Array Visualization

Animated sort algorithms in RubyLabs

John S. Conery

October, 2012

Array Visualizations

The lab modules for Chapter 4 ("Journey of 1000 Miles") and Chapter 5 ("Divide and Conquer") have been updated to include methods for drawing arrays on the RubyLabs Canvas. When an array is on the canvas, the sort algorithms described in those chapters will update the canvas to show how array elements are moved around by the algorithm.

Update Your RubyLabs Gem

To update your RubyLabs installation so you have these new visualization methods make sure your computer is connected to the Internet and then start your terminal emulator. Before you type `irb` to begin a session with Ruby, enter the following operating system command and hit the return key:

```
gem update rubylabs
```

You should see a message that says "Updating installed gems" and eventually a message showing which version of the RubyLabs was installed. Array visualizations are part of RubyLabs 0.9.7 and above.

Note for Linux and Mac OS X Users: If you get an error message that says something like "you do not have permission" it probably means you originally installed the lab software without using the `--user-install` option. Retype the command, but this time put `sudo` at the front of the line:

```
sudo gem update rubylabs
```

You will be prompted to enter your password (this should look familiar, since the same thing happened when your first installed RubyLabs). Type in your password, hit return, and the installation should succeed.

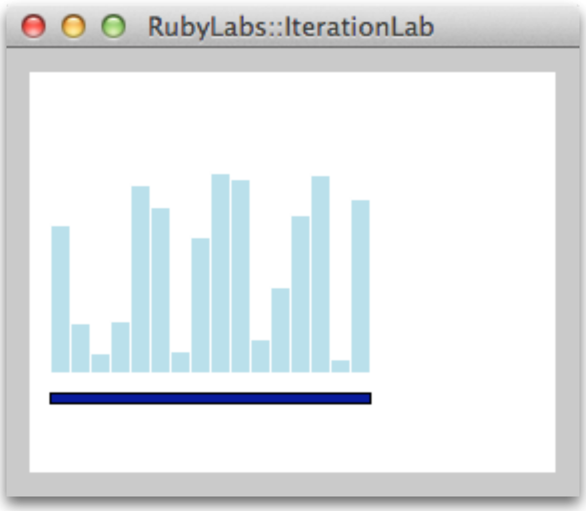

Figure 0.1: *Visualization of an array of 16 numbers. There are 16 light blue bars, one for each number in the array. The height of bar* `i` *is proportional to the value of* `a[i]`.

Drawing an Array on the Canvas

In the new version of the software a method named `view_array` is defined in the IterationLab and RecursionLab modules. If `a` is an array of integers you can draw the array on the RubyLabs canvas by passing it as an argument to `view_array`:

```
>> include IterationLab
=> Object
>> a = TestArray.new(16)
=> [70, 22, 7, 23, 90, 79, 8, 64, 96, 93, 14, 40, 75, 95, 4, 83]
>> view_array(a)
=> true
```

The result will be a drawing that looks something like the one in Figure 0.1. An array is drawn as a series of vertical bars, where the height of each bar is proportional to the value in the array.

Immediately below the array is a horizontal dark blue bar. This bar can be used by an algorithm to indicate which part of the array it is working on. For example, in `isort`, the implementation of insertion sort, the horizontal bar is drawn below the unsorted region of the array, and as the sort progresses the bar will become progressively shorter.

You can control the speed of an animation by specifying a `delay` in the call to `view_array`. The delay value tells the animation routines how long to pause after each operation. This call to `view_array` sets the delay to 0.2 seconds:

```
>> view_array(a, :delay => 0.2)
=> true
```

The default delay is fairly short, so if you want to slow down the visualization try setting the delay to longer values.

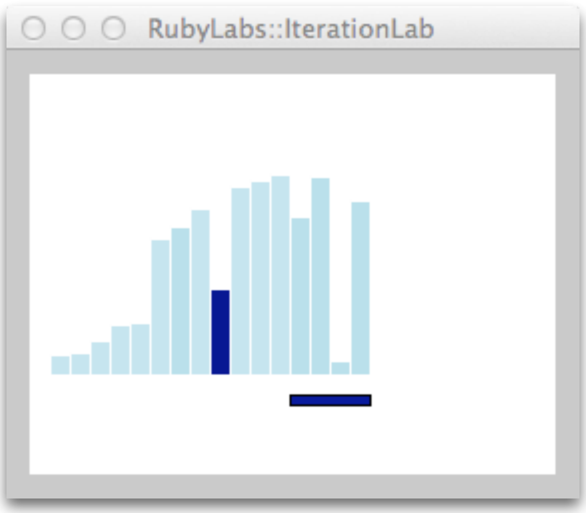

Figure 0.2: *Snapshot from the visualization of a call to* `isort(a)` *where* `a` *is the example array from Figure 0.1.*

Insertion Sort

If an array is on the canvas, a call to `isort` will cause the bars on the screen to be moved around as the array is sorted.

Recall that on iteration *i* of the insertion sort algorithm the item in location *i* is removed from the array and the computer starts scanning to the left (locations 0 to $i - 1$) to find a place to insert the item back into the array. When the array is displayed on the canvas you will see the bar for `a[i]` turn dark blue to show that it is the current item. Then you will see it start moving to the left until the computer finds a smaller item, *i.e.* it reaches a location where the bar is shorter than the bar that is moving (or until it reaches the front of the array).

Figure 0.2 shows a snapshot from the visualization of `isort` working on the example array. The horizontal "progress bar" below the array shows that the unsorted region has shrunk to the last four locations in the array and the current item is being moved to the left.

Binary Search

The "divide and conquer" algorithms (binary search, merge sort, and Quicksort) are implemented in the RecursionLab module. The version of `view_array` implemented in RecursionLab also draws an array as a set of light blue bars, but the canvas window is taller in order to make room for the extra space required by merge sort (described in the next section).

If you want to view an animation of binary search you need to attach a probe to one of the lines in the `bsearch` method. The exercises in the book show how to attach a probe so the array is printed in the terminal window with square brackets around the region that needs to be searched and an asterisk in front of the item in the middle of the region. Exactly the same approach is used for visualization: you will attach a probe, but the method activated by the probe will update the display on the canvas instead of printing the array on the console.

The first step is to create the array (don't forget to sort it) and display it. For this experiment a long delay of one second or more is recommended:

```
>> include RecursionLab
=> Object
>> a = TestArray.new(15).sort
=> [0, 10, 14, 16, 25, 28, 38, 50, 53, 54, 55, 68, 78, 80, 85]
>> view_array(a, :delay => 1.0)
=> true
```

Now attach the probe and call `bsearch` using the `trace` method:

```
>> Source.probe("bsearch", 6, "show_bsearch_region(a, lower, upper+1, mid)")
=> true
>> trace { bsearch(a, a.random(:fail)) }
=> nil
```

As the algorithm runs you should see the item being compared, which is in the middle of the current region, turn dark blue. Then the current region (indicated by the horizontal progress bar displayed below the main array) will be set to the part of the array to the left or right of the dark blue item and the next item compared will be in the middle of this new region.

Merge Sort

At any point in the execution of merge sort the algorithm is merging two adjacent groups of size n into a larger group of size $2n$. On the screen you will see the horizontal "progress bar" below the two groups that are currently being merged.

The key step in the algorithm is to compare the first item from each group and to move the smaller one to the temporary area where the merged group is being built. What you will see on the screen is the bar at the front of one of the two groups turn dark blue and then see it moved to the temporary area.

The snapshot in Figure 0.3 was taken near the end of the call to `msort`, when the algorithm was merging two groups of size 4 into a group of size 8. The progress bar is drawn below the two groups in the main array. The three smallest items from these groups have already been moved to the temporary area and then next smallest item (the vertical dark blue bar) is about to join them.

When the algorithm is finished merging two groups, the result is moved from the temporary area back to the main array directly above.

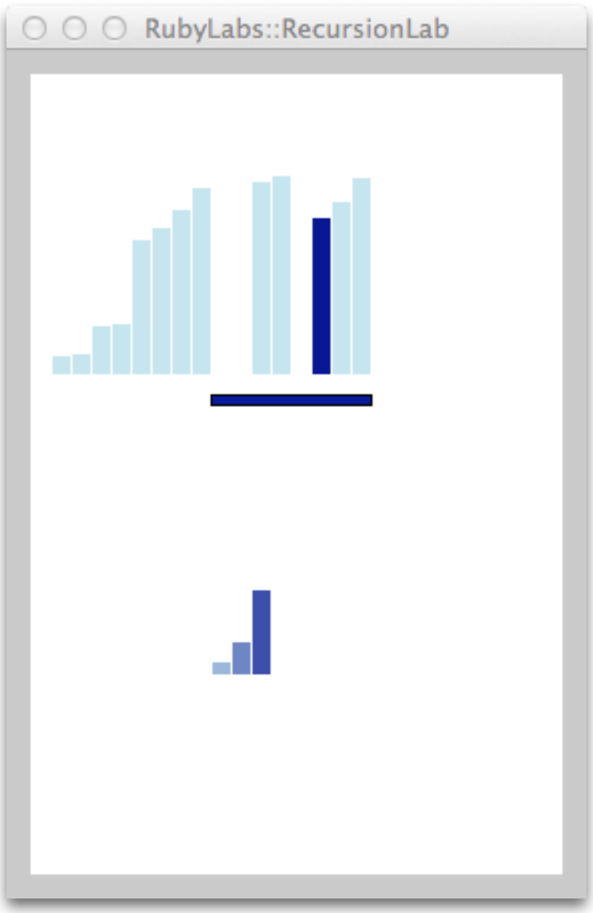

Figure 0.3: *Snapshot from the visualization of a call to `msort(a)` where `a` is the example array from Figure 0.1.*

♦ Quicksort

The main step in the Quicksort algorithm is the partitioning of the current region into items larger and smaller than some "pivot" value. The `qsort` method in RubyLabs uses a common strategy: the pivot is the first item in the region being sorted. When an array is on the canvas, `qsort` will identify the pivot by coloring it dark green. Then you will see the algorithm scan through the region moving items around. When the scan is complete, the pivot is moved to the dividing line in the middle of the region: everything to the left will be smaller than the pivot value, and everything to the right will be larger. At this point the algorithm will make two recursive calls, one to sort the items in the region to the left of the pivot and one to sort the items in the region to the right of the pivot.

The logic is more difficult to follow as a result of these recursive calls. To see how the algorithm is progressing, try to keep track of the green bars. At any point you may see a set of such bars. That means the algorithm is working on a small portion of the array, but when it finishes what it is doing it needs to "pop back up" and sort the region to the right of the closest green bar.