>

# CodeKata

## Because experience is the *only* teacher

- RSS

Search

Navigate...

- PragDave
- Kata
- Archives

# Kata02: Karate Chop

A binary chop (sometimes called the more prosaic binary search) finds the position of value in a sorted array of values. It achieves some efficiency by halving the number of items under consideration each time it probes the values: in the first pass it determines whether the required value is in the top or the bottom half of the list of values. In the second pass in considers only this half, again dividing it in to two. It stops when it finds the value it is looking for, or when it runs out of array to search. Binary searches are a favorite of CS lecturers.

This Kata is straightforward. Implement a binary search routine (using the specification below) in the language and technique of your choice. Tomorrow, implement it again, using a totally different technique. Do the same the next day, until you have five totally unique implementations of a binary chop. (For example, one solution might be the traditional iterative approach, one might be recursive, one might use a functional style passing array slices around, and so on).

# Goals

This Kata has three separate goals:

1. As you're coding each algorithm, keep a note of the kinds of error you encounter. A binary search is a ripe breeding ground for "off by one" and fencepost errors. As you progress through the week, see if the frequency of these errors decreases (that is, do you learn from experience in one technique when it comes to coding with a different technique?).

2. What can you say about the relative merits of the various techniques you've chosen? Which is the most likely to make it in to production code? Which was the most fun to write? Which was the hardest to get working? And for all these questions, ask yourself "why?".

3. It's fairly hard to come up with five unique approaches to a binary chop. How did you go about coming up with approaches four and five? What techniques did you use to fire those "off the wall" neurons?

# Specification

Write a binary chop method that takes an integer search target and a sorted array of integers. It should return the integer index of the target in the array, or -1 if the target is not in the array. The signature will logically be:

```
1 chop(int, array_of_int)  -> int
```

You can assume that the array has less than 100,000 elements. For the purposes of this Kata, time and memory performance are not issues (assuming the chop terminates before you get bored and kill it, and that you have enough RAM to run it).

# Test Data

Here is the Test::Unit code I used when developing my methods. Feel free to add to it. The tests assume that array indices start at zero. You'll probably have to do a couple of global search-and-replaces to make this compile in your language of choice (unless your enlightened choice happens to be Ruby).

```
1  def test_chop
2    assert_equal(-1, chop(3, []))
3    assert_equal(-1, chop(3, [1]))
4    assert_equal(0,  chop(1, [1]))
5    #
6    assert_equal(0,  chop(1, [1, 3, 5]))
7    assert_equal(1,  chop(3, [1, 3, 5]))
8    assert_equal(2,  chop(5, [1, 3, 5]))
9    assert_equal(-1, chop(0, [1, 3, 5]))
10   assert_equal(-1, chop(2, [1, 3, 5]))
11   assert_equal(-1, chop(4, [1, 3, 5]))
12   assert_equal(-1, chop(6, [1, 3, 5]))
13   #
14   assert_equal(0,  chop(1, [1, 3, 5, 7]))
15   assert_equal(1,  chop(3, [1, 3, 5, 7]))
16   assert_equal(2,  chop(5, [1, 3, 5, 7]))
17   assert_equal(3,  chop(7, [1, 3, 5, 7]))
18   assert_equal(-1, chop(0, [1, 3, 5, 7]))
19   assert_equal(-1, chop(2, [1, 3, 5, 7]))
20   assert_equal(-1, chop(4, [1, 3, 5, 7]))
21   assert_equal(-1, chop(6, [1, 3, 5, 7]))
22   assert_equal(-1, chop(8, [1, 3, 5, 7]))
23 end
```

Posted by Dave Thomas (@PragDave) Dec 28th, 2013

Tweet 0 g+1 1

« Kata03: How Big? How Fast? Kata01: Supermarket Pricing »

# Comments

Disqus seems to be taking longer than usual. Reload?

## Recent Posts



Copyright © 2014 - Dave Thomas (@PragDave) - Powered by Octopress