# 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 to find `second_movie_length` in $O(\lg n)$ time instead of $O(n)$ time. But sorting would cost $O(n lg(n))$, 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
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
```

Python 3.6

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?"

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.