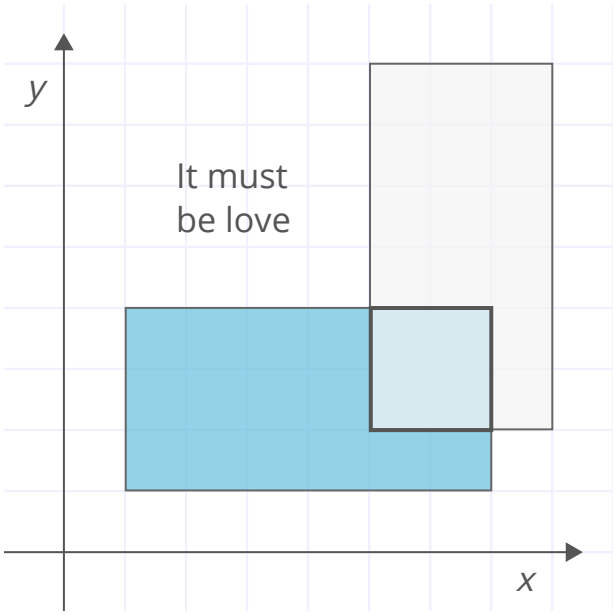
Interview Cake

A crack team of love scientists from OkEros (a hot new dating site) have devised a way to represent dating profiles as rectangles on a two-dimensional plane.

They need help writing an algorithm to find the intersection of two users' love rectangles. They suspect finding that intersection is the key to a matching algorithm so *powerful* it will cause an immediate acquisition by Google or Facebook or Obama or something.



Write a function to find the rectangular intersection of two given love rectangles.

As with the example above, love rectangles are always "straight" and never "diagonal." More rigorously: each side is parallel with either the x-axis or the y-axis.

They are defined as dictionaries↴

A **hash table** organizes data so you can quickly look up values for a given key.

	Average	Worst Case
space	$O(n)$	$O(n)$
insert	$O(1)$	$O(n)$
lookup	$O(1)$	$O(n)$
delete	$O(1)$	$O(n)$

Strengths:

- **Fast lookups**. Lookups take $O(1)$ time *on average*.
- **Flexible keys**. Most data types can be used for keys, as long as they're hashable (/concept/hashing).

Weaknesses:

- **Slow worst-case lookups**. Lookups take $O(n)$ time *in the worst case*.
- **Unordered**. Keys aren't stored in a special order. If you're looking for the smallest key, the largest key, or all the keys in a range, you'll need to look through every key to find it.
- **Single-directional lookups**. While you can look up the *value* for a given key in $O(1)$ time, looking up the *keys* for a given *value* requires looping through the whole dataset—$O(n)$ time.
- **Not cache-friendly**. Many hash table implementations use linked lists (/concept/linked-list), which don't put data next to each other in memory.

In Python 3.6

In Python 3.6, hash tables are called dictionaries.

Python 3.6

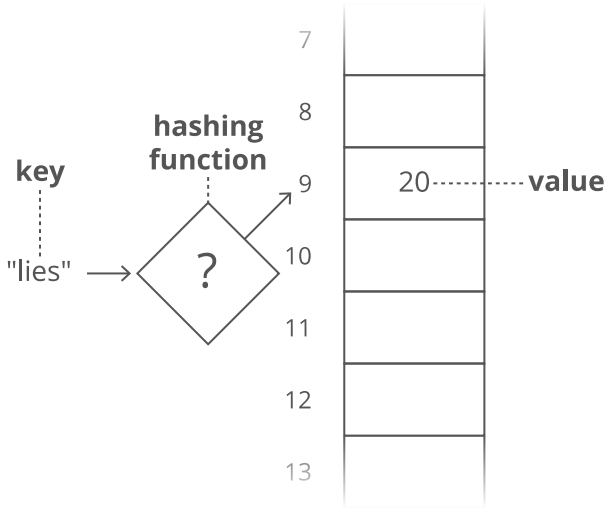
```
light_bulb_to_hours_of_light = {
    'incandescent': 1200,
    'compact fluorescent': 10000,
    'LED': 50000,
}
```

Hash maps are built on arrays

Arrays (/concept/array) are pretty similar to hash maps already. Arrays let you quickly look up the value for a given "key" . . . except the keys are called "indices," and we don't get to pick them—they're always sequential integers (0, 1, 2, 3, etc).

Think of a hash map as a "hack" on top of an array to let us use flexible keys instead of being stuck with sequential integer "indices."

All we need is a function to convert a key into an array index (an integer). That function is called a **hashing function (/concept/hashing)**.



To look up the value for a given key, we just run the key through our hashing function to get the index to go to in our underlying array to grab the value.

How does that hashing function work? There are a few different approaches, and they can get pretty complicated. But here's a simple proof of concept:

Grab the number value for each character and add those up.

The result is 429. But what if we only have *30* slots in our array? We'll use a common trick for forcing a number into a specific range: the modulus operator (%). (/concept/modulus) Modding our sum by 30 ensures we get a whole

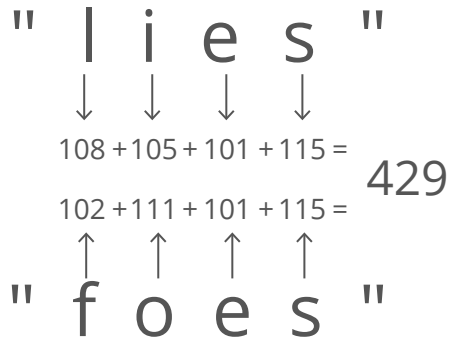
number that's less than 30 (and at least 0):

$$429 \% 30 = 9$$

> The hashing functions used in modern systems get pretty complicated—the one we used here is a simplified example.

Hash collisions

What if two keys hash to the same index in our array? In our example above, look at "lies" and "foes":

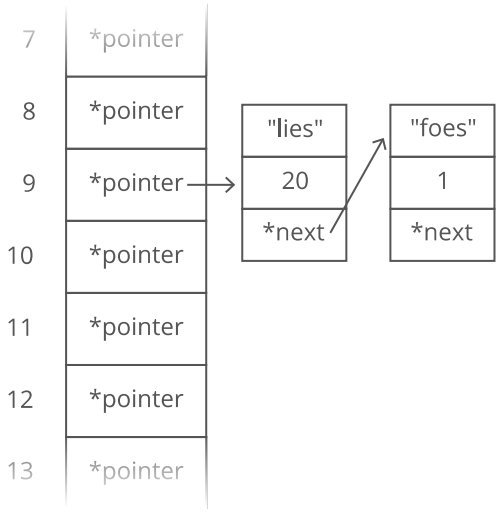


They both sum up to 429! So of course they'll have the same answer when we mod by 30:

$$429 \% 30 = 9$$

This is called a **hash collision**. There are a few different strategies for dealing with them.

Here's a common one: instead of storing the actual values in our array, let's have each array slot hold a *pointer* to a *linked list (/concept/linked-list)* holding the values for all the keys that hash to that index:



Notice that we included the *keys* as well as the values in each linked list node. Otherwise we wouldn't know which key was for which value!

> There are other ways to deal with hash collisions. This is just one of them.

When hash table operations cost $O(n)$ time

Hash collisions

If *all* our keys caused hash collisions, we'd be at risk of having to walk through all of our values for a single lookup (in the example above, we'd have one big linked list). This is unlikely, but it *could* happen. That's the worst case.

Dynamic array resizing

Suppose we keep adding more items to our hash map. As the number of keys and values in our hash map exceeds the number of indices in the underlying array, hash collisions become inevitable.

To mitigate this, we could expand our underlying array whenever things start to get crowded. That requires allocating a larger array and rehashing all of our existing keys to figure out their new position—$O(n)$ time.

Sets

A **set** is like a hash map except it only stores keys, without values.

Sets often come up when we're tracking groups of items—nodes we've visited in a graph, characters we've seen in a string, or colors used by neighboring nodes. Usually, we're interested in whether something is in a set or not.

Sets are usually implemented very similarly to hash maps—using hashing to index into an array—but they don't have to worry about storing values alongside keys. In Python, the set implementation is largely copied from the dictionary implementation (https://markmail.org/message/ktzomp4uwrmnzao6).

Python 3.6

```
light_bulbs = set()

light_bulbs.add('incandescent')
light_bulbs.add('compact fluorescent')
light_bulbs.add('LED')

'LED' in light_bulbs  # True
'halogen' in light_bulbs  # False
```

like this:

Python 3.6

```
my_rectangle = {

    # Coordinates of bottom-left corner
    'left_x'   : 1,
    'bottom_y' : 1,

    # Width and height
    'width'    : 6,
    'height'   : 3,

}
```

Your output rectangle should use this format as well.

Gotchas

What if there is *no* intersection? Does your function do something reasonable in that case?

What if one rectangle is entirely contained in the other? Does your function do something reasonable in that case?

What if the rectangles don't really intersect but share an edge? Does your function do something reasonable in that case?

Do some parts of your function seem very similar? Can they be refactored so you repeat yourself less?

Breakdown

Let's break this problem into subproblems. How can we divide this problem into smaller parts?

We could look at the two rectangles' "horizontal overlap" or "x overlap" separately from their "vertical overlap" or "y overlap."

Lets start with a helper function `find_x_overlap()`.

Need help finding the x overlap?

Since we're only working with the x dimension, we can treat the two rectangles' widths as ranges on a 1-dimensional number line.

What are the possible cases for how these ranges might overlap or not overlap? Draw out some examples!

There are four relevant cases:

1) The ranges partially overlap:

2) One range is completely contained in the other:

3) The ranges don't overlap:

4) The ranges "touch" at a single point:

Let's start with the first 2 cases. **How do we compute the overlapping range?**

One of our ranges starts "further to the right" than the other. We don't know ahead of time which one it is, but we can check the starting points of each range to see which one has the `highest_start_point`. **That `highest_start_point` is always the left-hand side of the overlap**, if there is one.

Not convinced? Draw some examples!

Similarly, **the right-hand side of our overlap is always the `lowest_end_point`**. That *may or may not* be the end point of the same input range that had the `highest_start_point`—compare cases (1) and (2).

This gives us our x overlap! So we can handle cases (1) and (2). **How do we know when there is *no* overlap?**

If `highest_start_point > lowest_end_point`, the two rectangles do not overlap.

But be careful—is it just *greater than* or is it *greater than or equal to*?

It depends how we want to handle case (4) above.

If we use *greater than*, we treat case (4) as an overlap. This means we could end up returning a rectangle with *zero width*, which ... may or may not be what we're looking for. You could make an argument either way.

Let's say a rectangle with zero width (or zero height) isn't a rectangle at all, so we should treat that case as "no intersection."

Can you finish `find_x_overlap()`?

Here's one way to do it:

Python 3.6

```
def find_x_overlap(x1, width1, x2, width2):

    # Find the highest ("rightmost") start point and
    # lowest ("leftmost") end point
    highest_start_point = max(x1, x2)
    lowest_end_point = min(x1 + width1, x2 + width2)

    # Return null overlap if there is no overlap
    if highest_start_point >= lowest_end_point:
        return (None, None)

    # Compute the overlap width
    overlap_width = lowest_end_point - highest_start_point

    return (highest_start_point, overlap_width)
```

How can we adapt this for the rectangles' `y`s and `height`s?

Can we just make one `find_range_overlap()` function that can handle x overlap and y overlap?

Yes! We simply use more general parameter names:

Python 3.6

```
def find_range_overlap(point1, length1, point2, length2):

    # Find the highest start point and lowest end point.
    # The highest ("rightmost" or "upmost") start point is
    # the start point of the overlap.
    # The lowest end point is the end point of the overlap.
    highest_start_point = max(point1, point2)
    lowest_end_point = min(point1 + length1, point2 + length2)

    # Return null overlap if there is no overlap
    if highest_start_point >= lowest_end_point:
        return (None, None)

    # Compute the overlap length
    overlap_length = lowest_end_point - highest_start_point

    return (highest_start_point, overlap_length)
```

We've solved our subproblem of finding the x and y overlaps! **Now we just need to put the results together.**

Solution

We divide the problem into two halves:

1. The intersection along the x-axis
2. The intersection along the y-axis

Both problems are basically the same as finding the intersection of two "ranges" on a 1-dimensional number line.

So we write a helper function `find_range_overlap()` that can be used to find both the x overlap and the y overlap, and we use it to build the rectangular overlap:

Python 3.6

```
def find_range_overlap(point1, length1, point2, length2):
    # Find the highest start point and lowest end point.
    # The highest ("rightmost" or "upmost") start point is
    # the start point of the overlap.
    # The lowest end point is the end point of the overlap.
    highest_start_point = max(point1, point2)
    lowest_end_point = min(point1 + length1, point2 + length2)

    # Return null overlap if there is no overlap
    if highest_start_point >= lowest_end_point:
        return (None, None)

    # Compute the overlap length
    overlap_length = lowest_end_point - highest_start_point

    return (highest_start_point, overlap_length)


def find_rectangular_overlap(rect1, rect2):
    # Get the x and y overlap points and lengths
    x_overlap_point, overlap_width  = find_range_overlap(rect1['left_x'],
                                                         rect1['width'],
                                                         rect2['left_x'],
                                                         rect2['width'])
    y_overlap_point, overlap_height = find_range_overlap(rect1['bottom_y'],
                                                         rect1['height'],
                                                         rect2['bottom_y'],
                                                         rect2['height'])

    # Return null rectangle if there is no overlap
    if not overlap_width or not overlap_height:
        return {
            'left_x'   : None,
            'bottom_y' : None,
            'width'    : None,
            'height'   : None,
        }

    return {
        'left_x'   : x_overlap_point,
```

```
        'bottom_y' : y_overlap_point,
        'width'    : overlap_width,
        'height'   : overlap_height,
    }
```

Complexity

$O(1)$ time and $O(1)$ space.

Bonus

What if we had a list of rectangles and wanted to find *all* the rectangular overlaps between all possible pairs of two rectangles within the list? Note that we'd be returning *a list of rectangles*.

What if we had a list of rectangles and wanted to find the overlap between *all* of them, if there was one? Note that we'd be returning *a single rectangle*.

What We Learned

This is an interesting one because the hard part isn't the time or space optimization—it's getting something that *works* and is *readable*.

For problems like this, I often see candidates who can describe the strategy at a high level but trip over themselves when they get into the details.

Don't let it happen to you. To keep your thoughts clear and avoid bugs, take time to:

1. Think up and draw out all the possible cases. Like we did with the ways ranges can overlap.
2. Use very specific and descriptive variable names.

Ready for more?