Interview Cake

You've built an inflight entertainment system with on-demand movie streaming.

Users on longer flights like to start a second movie right when their first one ends, but they complain that the plane usually lands before they can see the ending. **So you're building a feature for choosing two movies whose total runtimes will equal the exact flight length.**

Write a function that takes an integer `flight_length` (in minutes) and a list of integers `movie_lengths` (in minutes) and returns a boolean indicating whether there are two numbers in `movie_lengths` whose sum equals `flight_length`.

When building your function:

- Assume your users will watch *exactly* two movies
- Don't make your users watch the same movie twice
- Optimize for runtime over memory

Gotchas

We can do this in $O(n)$ time, where n is the length of `movie_lengths`.

Remember: your users shouldn't watch the same movie twice. **Are you sure your function won’t give a false positive if the list has one element that is half `flight_length`**?

Breakdown

How would we solve this by hand? We know our two movie lengths need to sum to `flight_length`. So for a given `first_movie_length`, we need a `second_movie_length` that equals `flight_length - first_movie_length`.

To do this by hand we might go through `movie_lengths` from beginning to end, treating each item as `first_movie_length`, and for each of those check if there's a `second_movie_length` equal to `flight_length - first_movie_length`.

How would we implement this in code? We could nest two loops (the outer choosing `first_movie_length`, the inner choosing `second_movie_length`). That'd give us a runtime of $O(n^2)$. We can do better.

To bring our runtime down we'll probably need to replace that inner loop (the one that looks for a matching `second_movie_length`) with something faster.

We could sort the `movie_lengths` first—then we could use binary search↴

A binary search algorithm finds an item in a *sorted* list in $O(lg(n))$ time.

A brute force search would walk through the whole list, taking $O(n)$ time in the worst case.

Let's say we have a sorted list of numbers. To find a number with a binary search, we:

1. **Start with the middle number: is it bigger or smaller than our target number?** Since the list is sorted, this tells us if the target would be in the *left* half or the *right* half of our list.
2. **We've effectively divided the problem in half**. We can "rule out" the whole half of the list that we know doesn't contain the target number.
3. **Repeat the same approach (of starting in the middle) on the new half-size problem**. Then do it again and again, until we either find the number or "rule out" the whole set.

We can do this recursively, or iteratively. Here's an iterative version:

Python 3.6

```
def binary_search(target, nums):
    """See if target appears in nums"""

    # We think of floor_index and ceiling_index as "walls" around
    # the possible positions of our target, so by -1 below we mean
    # to start our wall "to the left" of the 0th index
    # (we *don't* mean "the last index")
    floor_index = -1
    ceiling_index = len(nums)

    # If there isn't at least 1 index between floor and ceiling,
    # we've run out of guesses and the number must not be present
    while floor_index + 1 < ceiling_index:

        # Find the index ~halfway between the floor and ceiling
        # We use integer division, so we'll never get a "half index"
        distance = ceiling_index - floor_index
        half_distance = distance // 2
        guess_index = floor_index + half_distance

        guess_value = nums[guess_index]

        if guess_value == target:
            return True

        if guess_value > target:
            # Target is to the left, so move ceiling to the left
            ceiling_index = guess_index
        else:
            # Target is to the right, so move floor to the right
            floor_index = guess_index

    return False
```

How did we know the time cost of binary search was $O(lg(n))$? The only non-constant part of our time cost is the number of times our while loop runs. Each step of our while loop cuts the range (dictated by `floor_index` and `ceiling_index`) in half, until our range has just one element left.

So the question is, "how many times must we divide our original list size (n) in half until we get down to 1?"

$$n * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * ... = 1$$

How many $\frac{1}{2}$'s are there? We don't know yet, but we can call that number x:

$$n * (\frac{1}{2})^x = 1$$

Now we solve for x:

$$n * \frac{1^x}{2^x} = 1$$

$$n * \frac{1}{2^x} = 1$$

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

Now to get the x out of the exponent. How do we do that? Logarithms.

Recall that $\log_{10} 100$ means, "what power must we raise 10 to, to get 100"? The answer is 2.

So in this case, if we take the $\log_2$ of both sides...

$$\log_2 n = \log_2 2^x$$

The right hand side asks, "what power must we raise 2 to, to get 2^x?" Well, that's just x.

$$\log_2 n = x$$

So there it is. The number of times we must divide n in half to get down to 1 is $log_2 n$. So our total time cost is $O(lg(n))$

Careful: we can only use binary search if the input list is *already sorted*.

to find `second_movie_length` in $O(\lg n)$ time instead of $O(n)$ time. But sorting would cost $O(nlg(n))$, and we can do even better than that.

Could we check for the existence of our `second_movie_length` in constant time?

What data structure gives us convenient constant-time lookups?

A set!

So we could throw all of our `movie_lengths` into a set first, in $O(n)$ time. *Then* we could loop through our possible `first_movie_lengths` and replace our inner loop with a simple check in our set. This'll give us $O(n)$ runtime overall!

Of course, we need to add some logic to make sure we're not showing users the same movie twice...

But first, we can tighten this up a bit. Instead of two sequential loops, can we do it all in one loop? (Done carefully, this will give us protection from showing the same movie twice as well.)

Solution

We make one pass through `movie_lengths`, treating each item as the `first_movie_length`. At each iteration, we:

1. See if there's a `matching_second_movie_length` we've seen already (stored in our `movie_lengths_seen` set) that is equal to `flight_length - first_movie_length`. If there is, we short-circuit and return `True`.
2. Keep our `movie_lengths_seen` set up to date by throwing in the current `first_movie_length`.

Python 3.6

```
def can_two_movies_fill_flight(movie_lengths, flight_length):
    # Movie lengths we've seen so far
    movie_lengths_seen = set()

    for first_movie_length in movie_lengths:
        matching_second_movie_length = flight_length - first_movie_length
        if matching_second_movie_length in movie_lengths_seen:
            return True
        movie_lengths_seen.add(first_movie_length)

    # We never found a match, so return False
    return False
```

We know users won't watch the same movie twice because we check `movie_lengths_seen` for `matching_second_movie_length` *before* we've put `first_movie_length` in it!

Complexity

$O(n)$ time, and $O(n)$ space. Note while optimizing runtime we added a bit of space cost.

Bonus

1. What if we wanted the movie lengths to sum to something *close* to the flight length (say, within 20 minutes)?
2. What if we wanted to fill the flight length as nicely as possible with *any* number of movies (not just 2)?
3. What if we knew that `movie_lengths` was *sorted*? Could we save some space and/or time?

What We Learned

The trick was to use a set to access our movies *by length*, in $O(1)$ time.

Using hash-based data structures, like dictionaries or sets, is *so common* in coding challenge solutions, it should always be your *first* thought. Always ask yourself, right from the start: "Can I save time by using a dictionary?"