### **Finding Duplicate Elements in an Array**

12:15PM | [Geekery](http://www.google.com/url?q=http%3A%2F%2Faperiodic.net%2Fphil%2Farchives%2FGeekery&sa=D&sntz=1&usg=AFQjCNG2rAeCkkox24O9XtFcMZE2JDrjPw) | [#](http://www.google.com/url?q=http%3A%2F%2Faperiodic.net%2Fphil%2Farchives%2FGeekery%2Ffind-duplicate-elements.html&sa=D&sntz=1&usg=AFQjCNGkNAJjMQ9Vw9NgyOoZcTVfe1JkXA)

I came across an interesting programming puzzle today, and I'd like to share a couple of variants on it.

To start with, let's say we have an array *A* that contains some numbers. For simplicity, let's say that the array is 1-based. How can we efficiently find out if any of those numbers are duplicates?

The easiest approach is to make a hash table to keep track of the numbers we've seen so far. Something like this:

Given an array A of length n,

let h ← (new hash table)

for 1 <= i <= n:

if A[i] is present in h: return A[i]

set h[A[i]] ← True

return <no duplicates>

Now, suppose that the array is of length *n* and only contains positive integers less than *n*. We can be sure (by the [pigeonhole principle](http://www.google.com/url?q=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FPigeonhole_principle&sa=D&sntz=1&usg=AFQjCNHhd38e14YNKy0OKZQnapnlbK6gXg)) that there is at least one duplicate. The bounds on the values mean that we can be a little more efficient and store the already-seen numbers in an array rather than a hash table:

let s ← array [1..n]

initialize s to all zeroes

for 1 <= i <= n:

if s[A[i]] > 0: return A[i]

set s[A[i] ← 1

Now, suppose that we have to do this in constant space; no more creating hash tables and arrays on the fly. Is if still possible? If you add the constraint that there be exactly one duplicate value, this is easy.

Since there are *n* values from 1 to *n*-1 with one duplicate, we can use the fact that the sum of the integers from 1 to *m* is (*m*\*(*m*+1))/2, take the sum from 1 to *n*-1, subtract that from the actual sum of numbers in the array, and the result will be the duplicated number:

let s ← 0

for 1 <= i <= n:

s ← s + A[i]

return s - n\*(n-1)/2

But suppose there are no guarantees about the number of duplicates, and the only assurance you have is that the values are between 0 and *n*, exclusive. Is it still possible to find a duplicated value in O(n) time and O(1) space? Feel free to stop here and try to work things out yourself, if you want.

As you might have guessed from the length of this entry, there is an approach that will work. It's something of a radical departure from the previous approaches, and it relies rather heavily on the particular bounds we have on the values in the array.

First, there's the solution-by-cheating. If you're working in a language with bounded integers, you can exploit that knowledge to meet the letter of the law in the "constant space" requirement.

Let's say we're using a language and platform where integers are 32 bits, signed. The maximum value for an array element would therefore be 231-1, while the minimum value would still be 1. We only need one bit of information per value (seen/not seen), so, at 8 bits per byte, an array 223 bytes (8MB) long would be enough to track any set of integers the environment is capable of passing. This algorithm is therefore O(n) time and O(1) space on a system with 32-bit signed integers:

let s ← array [1..2^31-1] of bit

for 1 <= i <= n:

if s[A[i]] = 1: return A[i]

set s[A[i]] ← 1

Of course, it relies on implementation details that might change (a similar array on a 64-bit system would require 32 petabytes of memory) and it might be nice to have an algorithm that works regardless of any artifical bounds on your integers. Fortunately, there are a couple more approaches we can take.

Recall that the array is 1-based, so its elements are numbered from 1 to *n*. The values are positive integers less than *n*, so they can range anywhere from 1 to *n*-1. Because the values are a subset of the array bounds, we can treat those values as indices into the array itself.

If we're allowed to modify the array, we can just go through it reordering the elements, trying to put each value into the corresponding element (so the value 1 goes into the first element, 2 goes into the second element, and so on). When we find a collision, that's the duplicate.

for 1 <= i <= n:

while A[i] ≠ i:

if A[A[i]] = A[i]: return A[i]

swap(A[A[i]], A[i])

If the array is immutable, though, the solution is a little more involved. Given the view of each value as an index into the array, we can treat the array as a directed graph, where each element is a node and the value of that element is an edge pointing at the next node. Since every node points to another node, there is at least one cycle in the graph. We can also conclude that every node leads to a cycle, even if it is not directly part of one.

(Side note on terminology, for those not immediately familiar: A *graph* is simply a collection of *nodes* and *edges*. Each edge connects exactly two nodes. In a *directed* graph, each edge also has a direction; it goes from one node to another. If you start at a particular node, follow (one of) its edges to the next node, continue doing that for a certain amount of time, and eventually come back to the starting node, that's a *cycle*. If you start at a particular node, follow its edges out in the same way, and end up in a cycle that does not contain the starting node, all of the nodes and edges you crossed before getting to the cycle are called the *tail*. The first node that you reach that is in the cycle is defined to be the beginning of the cycle. The symbols *λ* (lambda) and *μ* (mu) represent the lengths of the tail and cycle, respectively. Technically, the directed graph we're dealing with here is also a *sequence*, since every node has at most one edge leading away from it (though it might have more than one leading into it).)

Since the values in the array are all less than *n*, no element in the array can point to the *n*th element. Therefore, the last element cannot be part of a cycle. Since every element must lead to a cycle, it must be part of a tail. If we follow the sequence from this point, we will eventually run into a cycle, and the last element in the tail will be a duplicate value.

So, how do we find the beginning of the cycle? The easiest approach is to use [Floyd's cycle-finding algorithm](http://www.google.com/url?q=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FFloyd%2527s_cycle-finding_algorithm&sa=D&sntz=1&usg=AFQjCNF9DnH_2uxYj7AUnAC129LBTf_QNw). It works roughly like this: Start at the beginning of the sequence. Keep track of two values (call them *ai* and *aj*). At each step of the algorithm, move *ai* one step along the sequence, but move *aj* two steps. Stop when *ai* = *aj*.

At this point, the difference between *i* and *j* must be a multiple of the cycle length. If the algorithm took *m* steps to get to this point, then *i* = *m* and *j* = 2*m*, so *m* is a multiple of the cycle length. *i* has also traveled *λ* steps along the tail and *m*-*λ* steps along the cycle.

If we now start another value (call it *ak*) at the beginning of the sequence and move both it and *ai* simultaneously and at the same rate, after *λ* steps, *ak* will have traveled the entire tail and will have arrived at the beginning of the cycle. At the same time, *ai* will have moved a total of *m* steps along the cycle and, since *m* is a multiple of *μ*, will also have arrived at the beginning of the cycle. Thus, if you stop as soon as *ai* = *ak*, you will have found the beginning of the cycle and the end of the tail.

So. Applying all of that to our problem means that we start at the last element of the array, have two variables following the pointers in the array, one going twice as fast as the other, until they are equal to each other. At that point, start one of them back at the beginning and run them at the same speed until they again equal each other. Their value will then be the duplicate value.

let i ← n, j ← n

do: i ← A[i], j ← A[A[j]]; until i = j

set j ← n

do: i ← A[i], j ← A[j]; until i = j

return i

This algorithm also lends itself nicely to functional programming, since it doesn't actually require any side effects, so here's an implementation in Haskell:

import Data.Array

findDup a = findDup' n $ findCycle (a!n) (a!(a!n))

where n = snd $ bounds a

findCycle i j | i == j = i

| otherwise = findCycle (a!i) (a!(a!j))

findDup' i j | i == j = i

| otherwise = findDup' (a!i) (a!j)