

You've built an in-flight 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 `flightLength` (in minutes) and an array of integers `movieLengths` (in minutes) and returns a boolean indicating whether there are two numbers in `movieLengths` whose sum equals `flightLength`.

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 `movieLengths`.

Remember: your users shouldn't watch the same movie twice. **Are you sure your method won't give a false positive if the array has one element that is half flightLength?**

Breakdown

How would we solve this by hand? We know our two movie lengths need to sum to flightLength. So for a given firstMovieLength, we need a secondMovieLength that equals flightLength - firstMovieLength.

To do this by hand we might go through movieLengths from beginning to end, treating each item as firstMovieLength, and for each of those check if there's a secondMovieLength equal to flightLength - firstMovieLength.

How would we implement this in code? We could nest two loops (the outer choosing firstMovieLength, the inner choosing secondMovieLength). 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 secondMovieLength) with something faster.

We could sort the movieLengths first—then we could use binary search to find secondMovieLength 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 secondMovieLength in constant time?

What data structure gives us convenient constant-time lookups?

A hash set!

So we could throw all of our `movieLengths` into a hash set first, in $O(n)$ time. Then we could loop through our possible `firstMovieLengths` and replace our inner loop with a simple check in our hash 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 `movieLengths`, treating each item as the `firstMovieLength`. At each iteration, we:

1. See if there's a `matchingSecondMovieLength` we've seen already (stored in our `movieLengthsSeen` hash set) that is equal to `flightLength - firstMovieLength`. If there is, we short-circuit and return `true`.
2. Keep our `movieLengthsSeen` hash set up to date by throwing in the current `firstMovieLength`.

```
public boolean canTwoMoviesFillFlight(int[] movieLengths, int flightLength) {

    // movie lengths we've seen so far
    Set<Integer> movieLengthsSeen = new HashSet<Integer>();

    for (int firstMovieLength : movieLengths) {

        int matchingSecondMovieLength = flightLength - firstMovieLength;
        if (movieLengthsSeen.contains(matchingSecondMovieLength)) {
            return true;
        }

        movieLengthsSeen.add(firstMovieLength);
    }

    // we never found a match, so return false
    return false;
}
```

Java ▼

We know users won't watch the same movie twice because we check `movieLengthsSeen` for `matchingSecondMovieLength` *before* we've put `firstMovieLength` 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 `movieLengths` was *sorted*? Could we save some space and/or time?

What We Learned

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

Using hash-based data structures, like hash maps or hash 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 hash map?"

Want more coding interview help?

Check out **[interviewcake.com](https://www.interviewcake.com)** for more advice, guides, and practice questions.