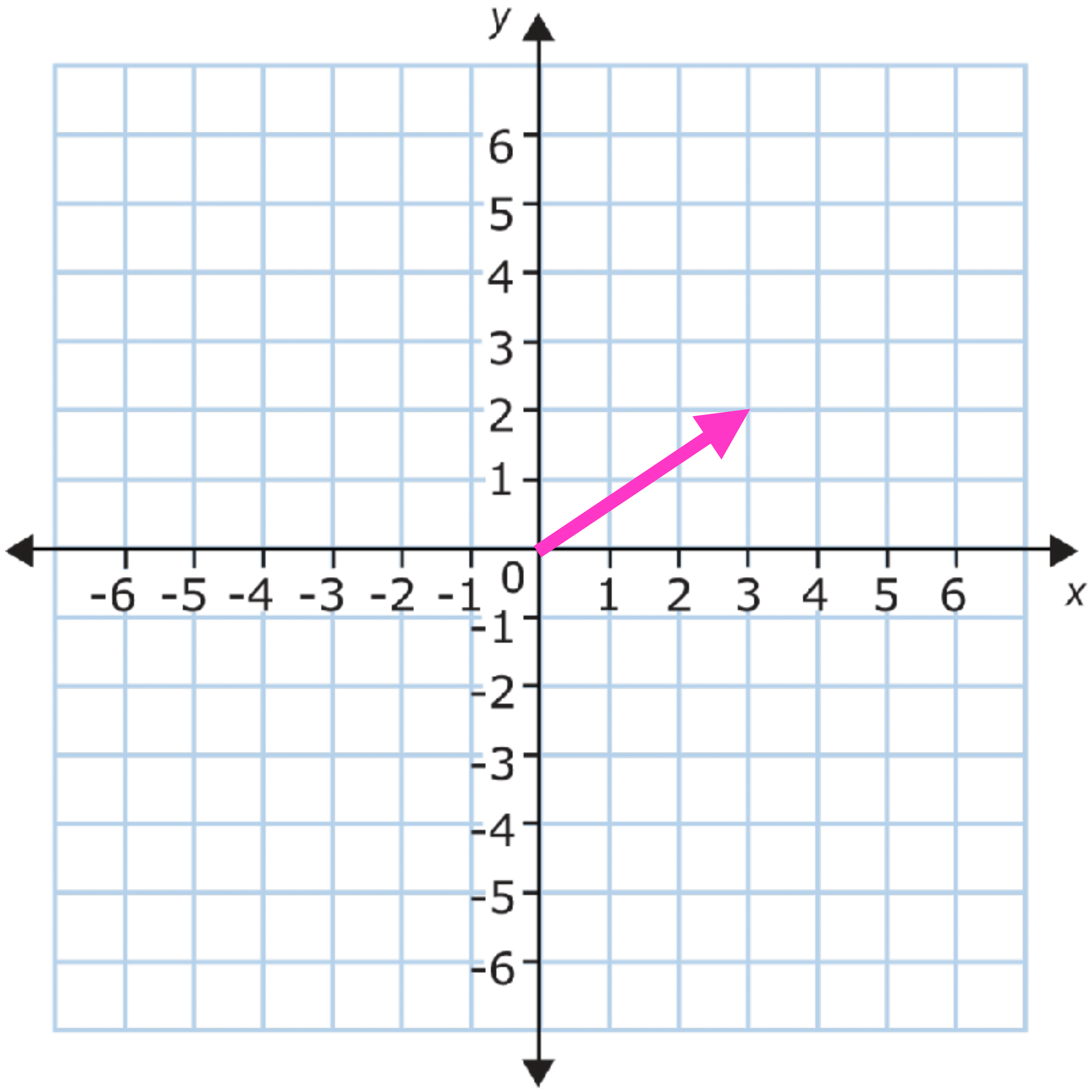


A vector is like a number…

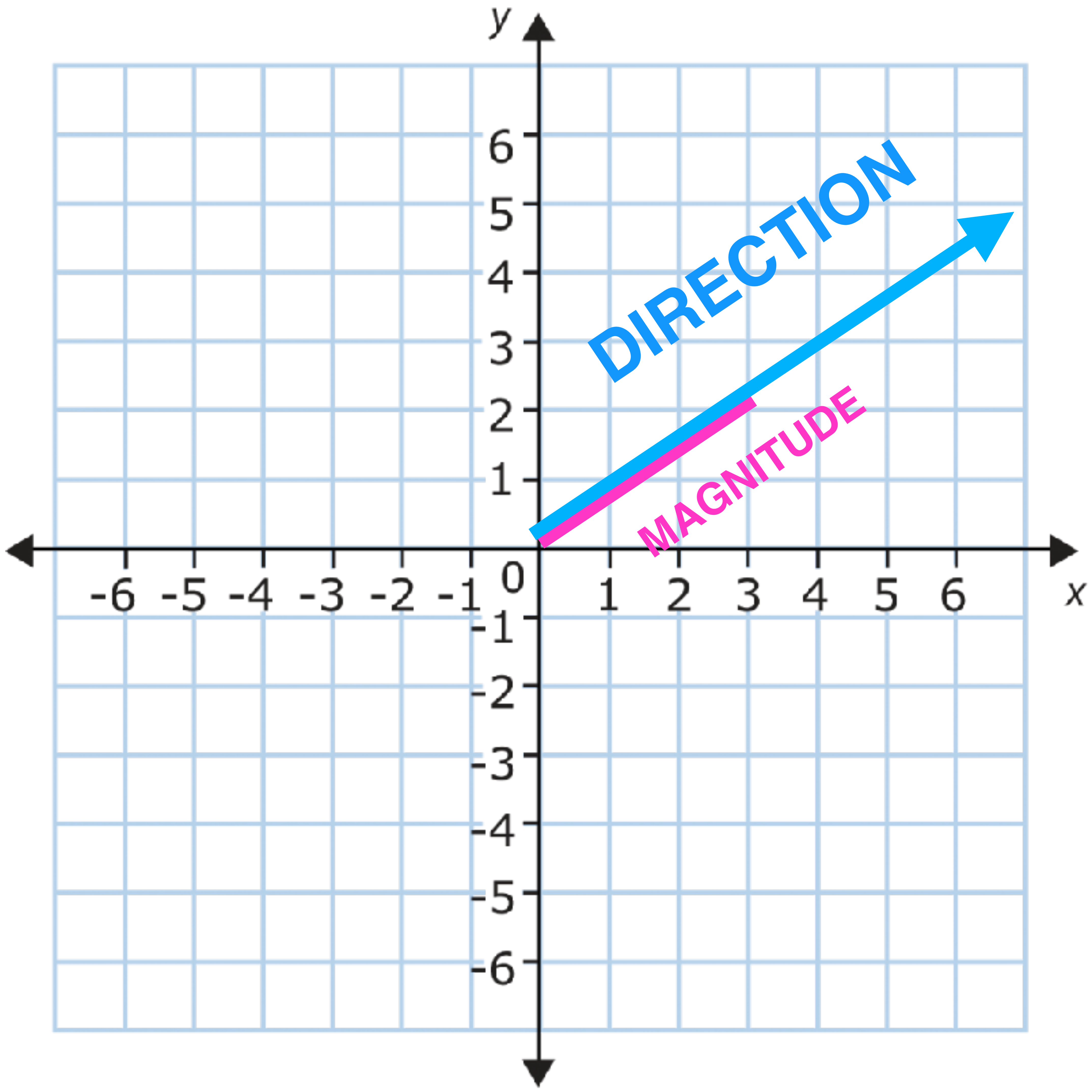
but it has a **magnitude** and a **direction**!



A **2D vector** is like a **2D coordinate**, but has a **magnitude** and a **direction**.



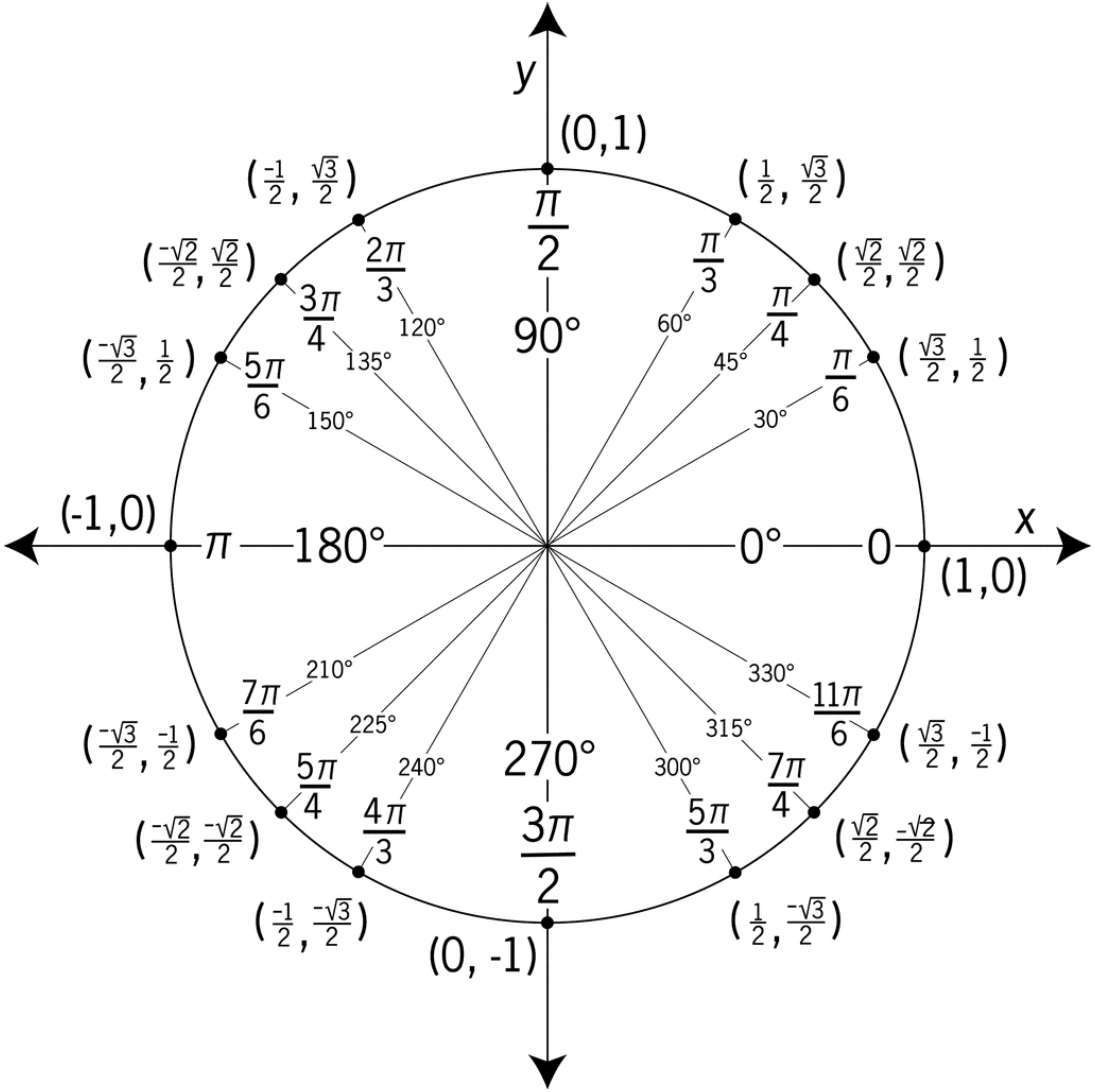
A **2D vector** is like a **2D coordinate**, but has a **magnitude** and a **direction**.

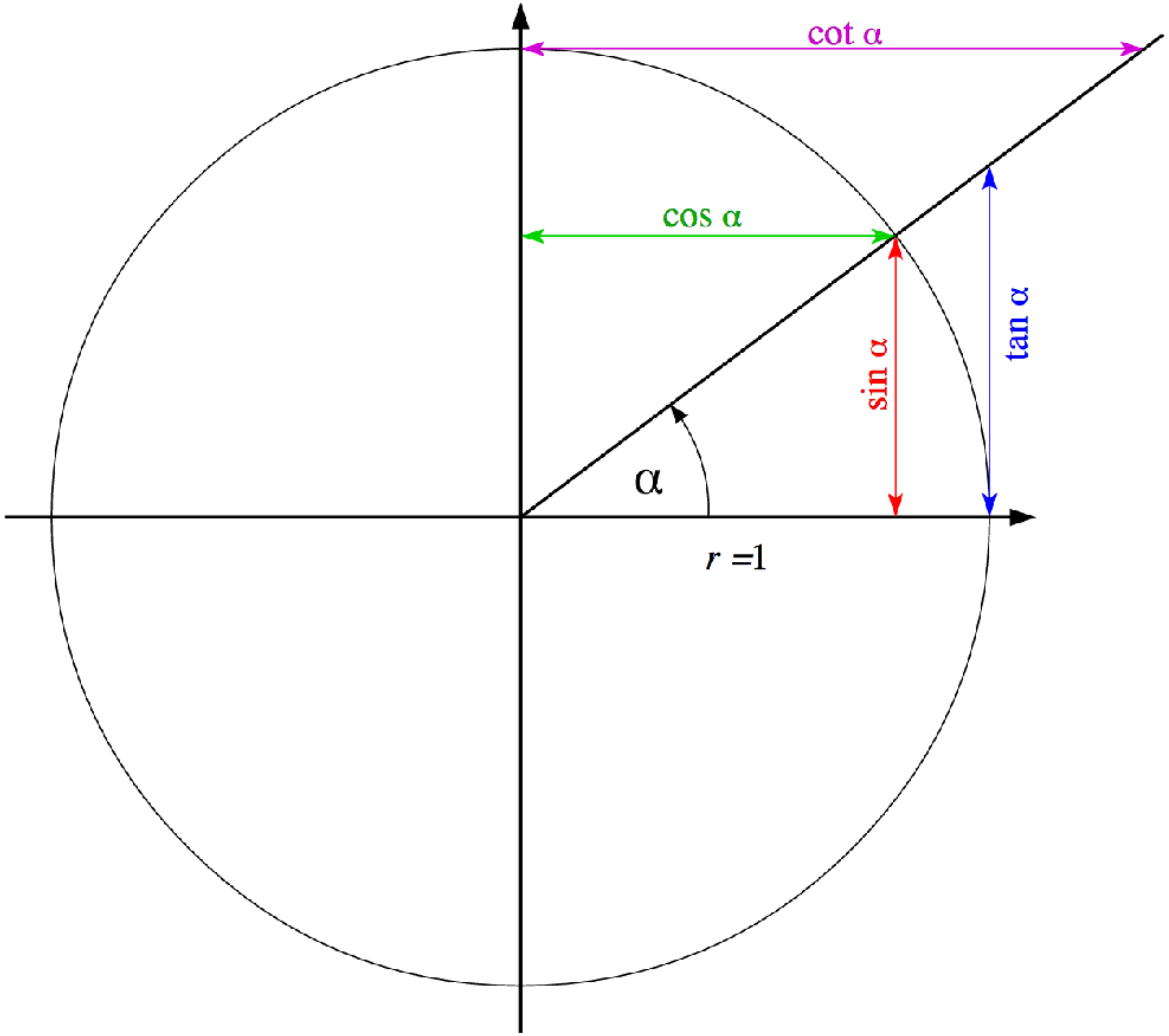


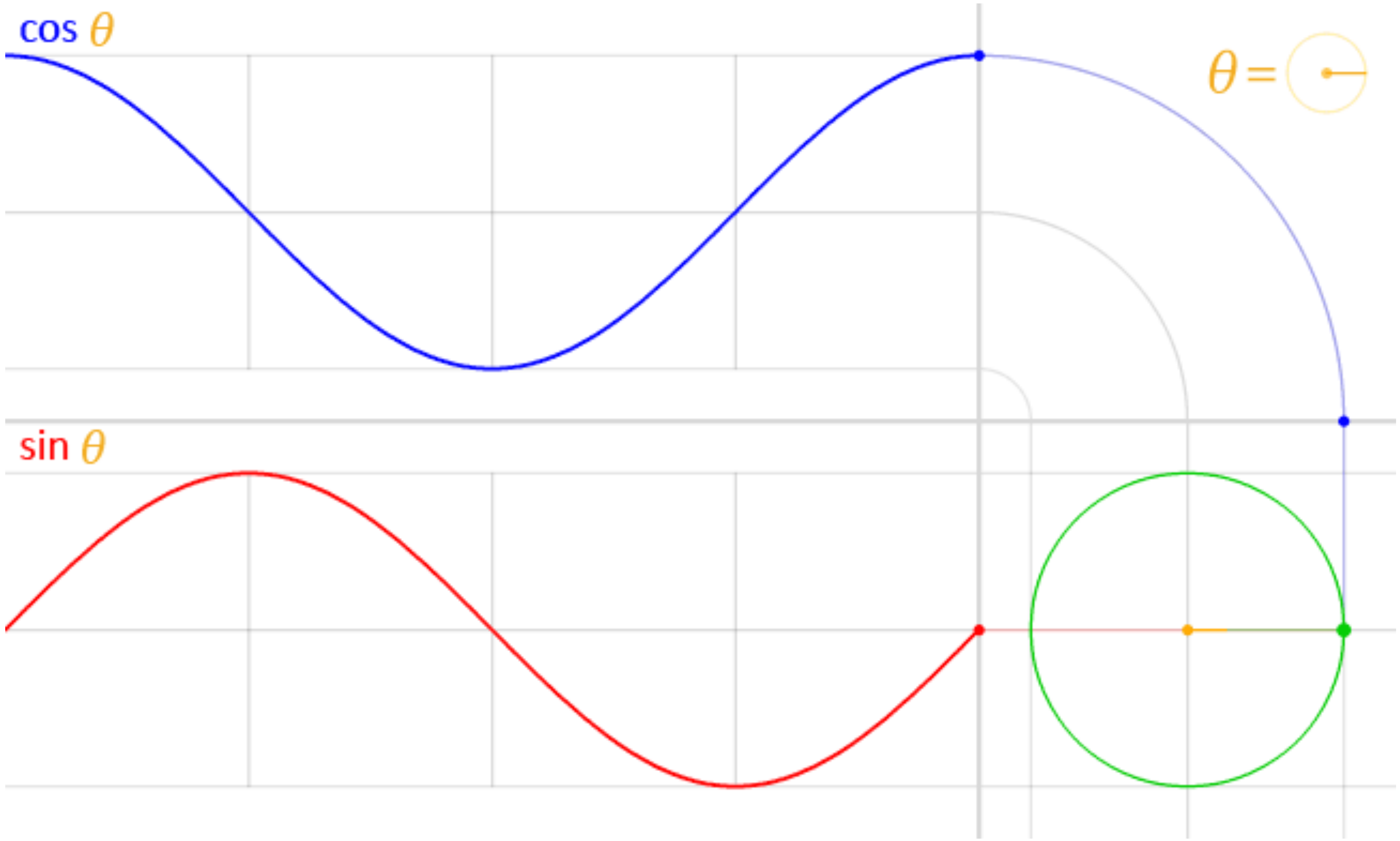
A **2D vector** is like a **2D coordinate**, but has a **magnitude** and a **direction**.

2D direction?

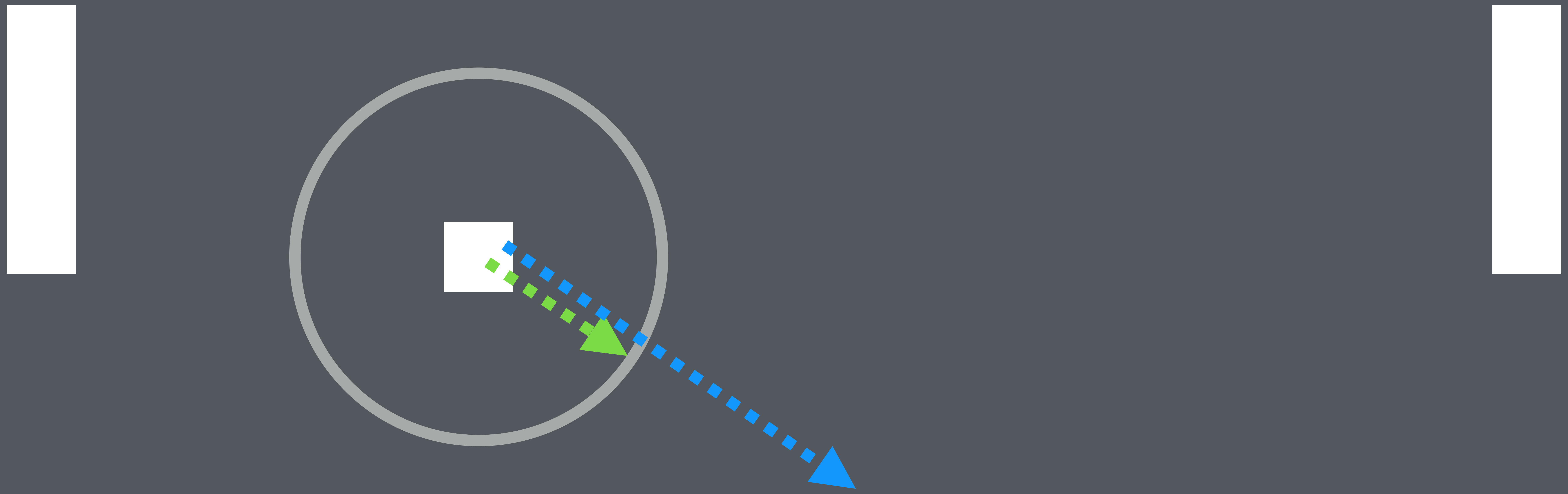
Unit vector!


y
x
(0,1)
$\frac{\pi}{2}$
90°
$\left(\frac{-1}{2}, \frac{\sqrt{3}}{2}\right)$
$\left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$
$\frac{2\pi}{3}$
120°
$\frac{\pi}{3}$
60°
$\left(\frac{-\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$
$\left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$
$\frac{3\pi}{4}$
135°
$\frac{\pi}{4}$
45°
$\left(\frac{-\sqrt{3}}{2}, \frac{1}{2}\right)$
$\left(\frac{\sqrt{3}}{2}, \frac{1}{2}\right)$
$\frac{5\pi}{6}$
150°
$\frac{\pi}{6}$
30°
(-1,0)
π
180°
0°
0
(1,0)
210°
$\frac{7\pi}{6}$
330°
$\frac{11\pi}{6}$
$\left(\frac{-\sqrt{3}}{2}, \frac{-1}{2}\right)$
$\left(\frac{\sqrt{3}}{2}, \frac{-1}{2}\right)$
225°
$\frac{5\pi}{4}$
315°
$\frac{7\pi}{4}$
240°
$\frac{4\pi}{3}$
270°
$\frac{3\pi}{2}$
300°
$\frac{5\pi}{3}$
$\left(\frac{-\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}\right)$
$\left(\frac{\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}\right)$
$\left(\frac{-1}{2}, \frac{-\sqrt{3}}{2}\right)$
$\left(\frac{1}{2}, \frac{-\sqrt{3}}{2}\right)$
(0, -1)


cot α
cos α
sin α
tan α
α
r =1

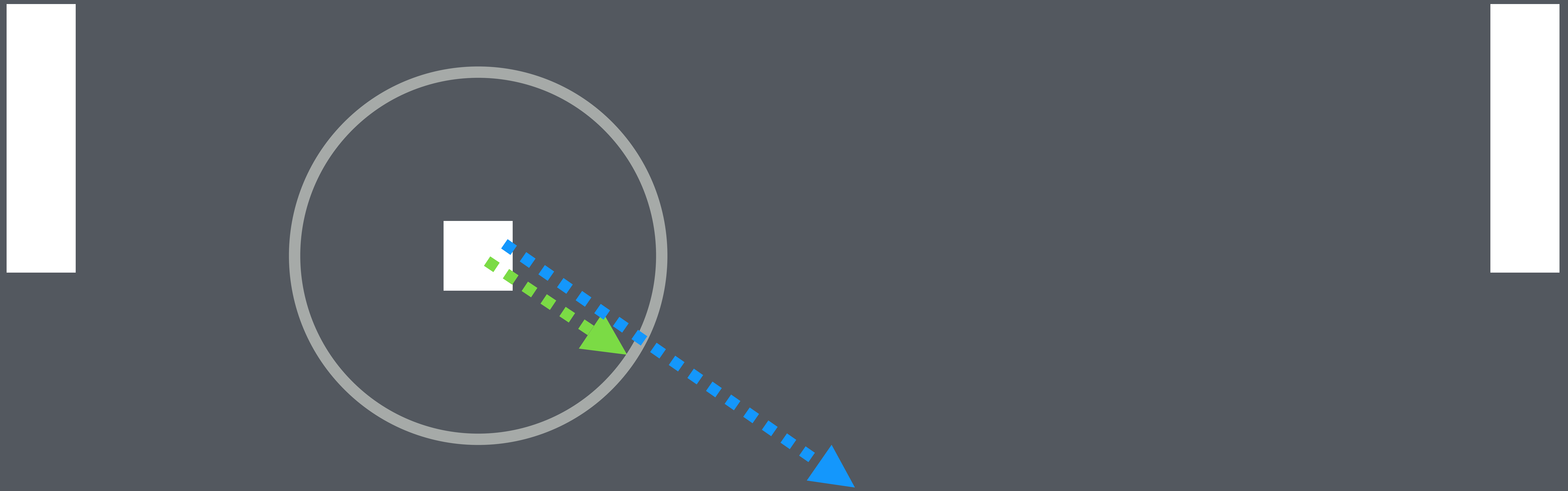

cos θ
θ =
sin θ

position += direction_vector * elapsed * units_a_second



```
position.x += cos(angle) *
elapsed * units_a_second
```

```
position.y += sin(angle) *
elapsed * units_a_second
```



Reading keyboard input.

Polling input vs. input events.

Polling input

Checking to see if a key is **pressed**.
Useful for **continuous player actions**, such as **movement**, or checking modifier keys.

Uint8 *SDL_GetKeyboardState(int *numkeys);

Returns a **pointer to an array of key states**. A value of **1 means that the key is pressed** and a **value of 0 means that it is not**. Indexes into this array are obtained by using **SDL scancode values**. The pointer returned is a pointer to an internal SDL array. It will be valid for the whole lifetime of the application and **should not be freed by the caller**. We can pass it a pointer to an int if we want to know the size of the array.

```
const Uint8 *keys = SDL_GetKeyboardState(NULL);

if(keys[SDL_SCANCODE_LEFT]) {
    // go left!
} else if(keys[SDL_SCANCODE_RIGHT]) {
    // go right!
}
```

SDL scancodes:

All start with SDL_SCANCODE_

Full list here:

https://wiki.libsdl.org/SDL_Scancode

Input events.

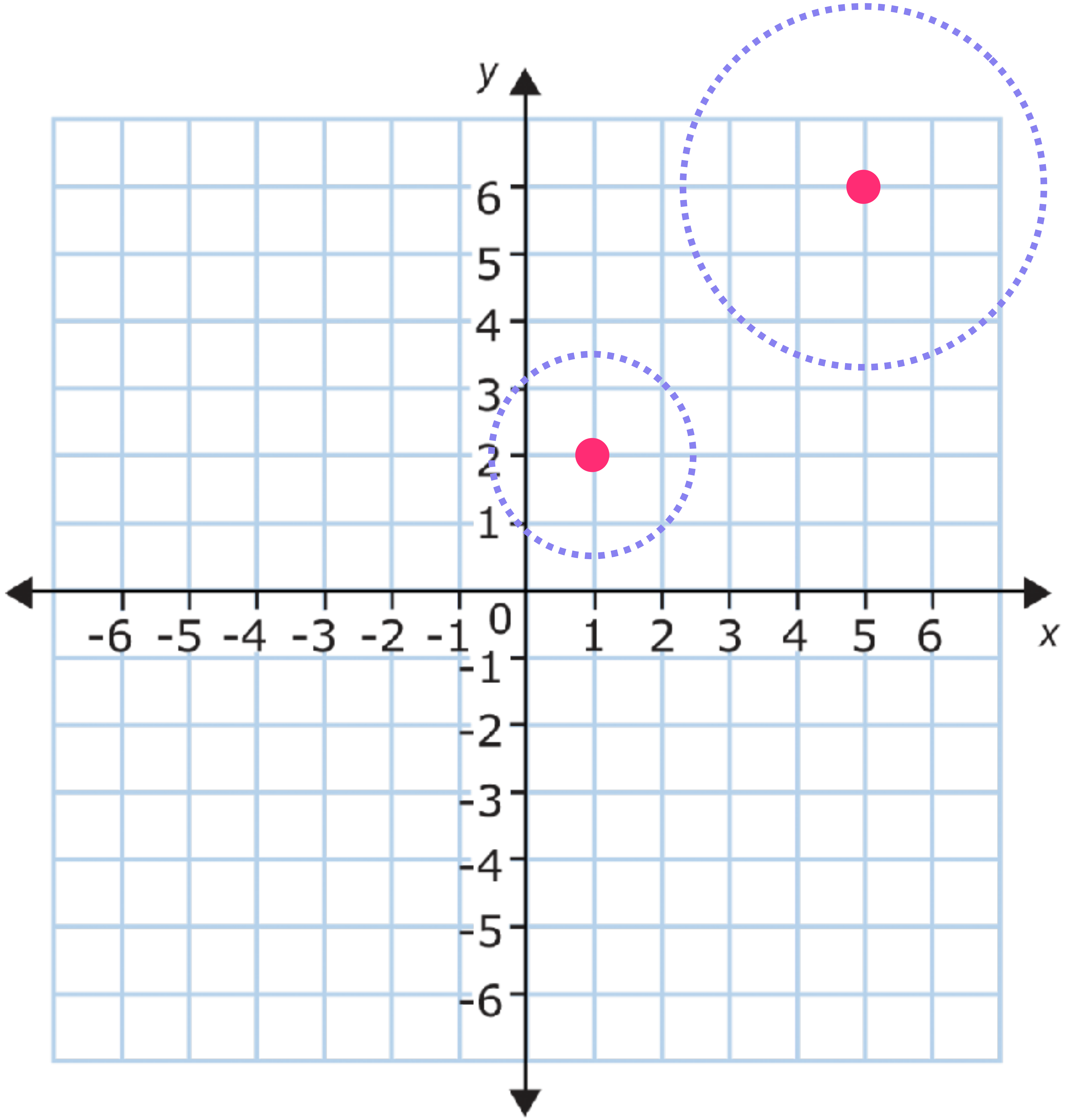
Knowing **exactly when the player pressed or released a key**. Useful for **action events** like **shooting** or **jumping**.

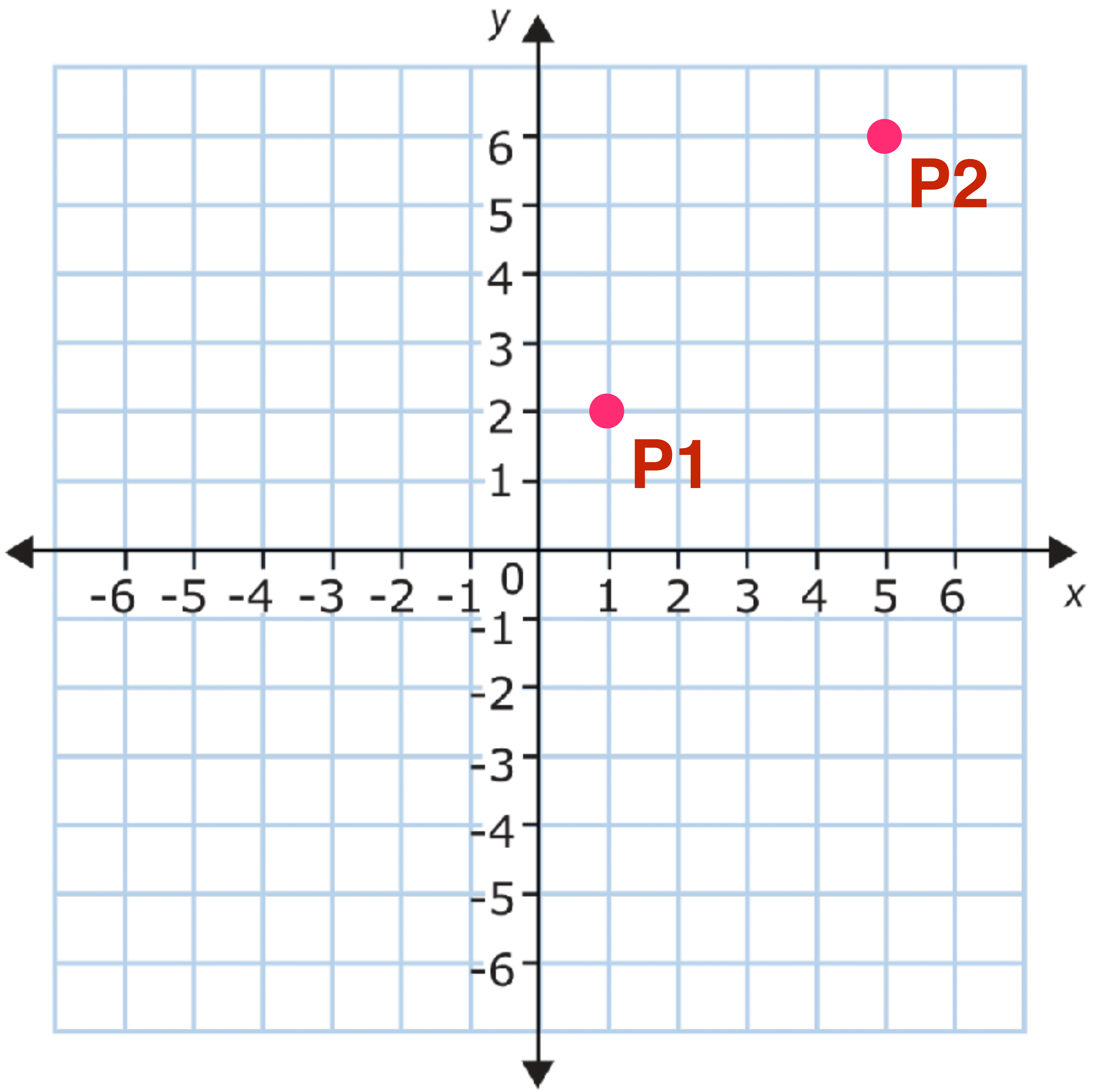
To read **input events**, we use our **event loop** to see if the event has a **type** of **SDL_KEYDOWN** or **SDL_KEYUP**. We can then **check the key** that was pressed or released by checking the **key** member of the **SDL event structure**.

```
while (SDL_PollEvent(&event)) {
    if (event.type == SDL_QUIT || event.type == SDL_WINDOWEVENT_CLOSE) {
        done = true;
    } else if(event.type == SDL_KEYDOWN) {
        if(event.key.keysym.scancode == SDL_SCANCODE_SPACE) {
            // DO AN ACTION WHEN SPACE IS PRESSED!
        }
    }
}
```

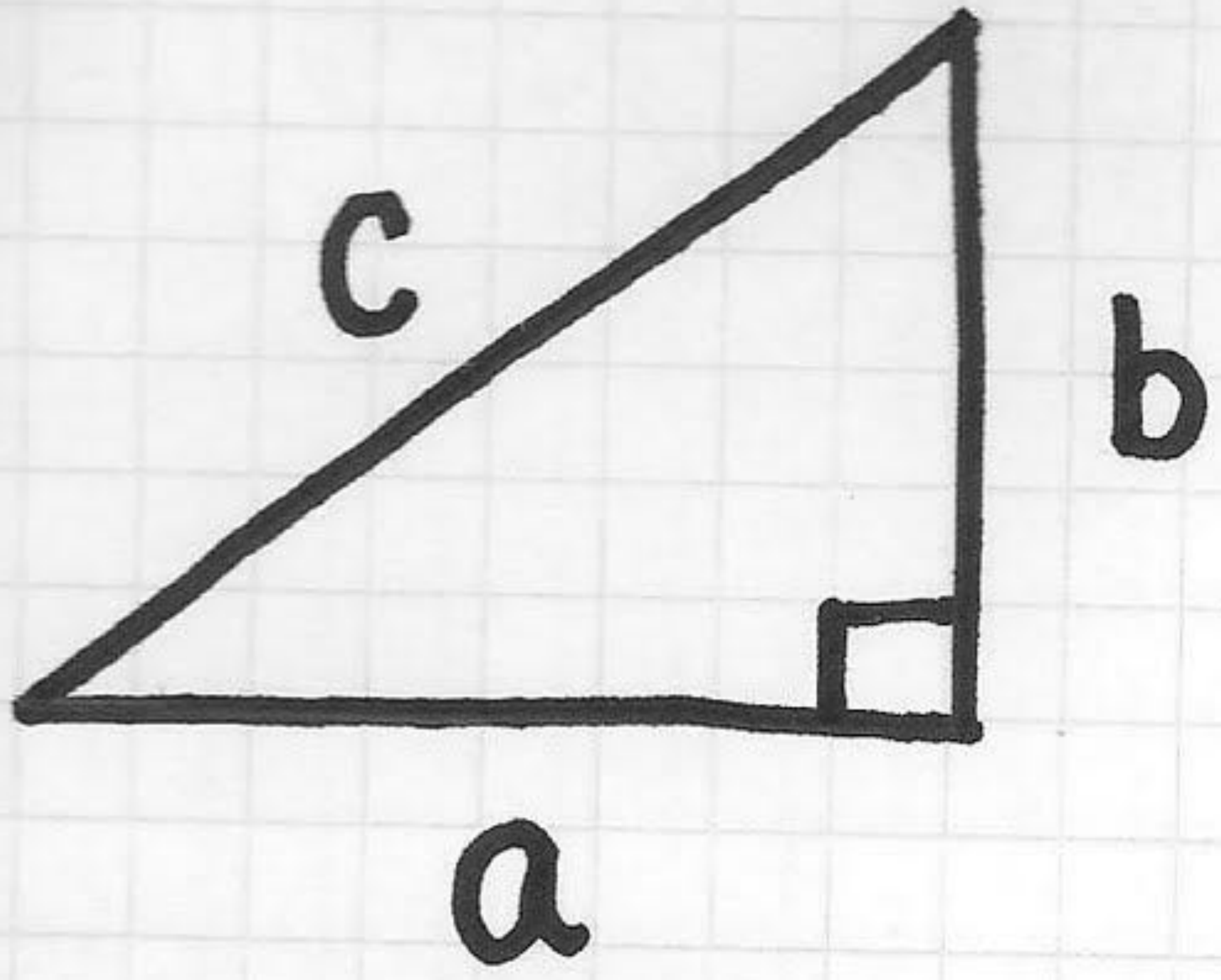
Collision detection.

Circle - circle collision detection.

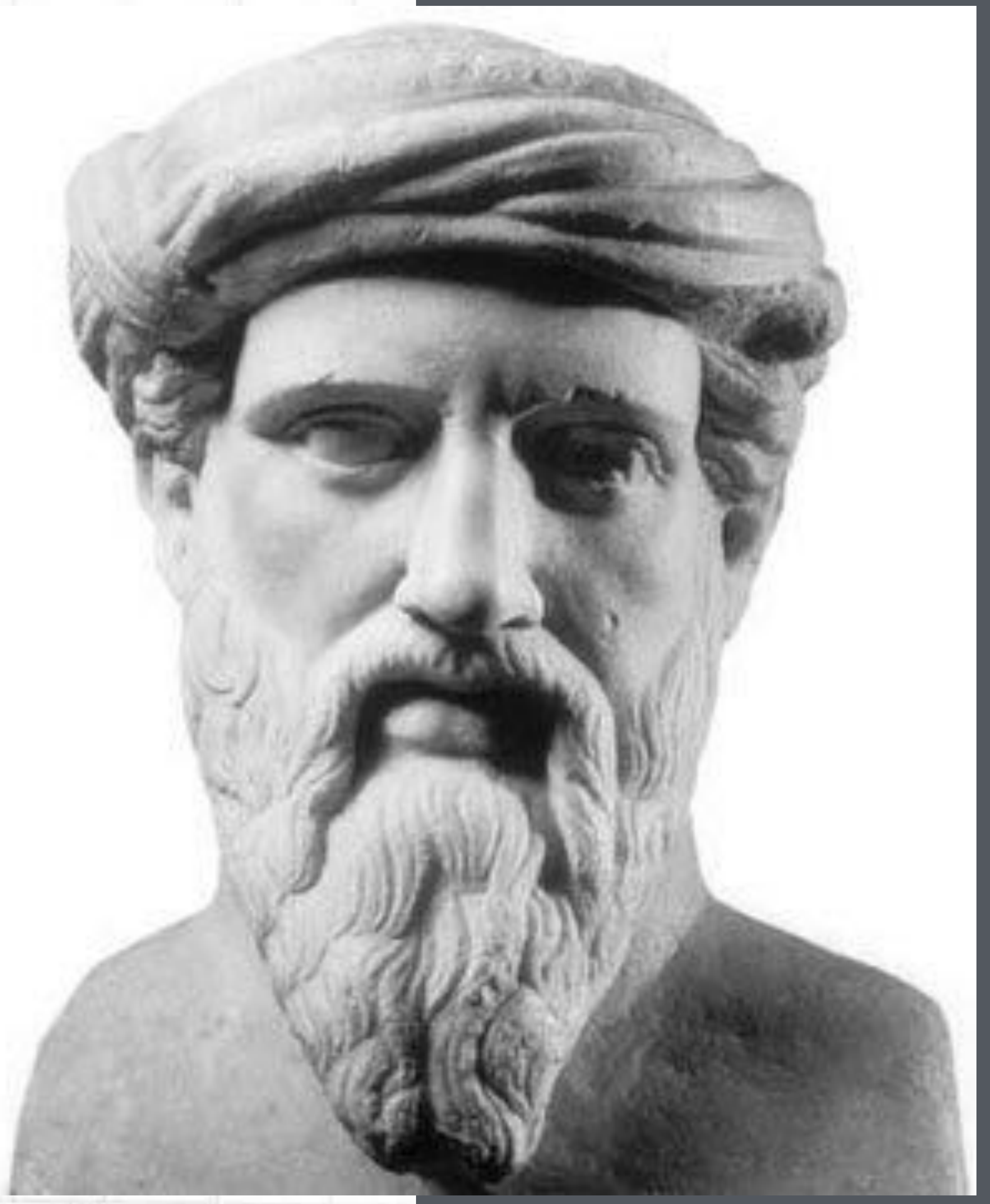

y
x
0
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
6
5
4
3
2
1
-1
-2
-3
-4
-5
-6


y
x
0
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
P1
P2

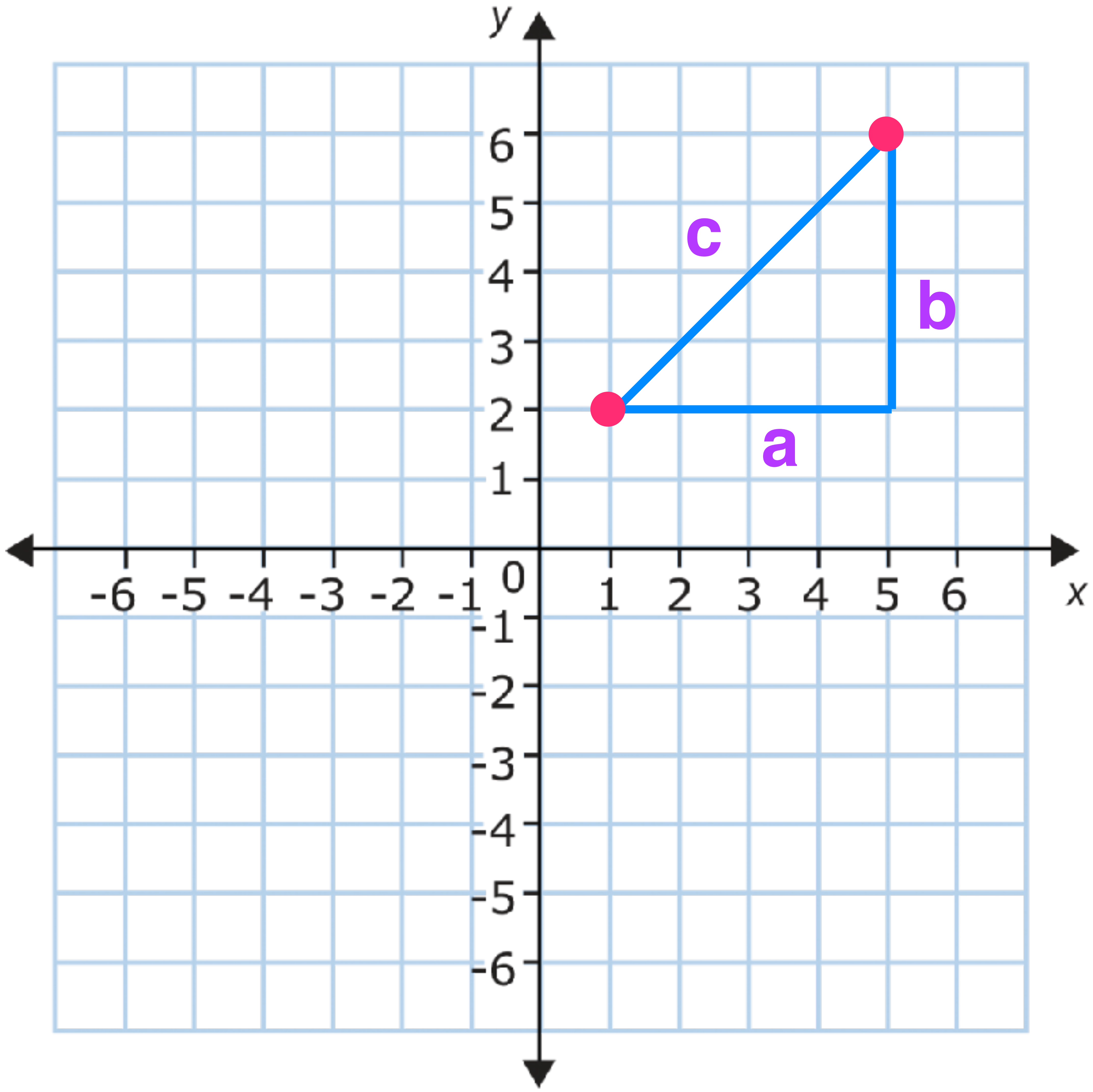
Pythagorean theorem

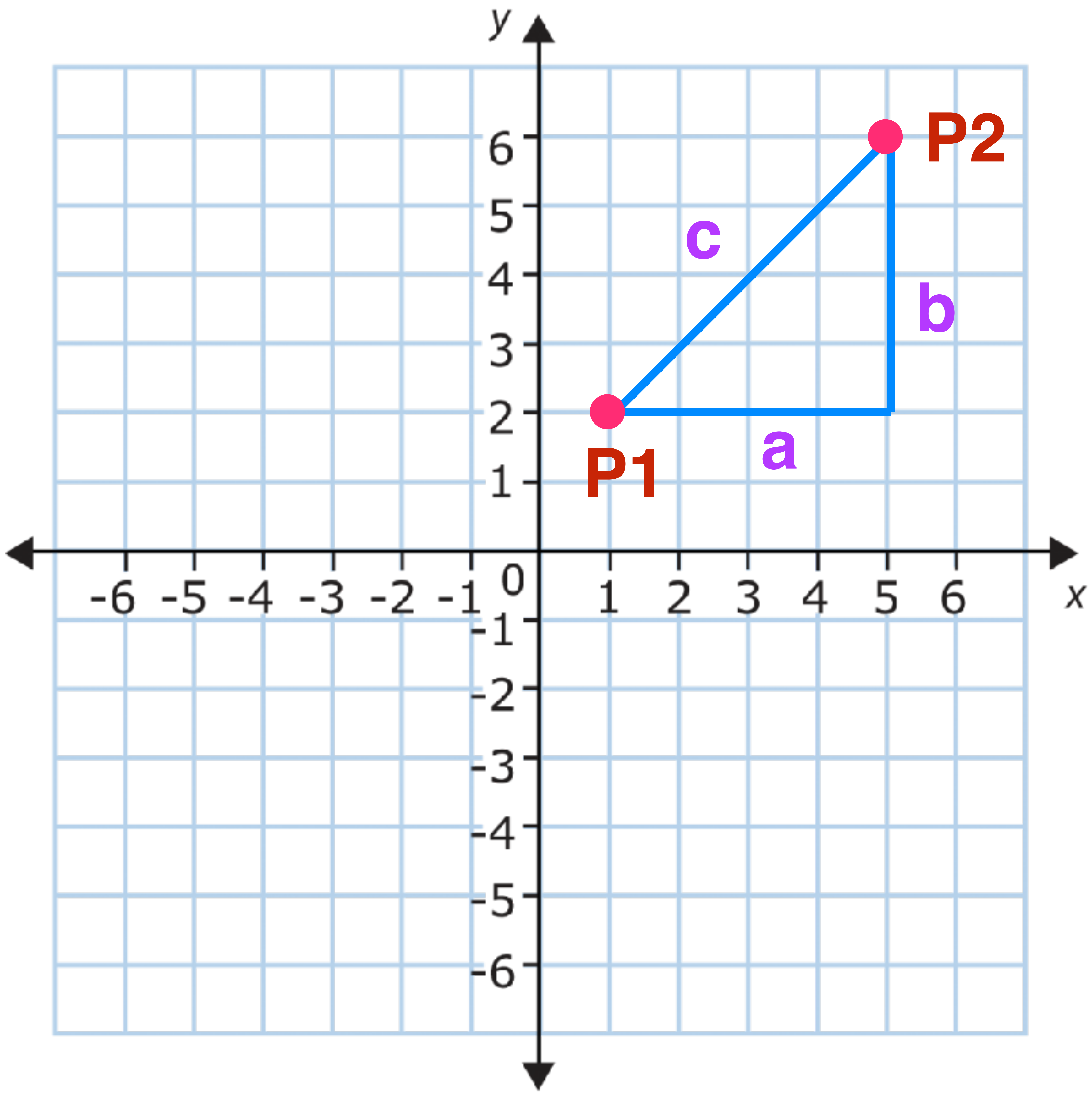


$$a^2 + b^2 = c^2$$



Distance between 2 points.





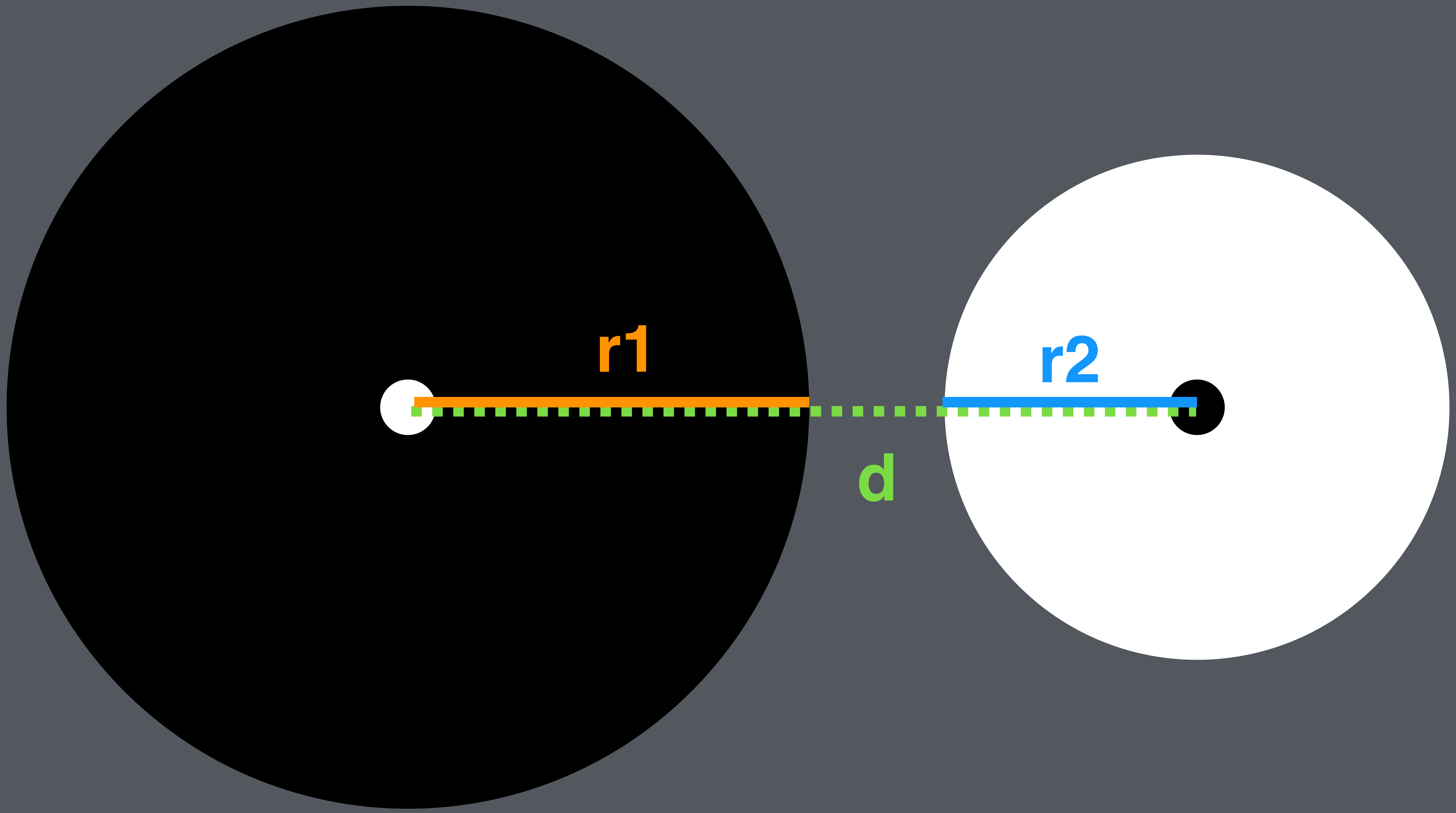
$$a = x_2 - x_1$$

$$b = y_2 - y_1$$

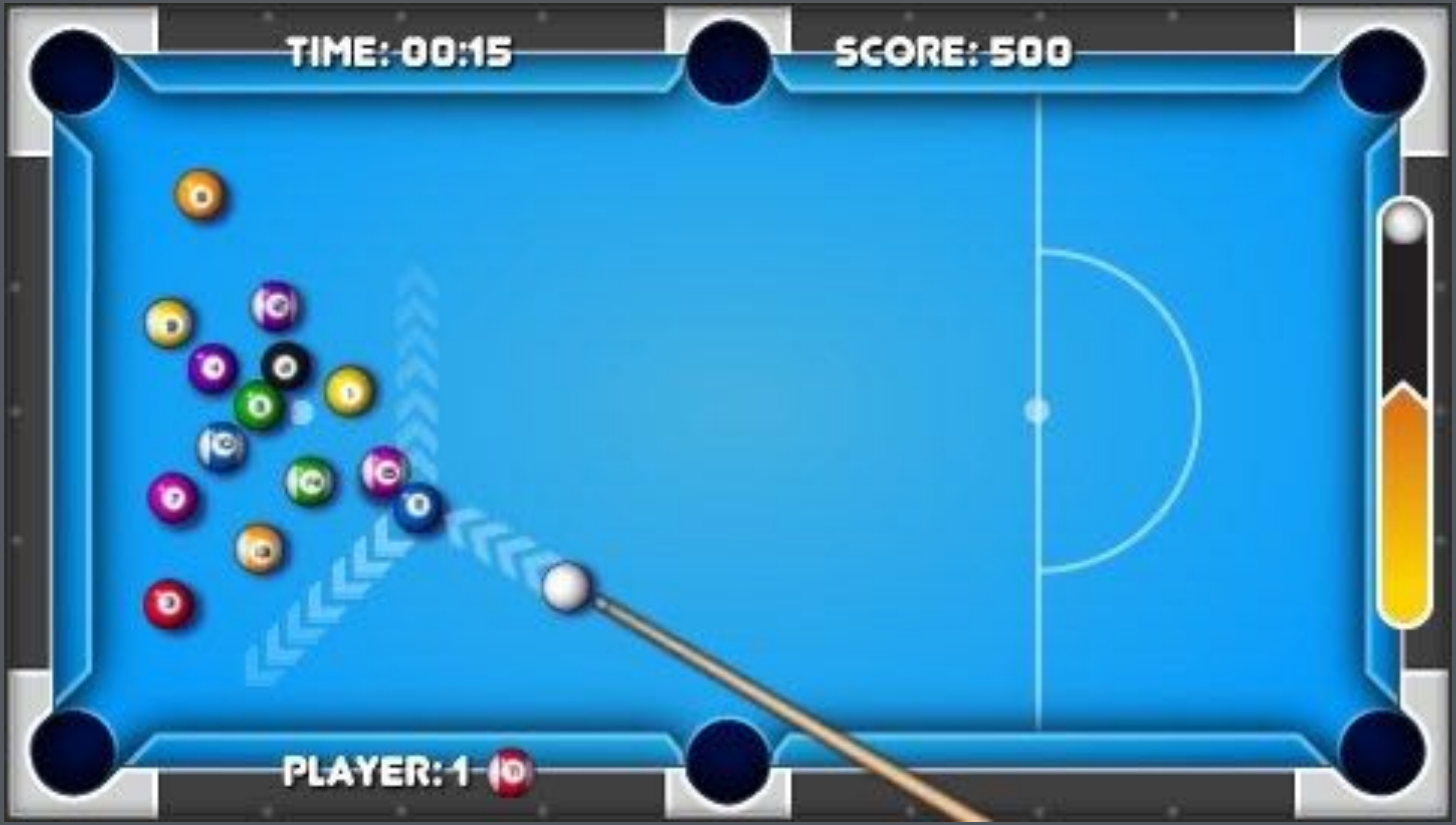
$$c^2 = a^2 - b^2$$

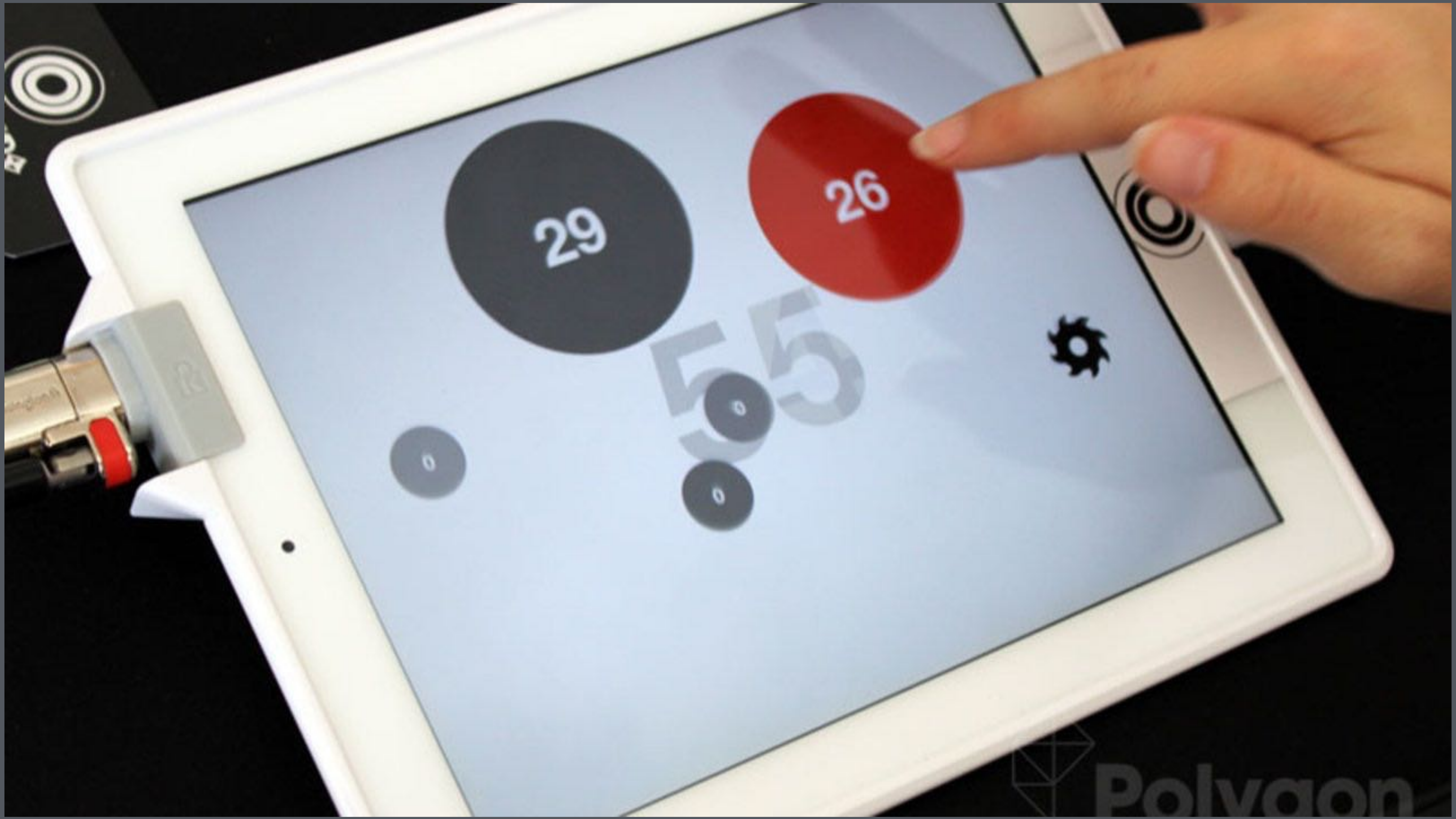
$$c = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Circle - circle collision detection.

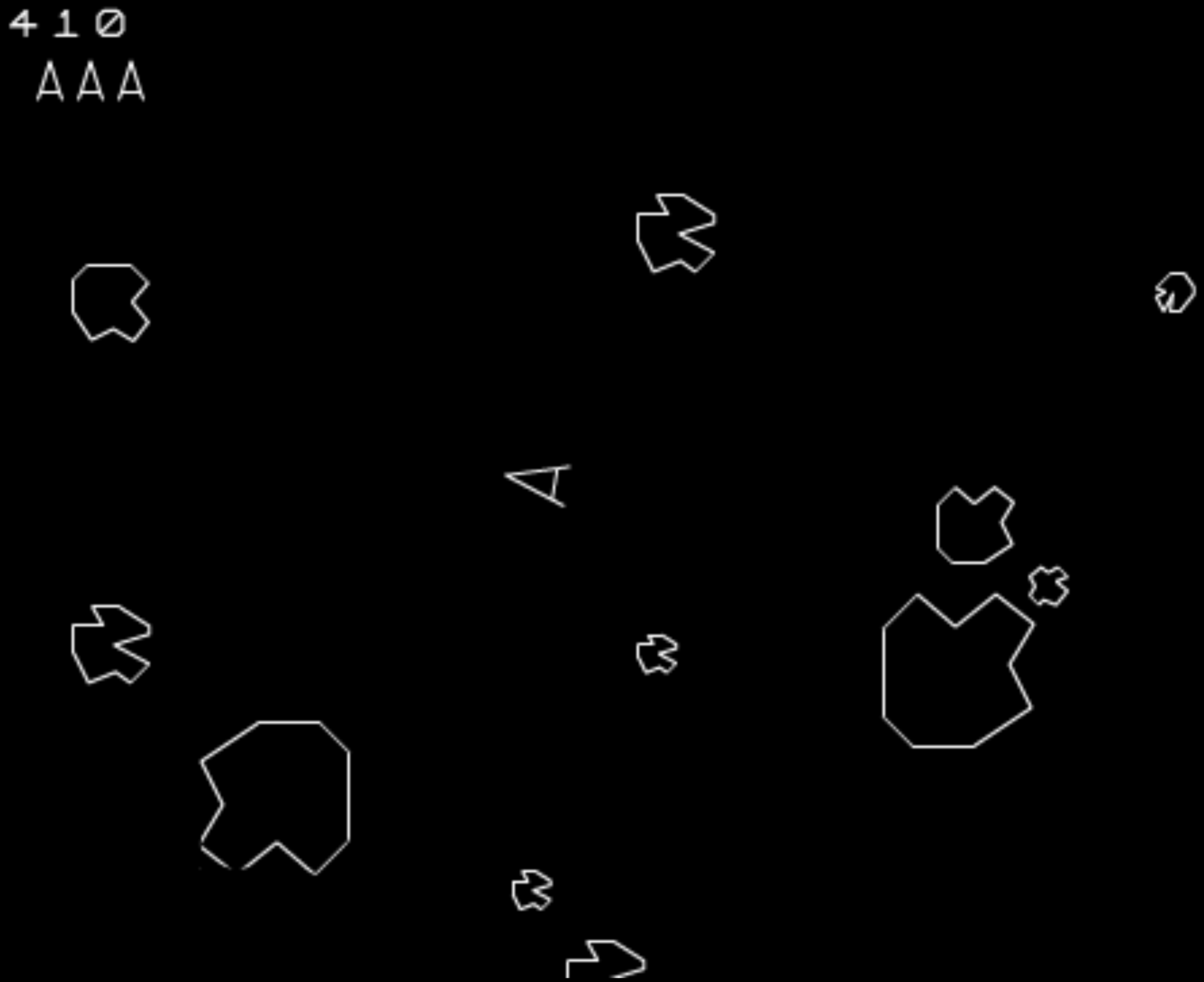


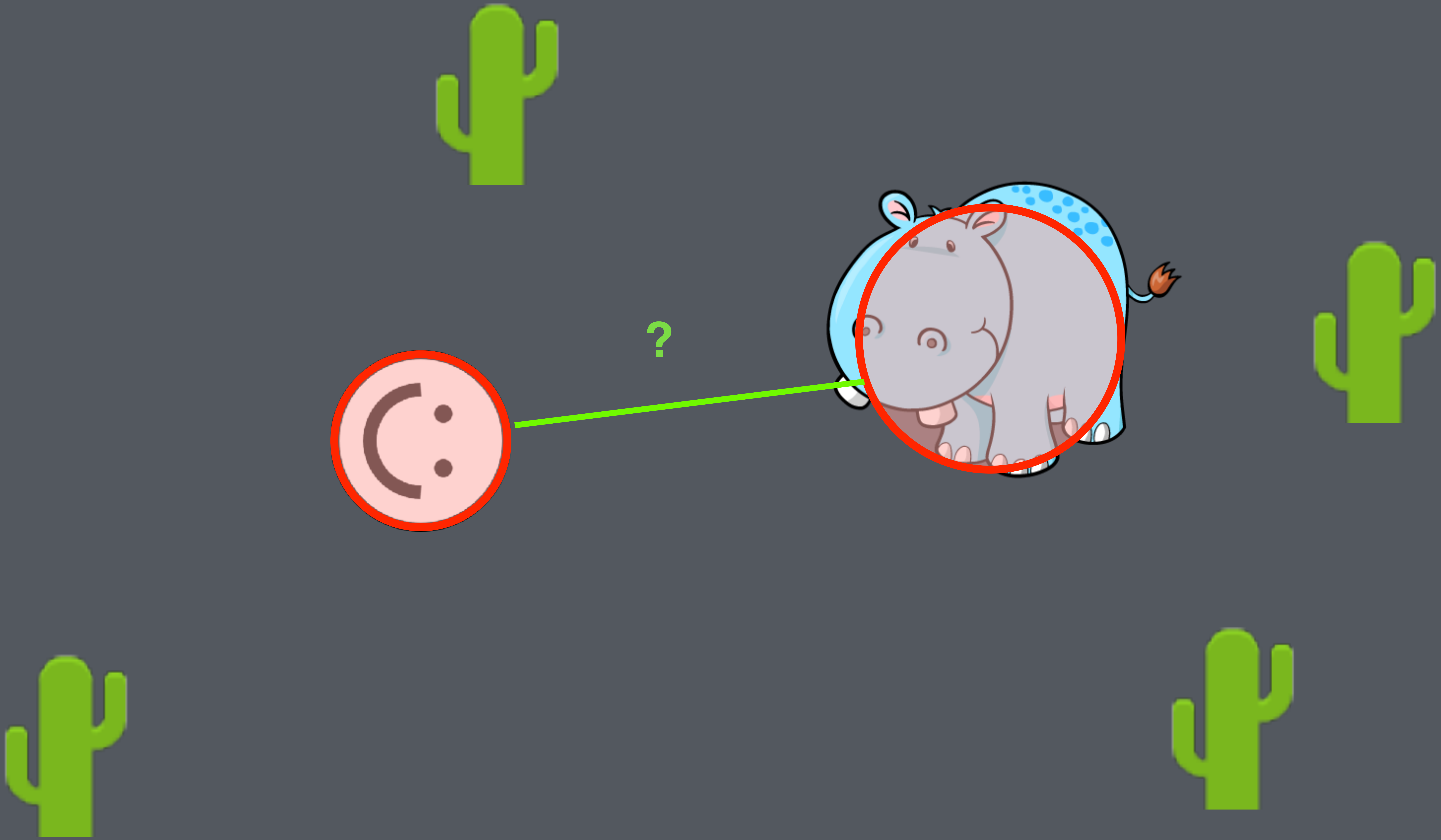
If the distance **between two circles** is **less than or equal to the sum of their radii**, the circles are **colliding**!


TIME: 00:15
SCORE: 500
PLAYER: 1

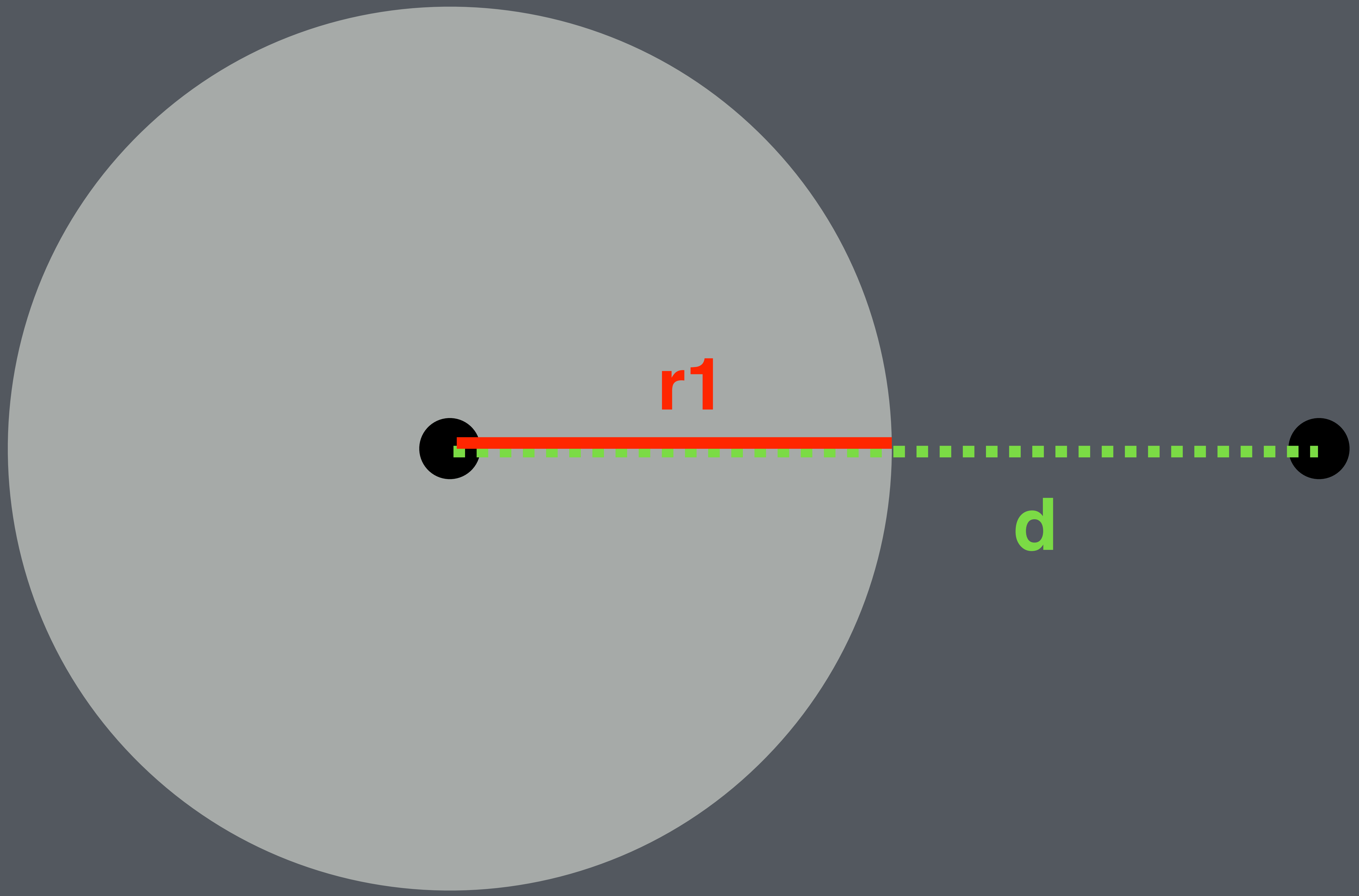


29
26
55
0
0
0
Polygon


41Ø

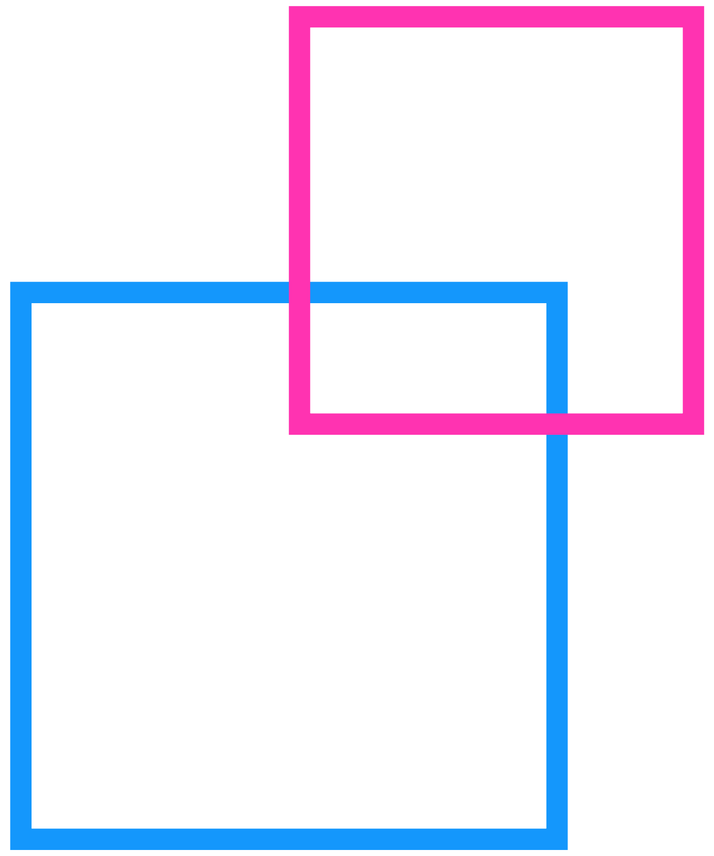


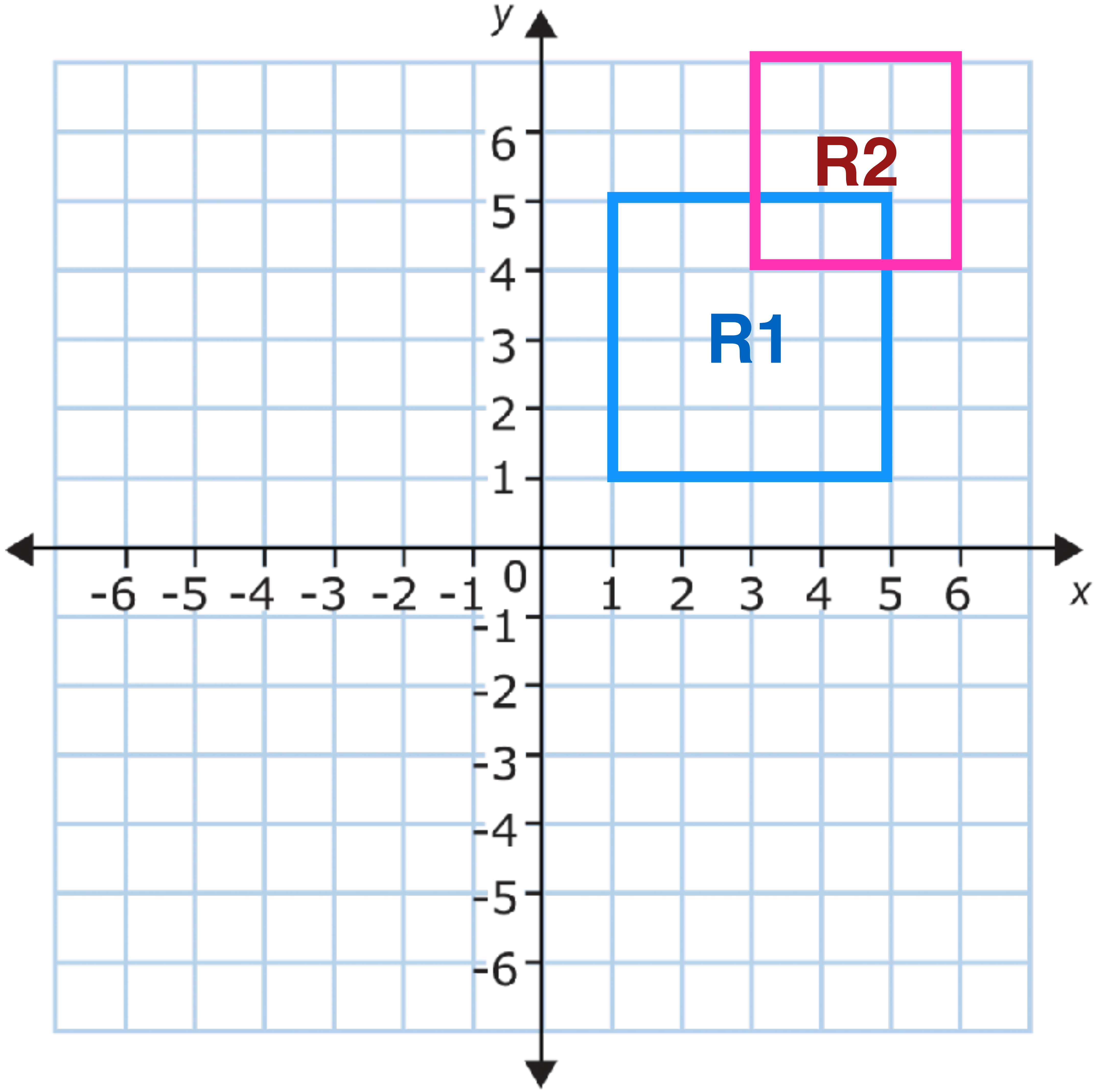
Circle-point collision detection.

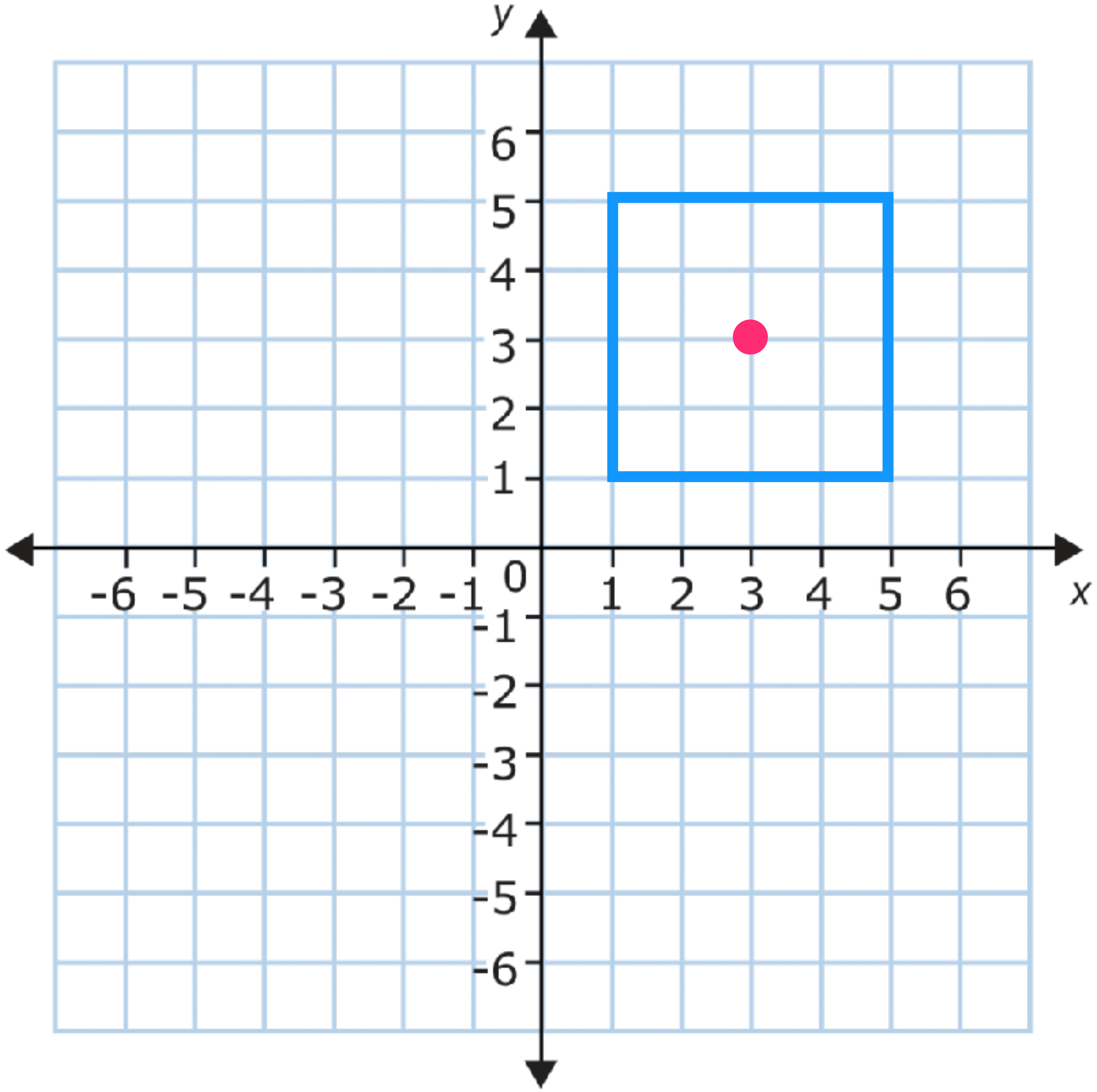


If the **distance between the point and the circle center** is less than its **radius**, then they are **colliding**.

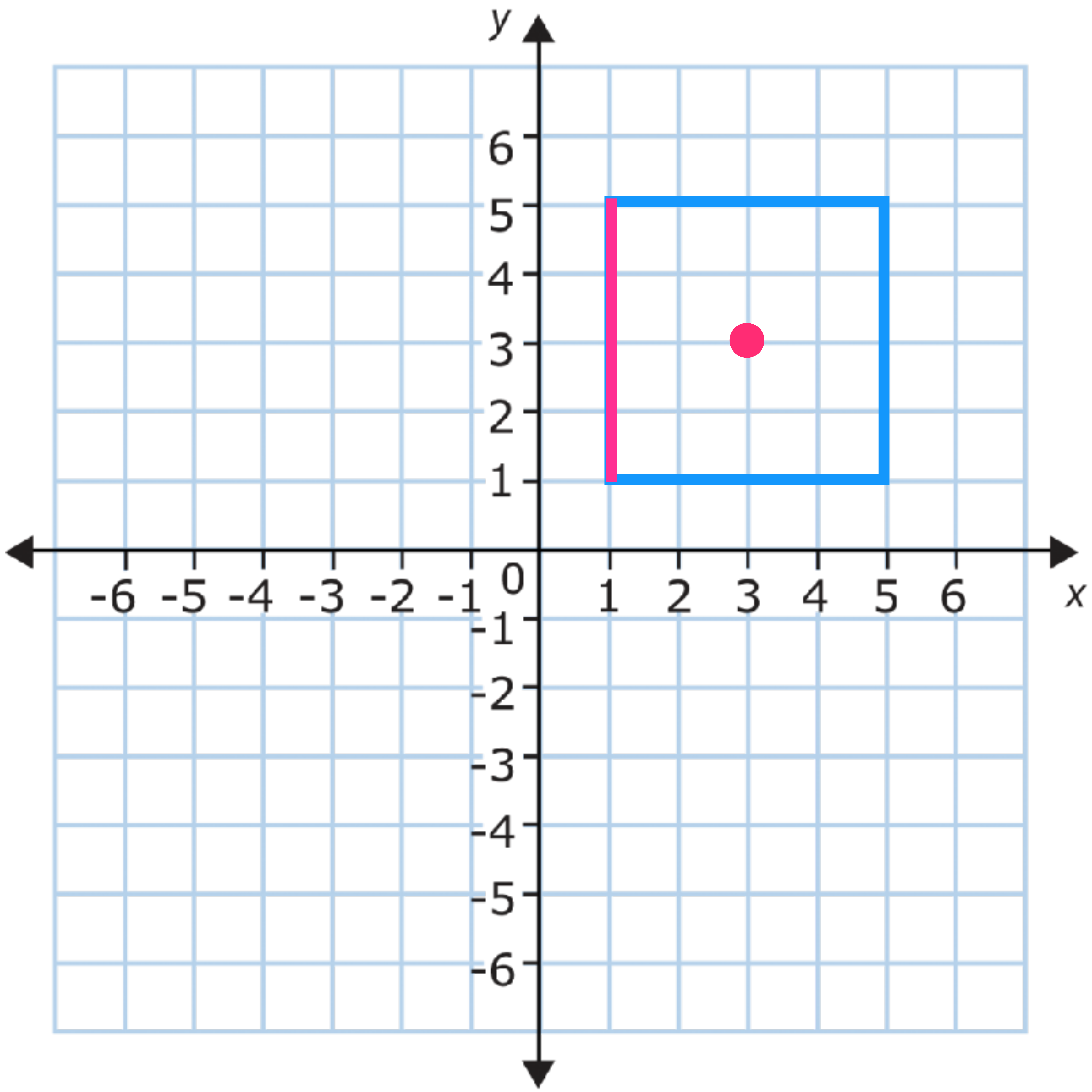
Box-box collision detection.




y
x
0
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
R1
R2

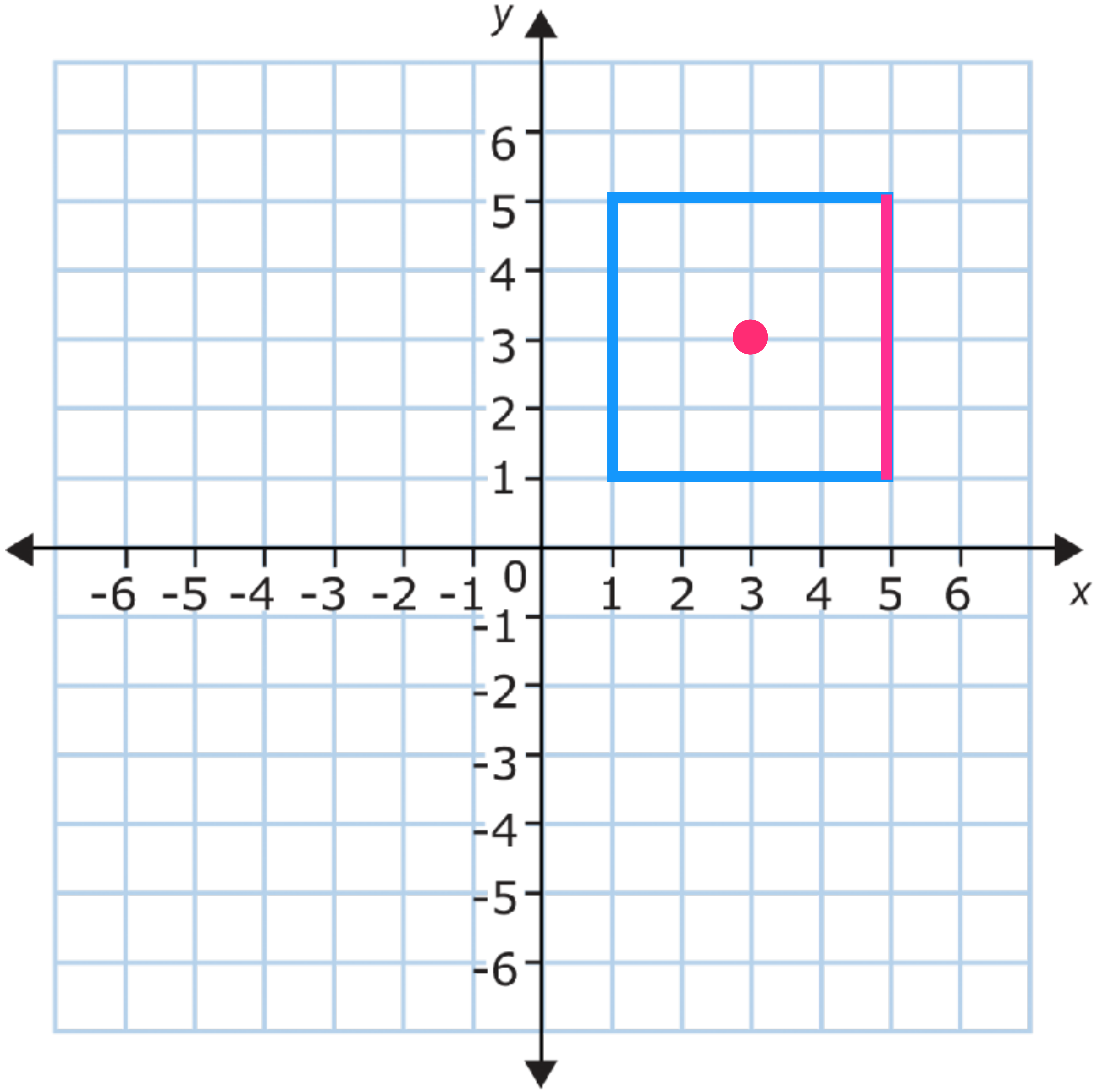


y
x
6
5
4
3
2
1
0
-1
-2
-3
-4
-5
-6
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6



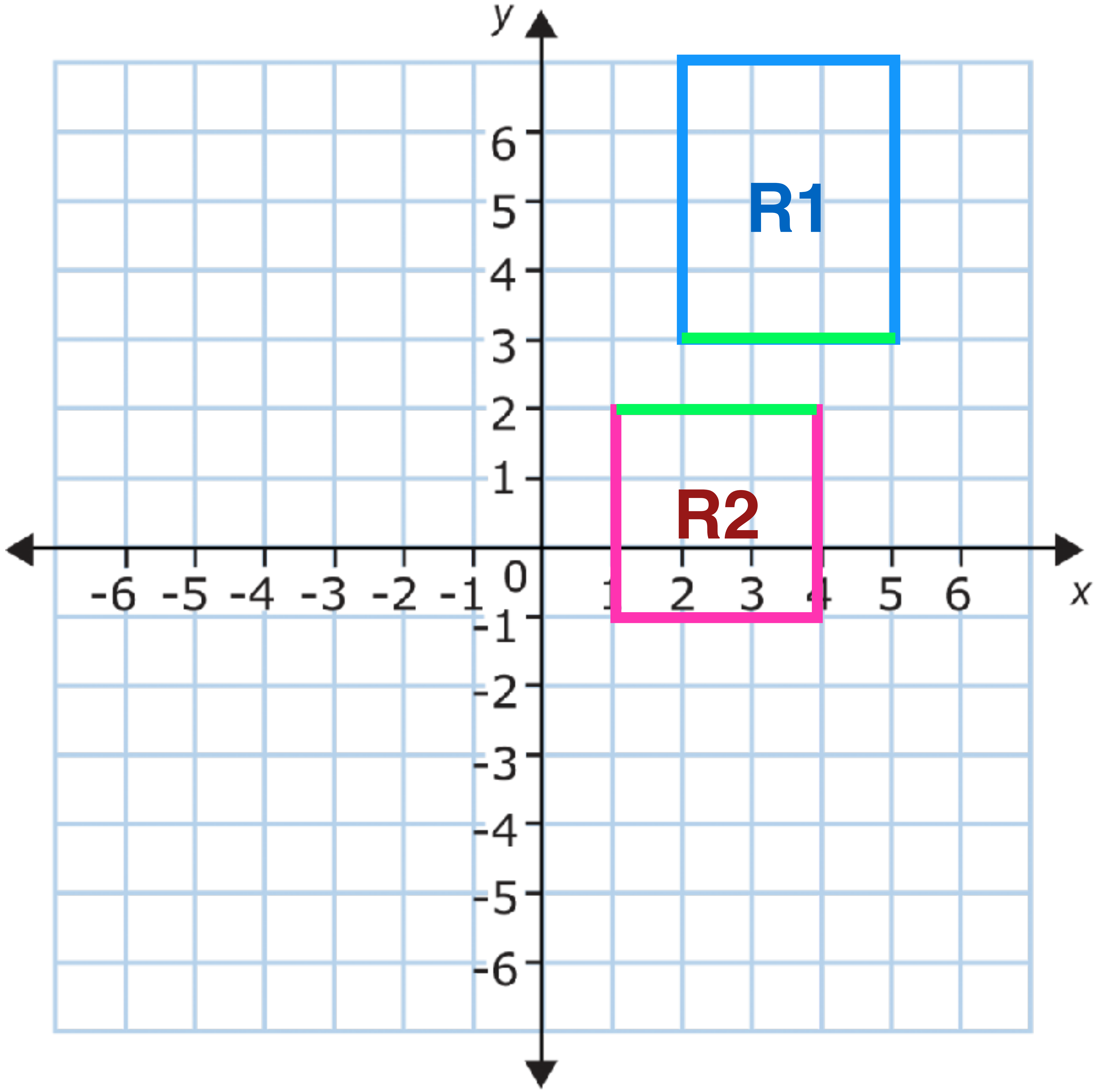
Left side

rectangle.x - rectangle.width / 2

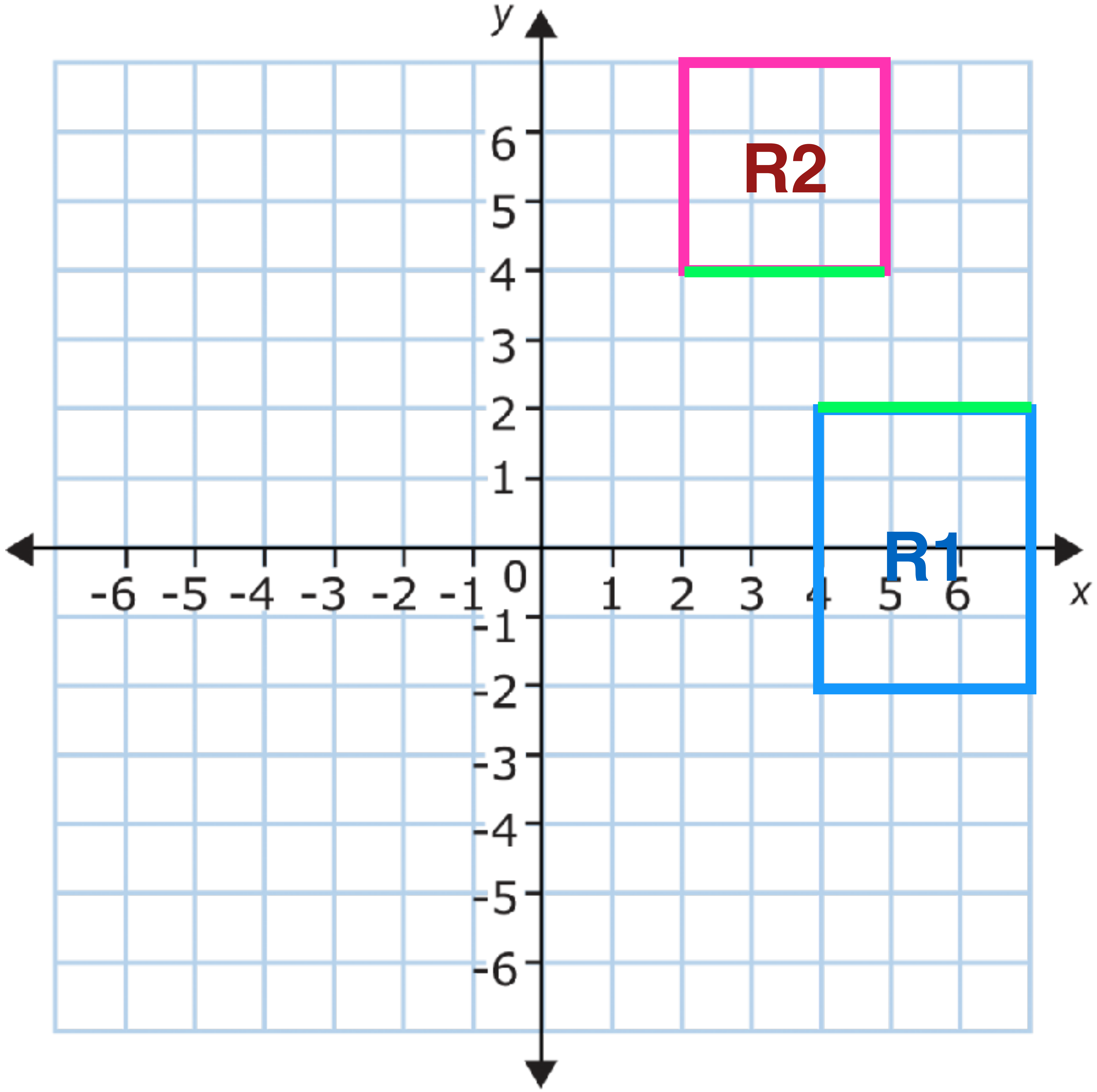


Right side

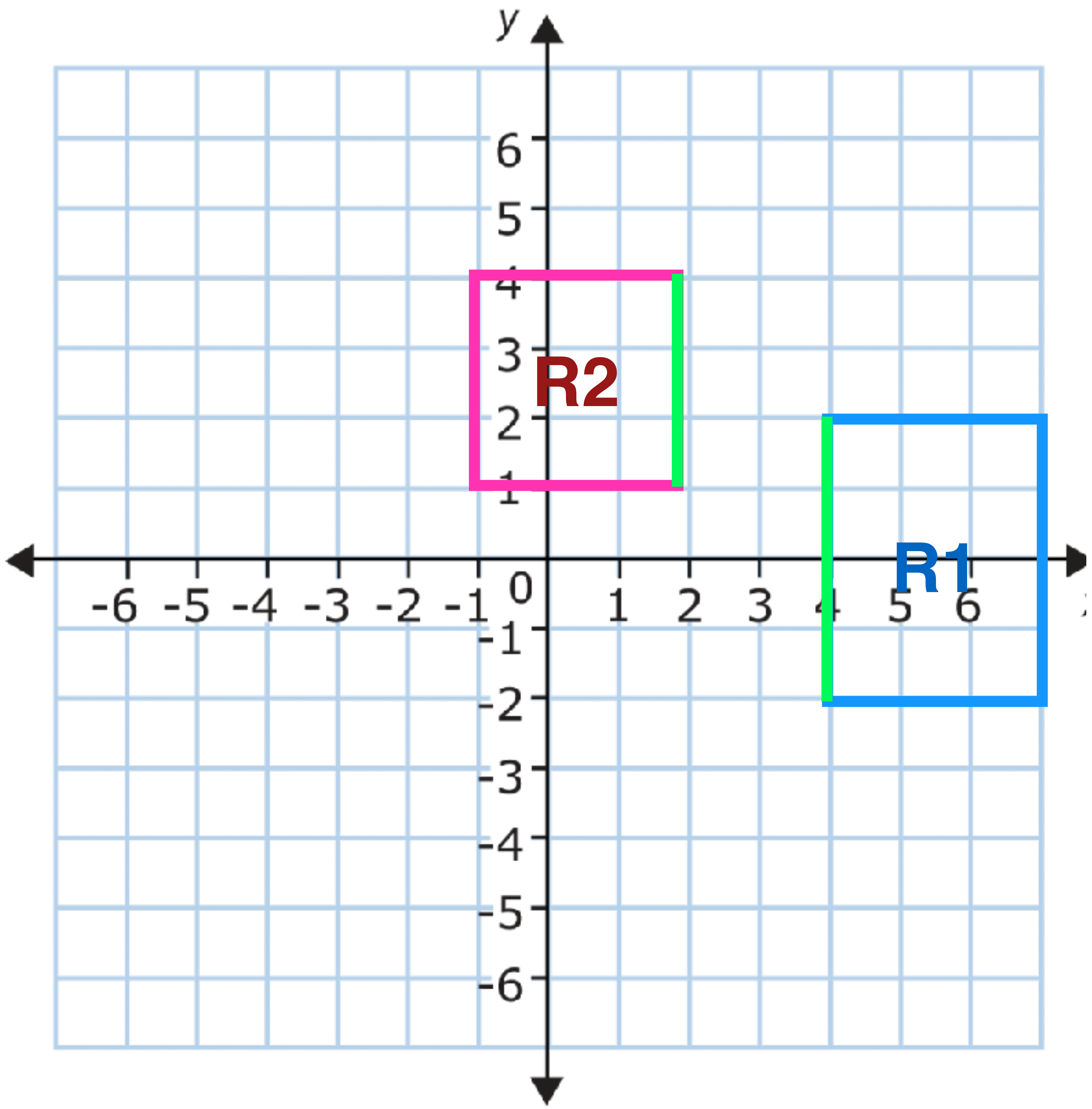
rectangle.x + rectangle.width / 2



a) R1's bottom is higher than R2's top



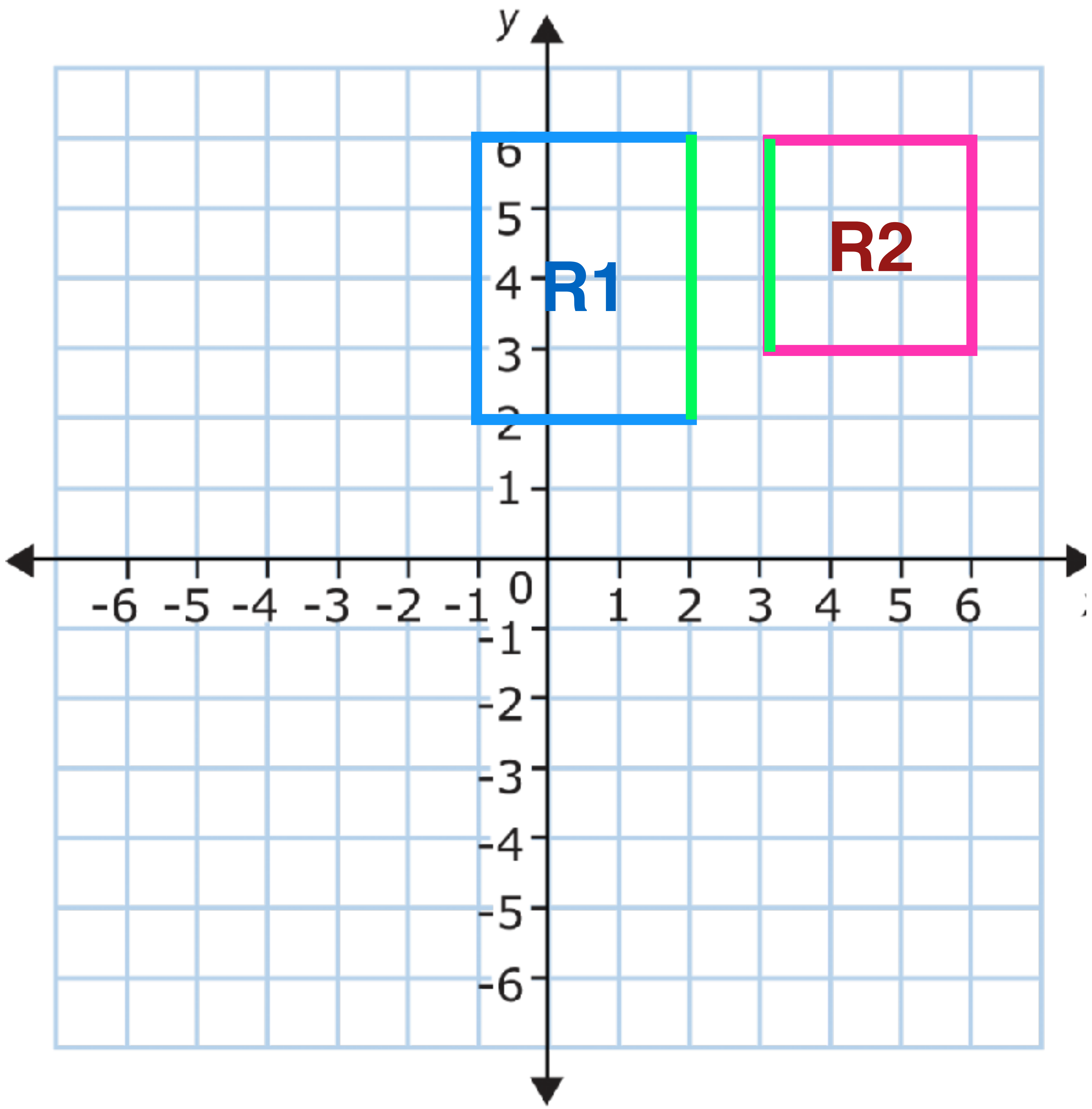
a) R1's bottom is higher than R2's top
b) R1's top is lower than R2's bottom



a) R1's bottom is higher than R2's top

b) R1's top is lower than R2's bottom

c) R1's left is larger than R2's right



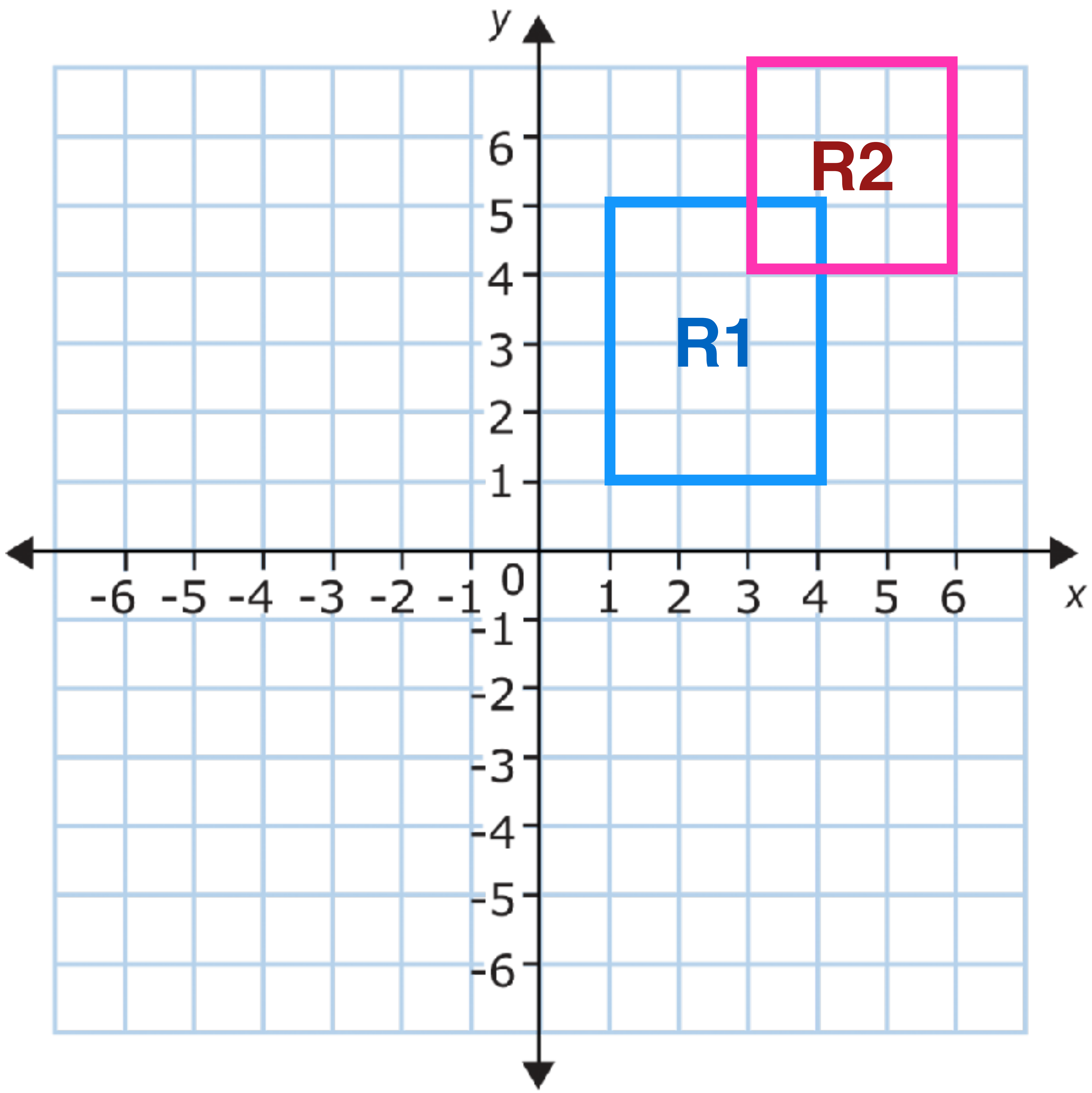
a) R1's bottom is higher than R2's top
b) R1's top is lower than R2's bottom
c) R1's left is larger than R2's right
d) R1's right is smaller than R2's left

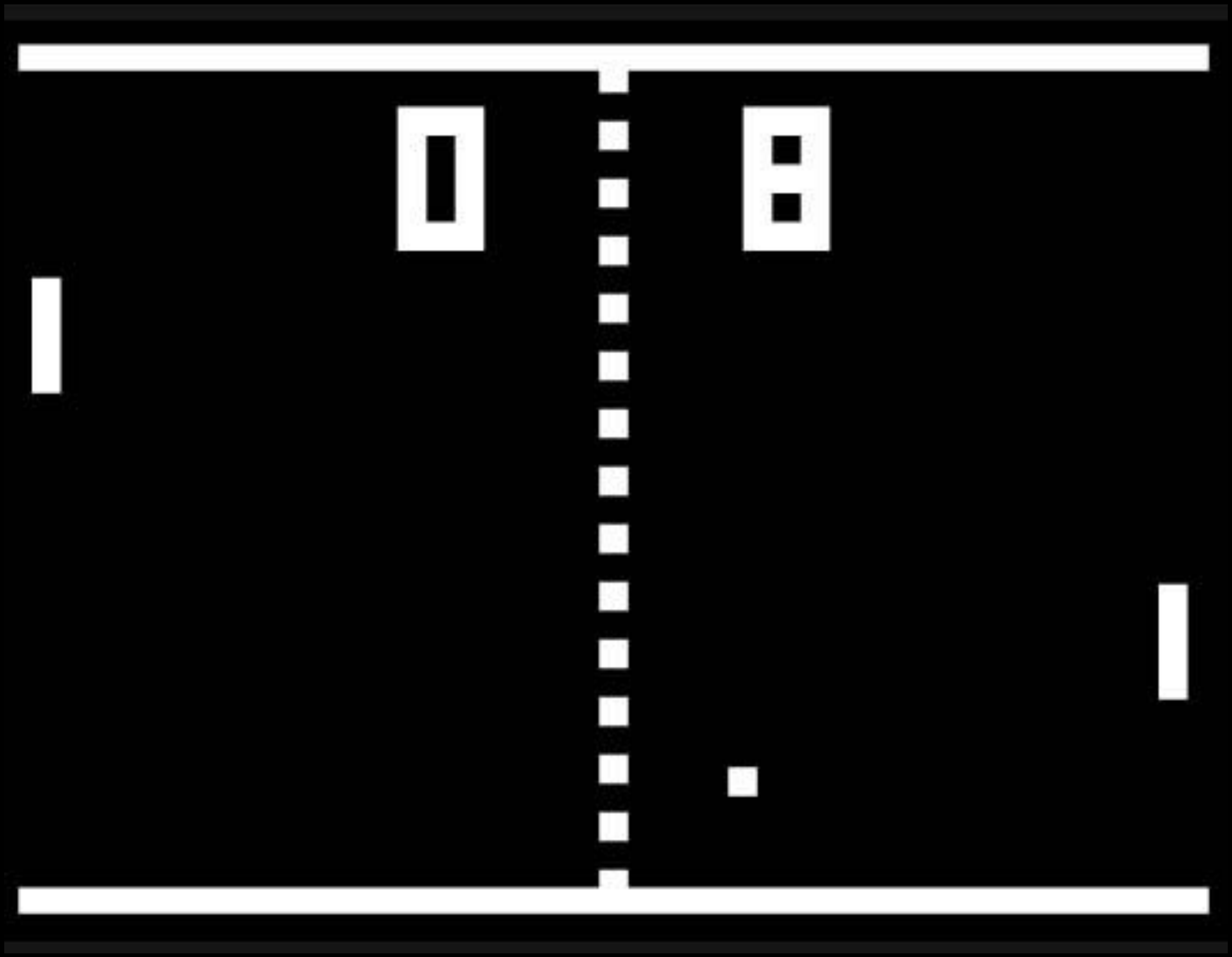
a) is R1's bottom higher than R2's top?
b) is R1's top lower than R2's bottom?
c) is R1's left larger than R2's right?
d) is R1's right smaller than R2's left

If ANY of the above are true, then the two rectangles are NOT intersecting!

OR

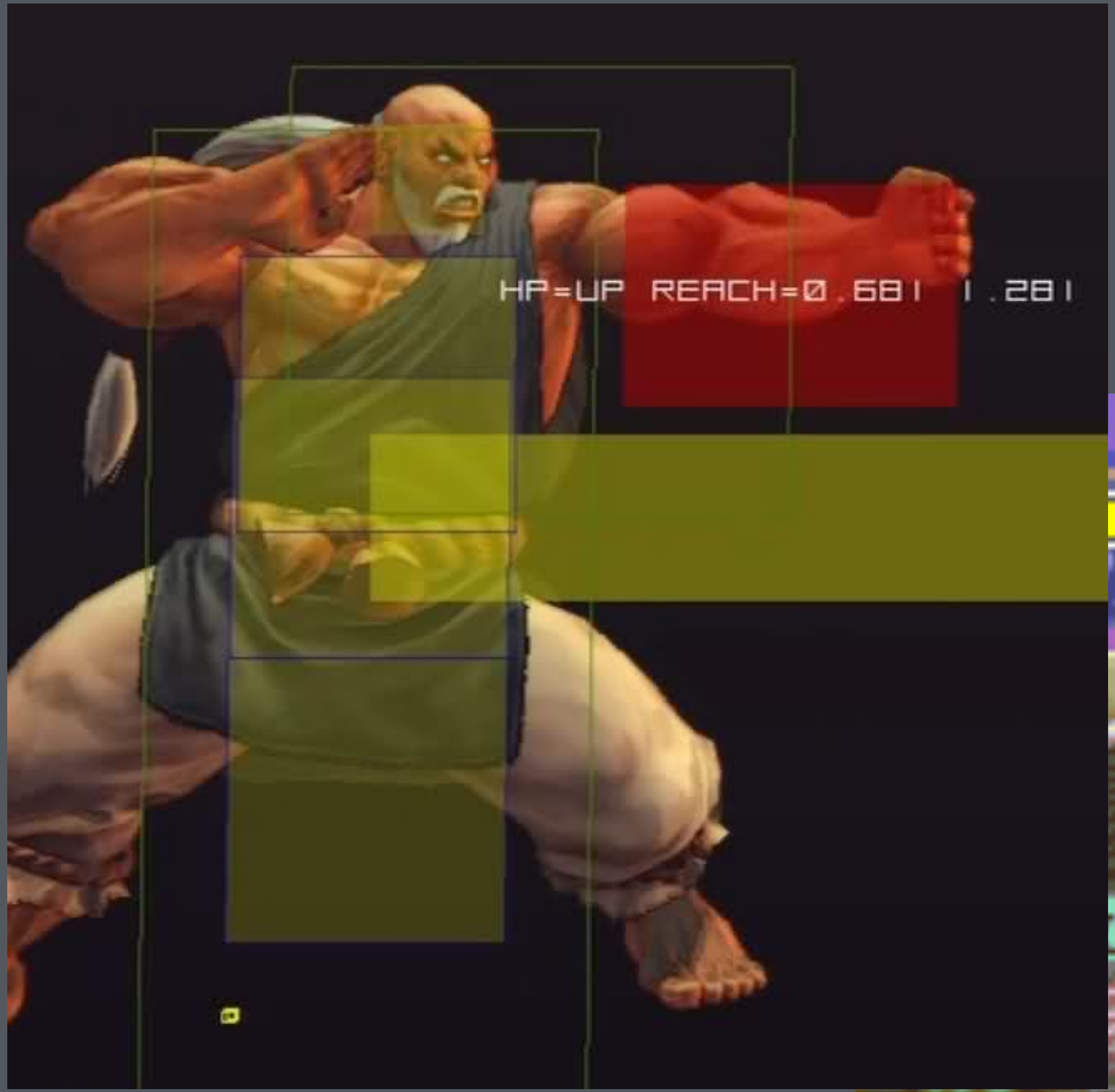
The rectangles are intersecting if NONE of the above are true.

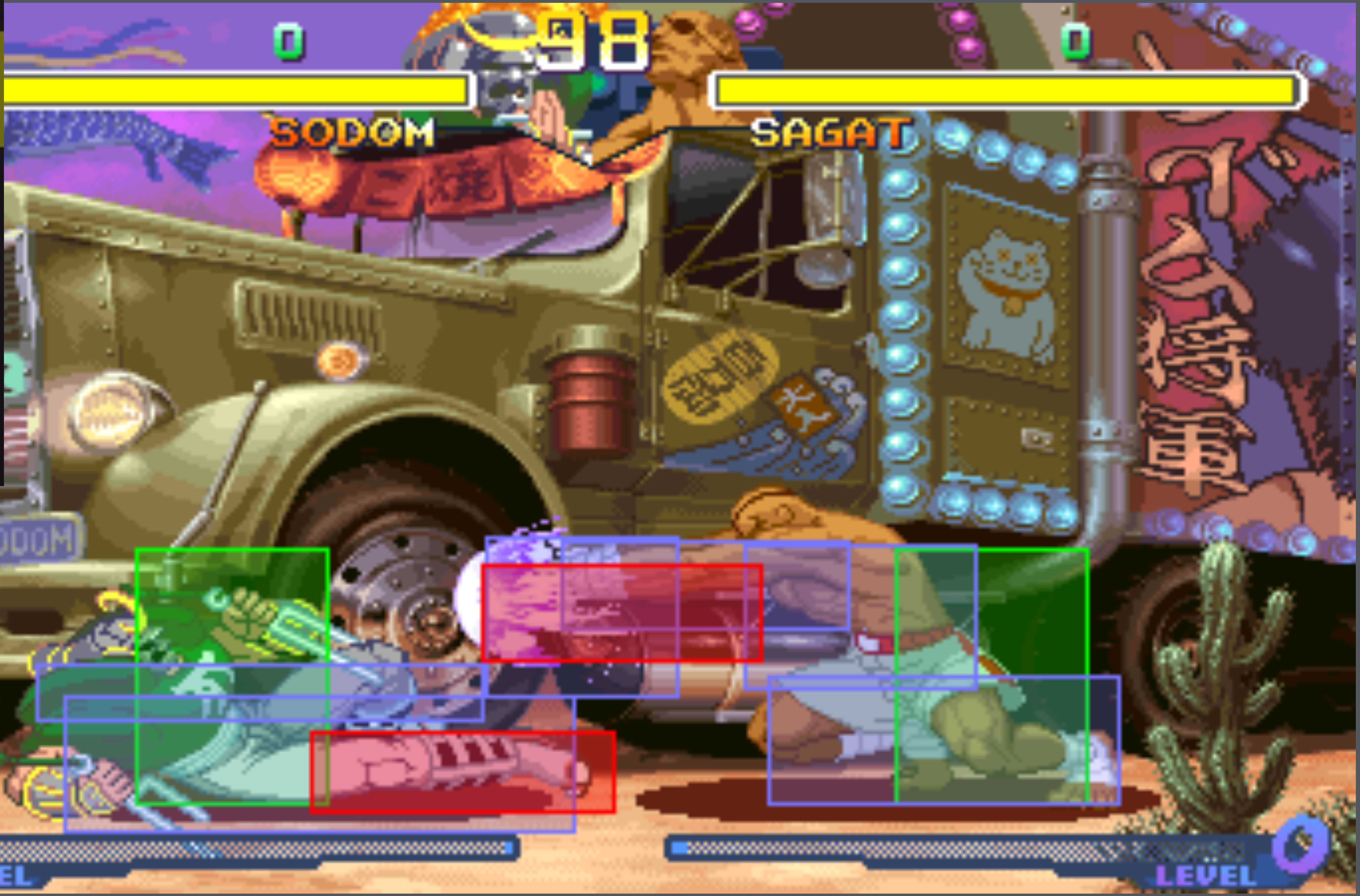




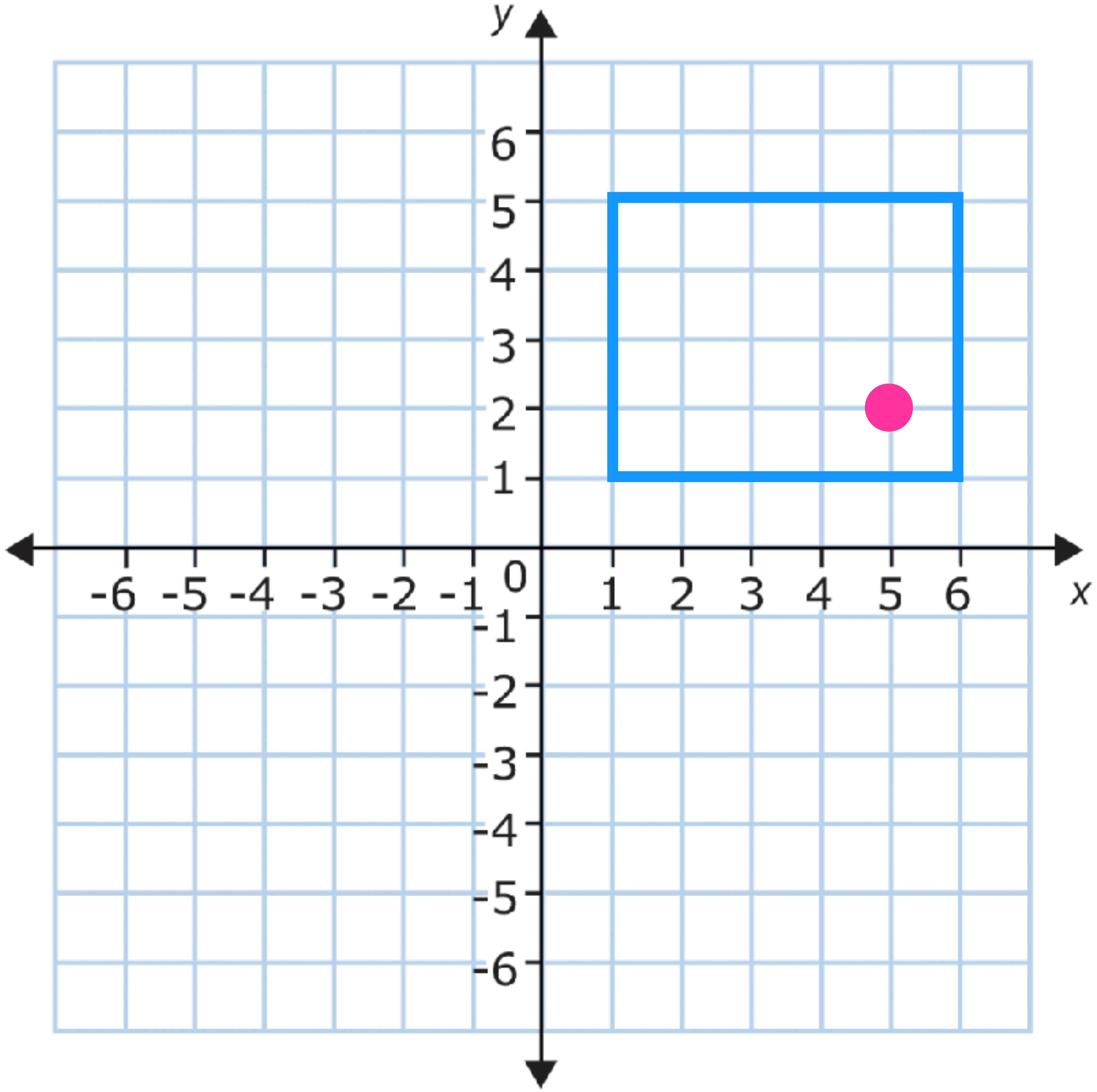
0
8

MARIO
004250
×05
WORLD
1-1
TIME
283
100


HP=UP REACH=0.681 1.281


0
98
0
SODOM
SAGAT
SODOM
LEVEL
LEVEL

Box-point collision detection.



collision is happening if:

- point x is larger than box left and smaller than box right

- point y is larger than box bottom and smaller than box top