Raspberry Pi Activity 4: Arraynging Things

In this activity, you will implement the insertion sort. You will need the following items:

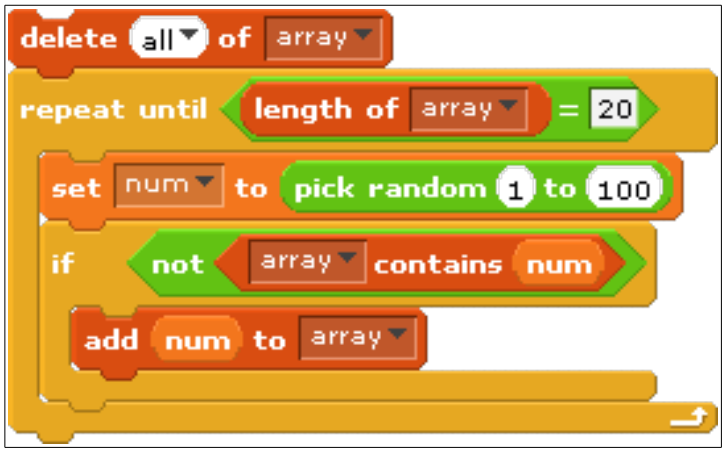
- Raspberry Pi B v2 with power adapter;
- LCD touchscreen with power adapter and HDMI cable;
- Wireless keyboard and mouse with USB dongle;
- USB-powered speakers (optional);
- Wi-Fi USB dongle; and
- MicroSD card with NOOBS pre-installed.

The goal of this activity is to implement the insertion sort in Scratch. Your script will sort a list of 20 values. You will be provided with scripts to (1) randomly populate a list with 20 unique values; and (2) perform a binary search in the sorted list for some value.

You may refer to the lesson on data structures (Introduction to Data Structures) as necessary.

Randomly populating a list with unique values

First, declare a new list and call it **array**. Code and execute the following script to populate the list with 20 unique random values:



Performing the insertion sort

Implement the insertion sort algorithm in Scratch. This is the core of this activity, and ultimately what you will submit for a grade. Although we discussed and performed the insertion sort in a previous lesson, we never developed the pseudocode for it. Let's do that now. First, recall how we described the insertion sort. It is the procedure that most people use to arrange a hand of cards. Think of the list as having both a sorted portion and an unsorted portion. The first item of the list is considered to be a sorted list one item long, with the rest of the list (items 2 through *n*) forming an unsorted portion.

The insertion sort removes the *first* item from the unsorted portion of the list and marks it as the item to be inserted. It then works its way from the *back* to the *front* of the sorted portion of the list, at each step comparing the item to be inserted with the current item. As long as the current item is larger than the

item to be inserted, the algorithm continues moving *backward* through the sorted portion of the list. Eventually it will either reach the beginning of the sorted portion of the list or encounter an item that is less than or equal to the item to be inserted. When that happens the algorithm inserts the item at the current insertion point.

The entire process of selecting the first item from the unsorted portion of the list and scanning backwards through the sorted portion of the list for the insertion point is then repeated. Eventually, the unsorted portion of the list will be empty since all of the items will have been inserted into the sorted portion of the list. When this occurs, the sort is complete.

Here's how to perform the insertion sort on the list [7 9 3 5 1]:

Pass 1			
List	**First item**	**Comparison**	**Action**
7 3 5 1	9	7 < 9	
7 9 3 5 1		done with this pass	insert 9
Pass 2			
List	**First item**	**Comparison**	**Action**
7 9 5 1	3	9 > 3	slide 9 over
7 9 5 1	3	7 > 3	slide 7 over
3 7 9 5 1		done with this pass	insert 3
Pass 3			
List	**First item**	**Comparison**	**Action**
3 7 9 1	5	9 > 5	slide 9 over
3 7 9 1	5	7 > 5	slide 7 over
3 7 9 1	5	3 < 5	
3 5 7 9 1		done with this pass	insert 5
Pass 4			
List	**First item**	**Comparison**	**Action**
3 5 7 9	1	9 > 1	slide 9 over
3 5 7 9	1	7 > 1	slide 7 over
3 5 7 9	1	5 > 1	slide 5 over
3 5 7 9	1	3 > 1	slide 3 over
1 3 5 7 9		done with the sort	insert 1

That was the long way. Here's the short way, where each pass is summarized in a single row of the table and the unsorted portion of the list is underlined at each pass:

Pass	List
original list	7 9 3 5 1
1	7 9 3 5 1
2	3 7 9 5 1
3	3 5 7 9 1
4	1 3 5 7 9

The following is the pseudocode for the insertion sort:

```
 1: n ← length of the list
 2: i ← 2
 3: repeat
 4:     if item i of list < item i-1 of list
 5:     then
 6:         temp ← item i of list
 7:         j ← i - 1
 8:         repeat
 9:             if item j of list > temp
10:             then
11:                 replace item j+1 of list with item j of list
12:             end
13:             j ← j - 1
14:         until j = 0 or item j of list not > temp
15:         replace item j+1 of list with temp
16:     end
17:     i ← i + 1
18: until i > n
```

Let's explain the algorithm a bit. Line 1 sets the length of the list in variable *n*. Throughout the algorithm, as the list changes in size (i.e., as it becomes smaller when values are removed), the variable *n* will be updated to reflect the current length of the list.

Line 2 sets the current item in the list to be inserted (the second value in the list). Recall that, initially, the insertion sort assumes the first item in the list to be the sole item in the unsorted portion of the list. Trivially, a single item is sorted! So the insertion sort begins with the second item.

Line 3 defines repetition that controls each pass through the list. At each pass, the first value in the unsorted portion of the list will be inserted into its proper position in the sorted portion of the list. The loop iterates from the second value through the last value in the list, using the variable *i*.

Line 4 checks to see if the current value to be inserted into the sorted portion of the list happens to be smaller than the last value currently in the sorted portion of the list. If this is *false*, then it is already in

its proper position (i.e., it is the largest value so far and belongs at the end of the sorted portion of the list).

Line 6 stores the value to be inserted in the variable *temp*.

Lines 7 through 14 compare the value to be inserted with the values in the sorted portion of the list (from right-to-left) using the variable *j*. Any value that is found to be larger than the value to be inserted (in the variable *temp*) is moved to the right one index. This occurs either until the beginning of the sorted portion of the list has been reached (i.e., *j*=0) or a value in the sorted portion of the list is less than the value to be inserted (i.e., *j* is the index **before** the proper position for the value to be inserted).

Line 15 is the last statement executed at each pass through the list. It inserts the value to be inserted (stored in the variable *temp*) into its proper place in the sorted portion of the list.

Homework: Insertion Sort

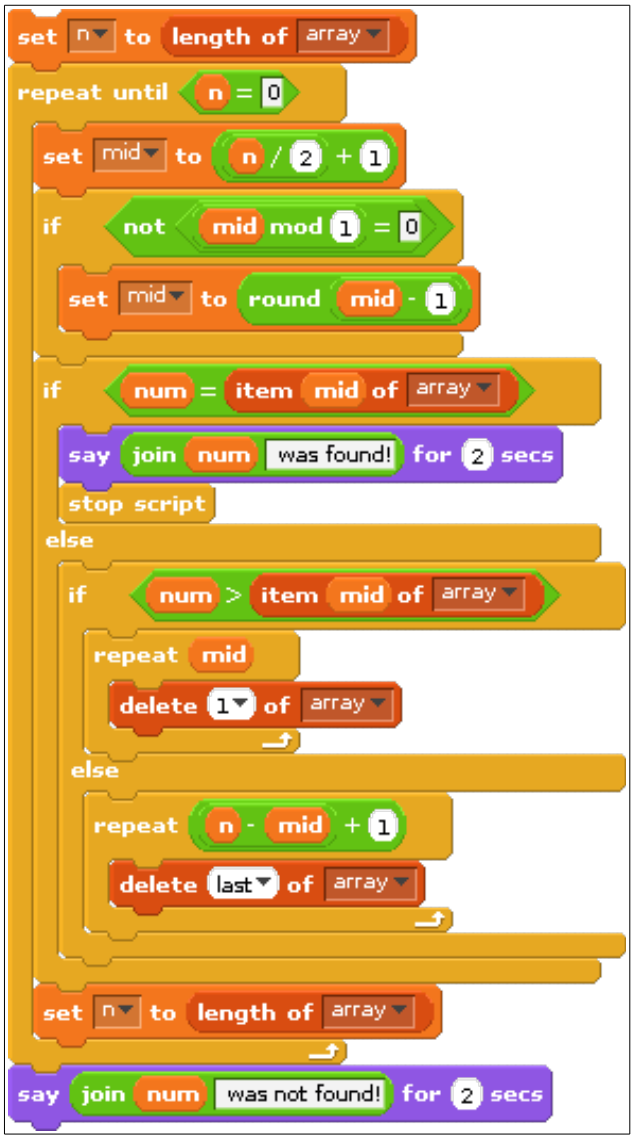
Your task, should you choose to accept it, is to implement the above pseudocode in Scratch as a script. Of course, you can test your script on the randomly populated list of 20 values to see if it works (i.e., it properly sorts the list).

The only requirement in terms of scripts in your submission is the insertion sort. However, if you include any other provided scripts, that's fine.

You are to submit your Scratch v1.4 (not v2.0!) file (with a .sb extension) through the upload facility on the web site.

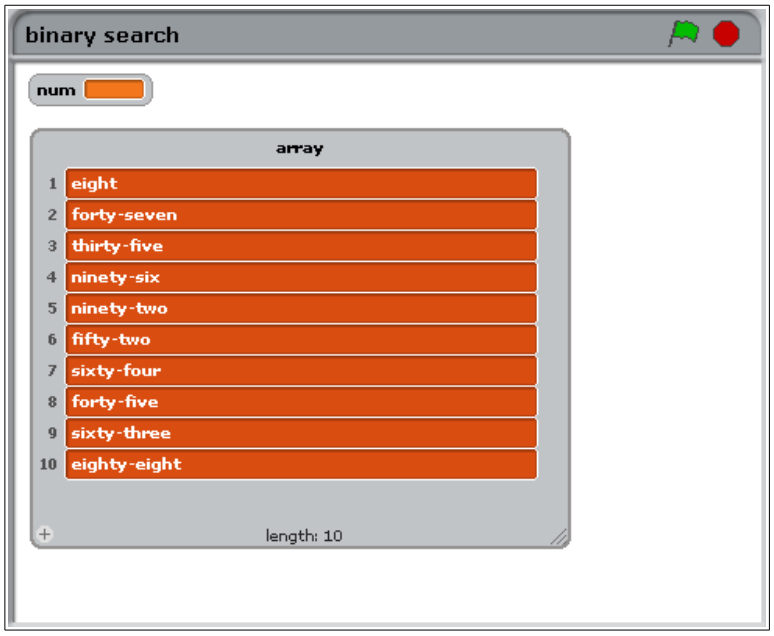
Searching through the list with the binary search

When your insertion works properly (and you have a sorted list), you can implement the binary search algorithm as follows:

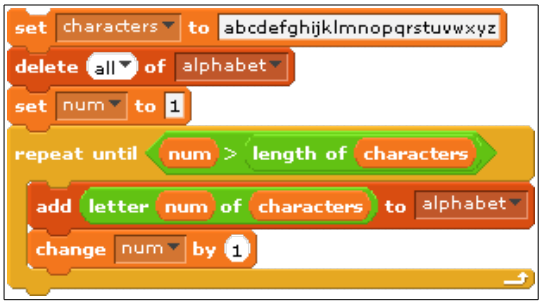


Of course, you will have to declare the proper variables (i.e., *mid* and *num*) and set an initial value for *num* (choose several, some of which are in the list and one of which is not).

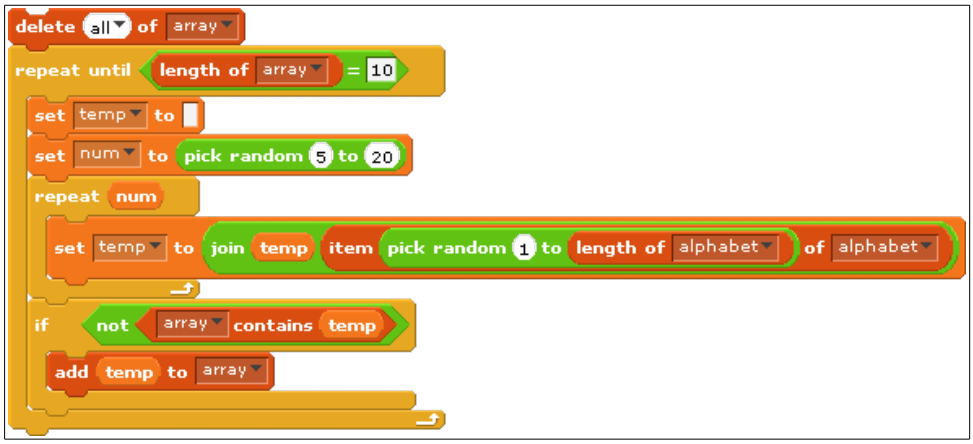
An experiment!
Try changing the list so that it contains text data as opposed to numeric data. For example, try filling the list with values like: thirty-seven, forty-five, eighty-nine, seven, eight, twenty-four, and so on. Or fill the list with random strings of characters. Here's an example:



Does your insertion sort algorithm still work (i.e., does it properly sort the list of text values in alphabetical order)? Does the binary search algorithm still work? Try it with a value that's in the list and one that's not. You may also want to code a script that creates a list of random text values. How would such a script work? Here's one way of generating random strings of characters of random lengths. First, we create a text variable (called *characters*) that contains all the valid characters that we want to include in our strings. We use this text variable to generate a list of its unique characters:



The script iterates through each letter of the variable characters and adds each one to the list *alphabet*. From there, we populate a list with random strings of random lengths:



Again, we limit the number of items in the list to 10. Initially, we clear the variable *temp* (it will eventually store one of the strings that we want to add in the list). We pick a random length (from 5 to 20) for each string. The string is then iteratively built, one letter at a time, from random letters in the alphabet. So long as the list does not contain the randomly generated string, it is added! And that's it!

Use these scripts to test if your insertion sort script and the provided binary search script works on text data.