**🍰 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 `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 function 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↴

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

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

Let's say we have a sorted array 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 array is sorted, this tells us if the target would be in the *left* half or the *right* half of our array.
2. **We've effectively divided the problem in half**. We can "rule out" the whole half of the array 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:

```javascript
function binarySearch(target, nums) {

    // see if target appears in nums


    // we think of floorIndex and ceilingIndex 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")
    var floorIndex = -1;
    var ceilingIndex = nums.length;


    // 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 (floorIndex + 1 < ceilingIndex) {

        // find the index ~halfway between the floor and ceiling
        // we have to round down, to avoid getting a "half index"
        var distance = ceilingIndex - floorIndex;
        var halfDistance = Math.floor(distance / 2);
        var guessIndex = floorIndex + halfDistance;


        var guessValue = nums[guessIndex];


        if (guessValue === target) {
            return true;
        }


        if (guessValue > target) {

            // target is to the left, so move ceiling to the left
            ceilingIndex = guessIndex;

        } else {

            // target is to the right, so move floor to the right
            floorIndex = guessIndex;

        }
    }


    return false;
}
```

JavaScript ▾

**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 `floorIndex` and `ceilingIndex`) in half, until our range has just one element left.

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

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

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

$$n * \left(\tfrac{1}{2}\right)^x = 1$$

Now we solve for $x$:

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

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

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

$$n = 2^x$$

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 array is *already sorted*.**

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 set!

So we could throw all of our `movieLengths` into a 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 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` set) that is equal to `flightLength - firstMovieLength`. If there is, we short-circuit and return `true`.
2. Keep our `movieLengthsSeen` set up to date by throwing in the current `firstMovieLength`.

<div style="float:right">JavaScript ▾</div>

```javascript
function canTwoMoviesFillFlight(movieLengths, flightLength) {

  // Movie lengths we've seen so far
  const movieLengthsSeen = new Set();

  for (let i = 0; i < movieLengths.length; i++) {
    const firstMovieLength = movieLengths[i];

    const matchingSecondMovieLength = flightLength - firstMovieLength;
    if (movieLengthsSeen.has(matchingSecondMovieLength)) {
      return true;
    }

    movieLengthsSeen.add(firstMovieLength);
  }

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

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 set to access our movies *by length*, in $O(1)$ time.

**Using hash-based data structures, like objects 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 an object?"

# Ready for more?

## Check out our full course ➡

---

Want more coding interview help?

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