**In-Place Array Operations Introduction**

In programming interviews, the interviewer often expects you to minimise the time and space complexity of your implementation. In-place Array operations help to reduce space complexity, and so are a class of techniques that pretty much everybody encounters regularly in interviews.

So, what *are* in-place array operations?

The best way of answering this question is to look at an example.

Given an Array of integers, return an Array where every element at an even-indexed position is squared.

**Input:** array = [9, -2, -9, 11, 56, -12, -3]

**Output:** [81, -2, 81, 11, 3136, -12, 9]

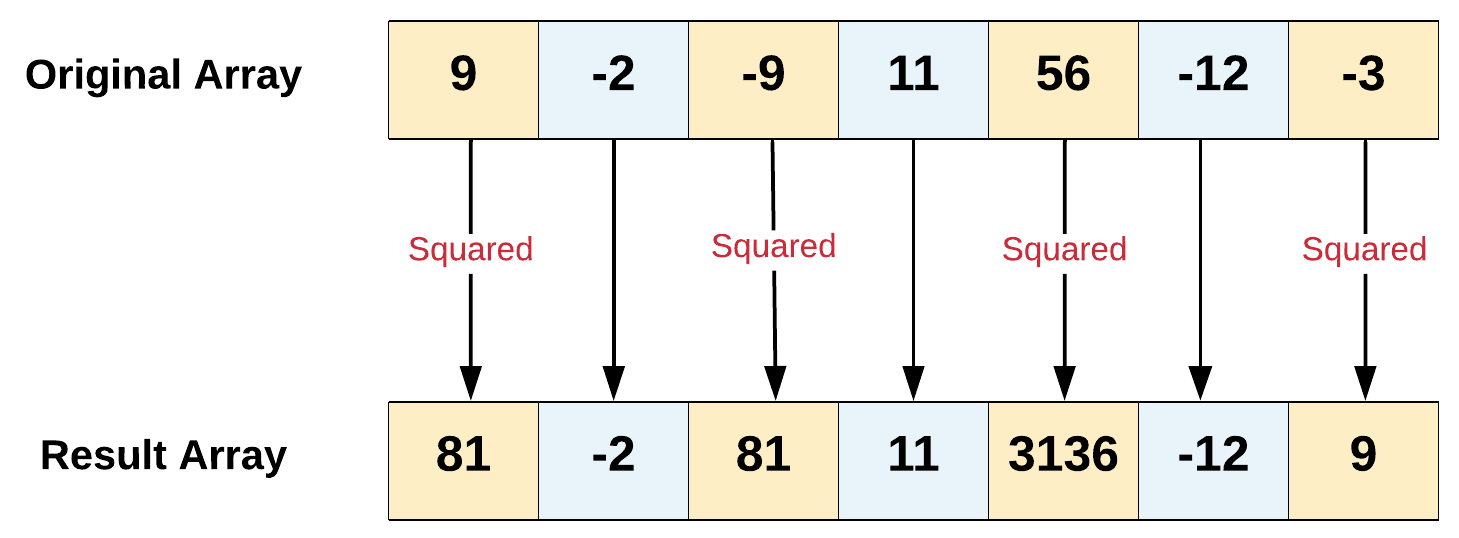
**Explanation:** The numbers at even indexes (0, 2, 4, 6) have been squared,

whereas the numbers at odd indexes (1, 3, 5) have been left the same.

This problem is hopefully very straightforward. Have a quick think about how you would implement it as an algorithm though, possibly jotting down some code on a piece of paper.

Anyway, there are two ways we could approach it. The first is to create a new Array, of the same size as the original. Then, we should copy the odd-indexed elements and square the even-indexed elements, writing them into the new Array.

|  |
| --- |
| // Check for edge cases.  if (array == null) {  return null;  }  // Create a resultant Array which would hold the result.  int result[] = new int[length];  // Iterate through the original Array.  for(int i = 0; i < length; i++) {  // Get the element from slot i of the input Array.  int element = array[i];  // If the index is an even number, then we need to square element.  if (i % 2 == 0) {  element \*= element;  }  // Write element into the result Array.  result[i] = element;  }  // Return the result Array.  return result;  } |



The above approach, although correct, is an *inefficient* way of solving the problem. This is because it uses O(\text{length})*O*(length) extra space.

Instead, we could iterate over the original input Array itself, overwriting every even-indexed element with its own square. This way, we won't need that extra space. It is this technique of working directly in the input Array, and *not* creating a new Array, that we call **in-place**. In-place Array operations are a big deal for programming interviews, where there is a big focus on minimising both time and space complexity of algorithms.

Here's the in-place implementation for our squareEven(...) function.

|  |
| --- |
| public int[] squareEven(int[] array, int length) {  // Check for edge cases.  if (array == null) {  return array;  }  // Iterate through the original array.  for(int i = 0; i < length; i++) {  // If the index is an even number, then we need to square the element  // and replace the original value in the Array.  if (i % 2 == 0) {  array[i] \*= array[i];  }  // Notice how we don't need to do \*anything\* for the odd indexes? :-)  }  // We just return the original array. Some problems on leetcode require you  // to return it, and other's dont.  return array;  } |

Here's an animation showing the algorithm!

// xem folder

An important difference for in-place vs not in-place is that in-place *modifies the input Array*. This means that other functions *can no longer access the original data*, because it has been overwritten. We'll talk more about this in a bit.

We now have a couple of straightforward in-place problems for you to try. Remember, you aren't allowed to create any new Arrays (or any other data structures). If the return type of the question is an Array, then simply return the input Array once you've modified it.