

### The need for data structures

The algorithms we design to solve problems rarely do so without requiring some sort of input and producing some sort of output. In the process, our algorithms do something with the inputs (e.g., number crunching, processing, and so on). That is, algorithms typically manipulate the inputs in some way. Think about what that means. Where is the information? What does it *look* like? How is it accessed? How is it manipulated? Where does it go? We generally refer to the inputs being processed and the output(s) being generated as data.

**Definition:** *Data is a term given to pieces of information that can be represented, stored, or manipulated using a computer. Often, combining data provides meaning (i.e., information).*

Although data is useful and necessary, it is not yet meaningful. The idea is that our algorithms will process this data and produce some sort of output (or result). In the process, meaning is given to the data. We call this information.

### Data structures

Data structures have to do with arranging or organizing data in some way. The memory capacity in today's computers is very large. Within the computer, data is stored in different memory locations. Often, the many pieces of data that our algorithms are dealing with are related in some way. Consequently, there is a need to have this data grouped in some way in memory. This grouping makes manipulation of that data much easier. Think about sorting a list of numbers. It would be much more difficult if the numbers were located randomly in memory. Somehow, we would need to know where each value is located, and that could technically be anywhere! Perhaps it would speed things up if each value was located in consecutive memory locations (i.e., next to each other in memory). We would then only need to know where the first value is located and the total number of values stored.

Data structures allow a programmer to arrange pieces of information (data) in a way that makes sense and allows the computer to manipulate the data easily for the programmer's task. Many times this data is made up of multiple instances of similar pieces of data, and other times it involves different kinds of data which are related. For example, a word (or a bunch of letters strung together) is made up of multiple pieces of similar data kept together in a specific order. In the case of a word, the letters of the alphabet are the pieces of data, and they have to be kept together in a specific order for it to make sense. Another example is a class roster which is made up of different entries in a specific order corresponding to each student in the class. Each entry is made up of dissimilar data such as the name and the grade on exam 1 of a student.

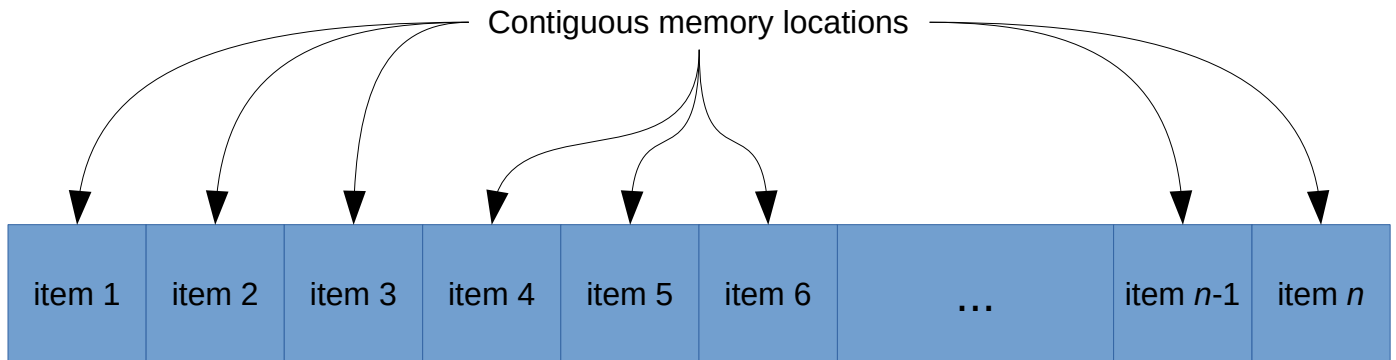
**Definition:** *A data structure is a way of organizing data in a computer so that it can be used efficiently.*

### Array

One of the most commonly used data structures is called the **array**. We will see that many concepts in computer science, and particularly in data structures, are derived from real life examples and arrays are not an exception. Arrays are comparable to a numbered list such as a grocery list, a class roster, or a set of numbered drawers. They are used to store multiple instances of anything, as long as they are all of the same kind (i.e., all numbers, all letters, all images, all books, etc). Imagine these things being in

some sort of order (i.e., we have a first thing, a last thing, and some number of things in between). The members of (or entries in) the array are called elements.

**Definition:** An *array* is a collection of similar pieces of data stored in contiguous memory locations. Contiguous memory locations means that the data is stored in memory locations that are next to each other.



The order in which elements are stored in an array is important. This is because very often a programmer needs to access a specific element of an array, and in order to do that, its position relative to the first element of the array must be known. The position of an element is also referred to as its *address*, and the relative address (how far away from the first element it is) is called its *index*.

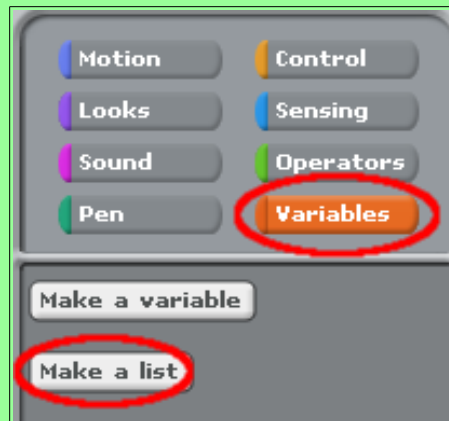
In the Scratch programming language, arrays are implemented using a *list*. The indices (relative positions) of lists in Scratch begin at index 1 and continue through the length of the list (i.e., index  $n$  for a list of  $n$  elements). This means that if you have a list of four elements, the indices begin at 1 and continue through 4 (i.e., the first element will have an index of 1, and the last element will have an index of 4). You would run into problems if you tried to access the element at index 5, because the program is only aware of elements 1 through 4. It is as if you had a list of four students, and someone asked you who the fifth student was. Similarly, if you had a list of 100 elements in Scratch, its indices would start at 1 and end at 100. The program would be unable to access an element at index 101 (or even index 0 or -1) because such an element does not exist.

**Activity 1: Creating and populating an array**

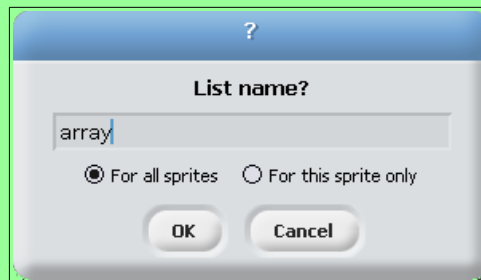
In this activity, you will create a list of 20 numbers in Scratch. You will then apply the sequential search from a previous lesson to find a specific number in the list.

## Creating a list

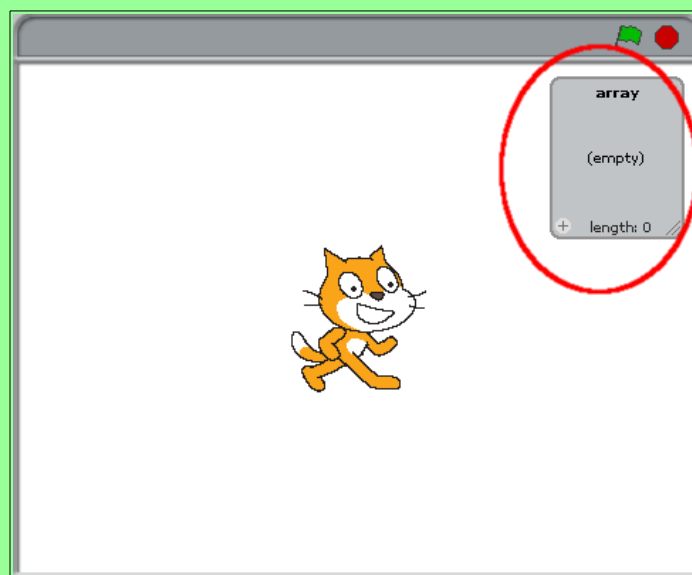
To create the list, select **make a list** from the variables blocks group:



This should bring up a pop-up menu which you can use to name the list. For now, let's call this list *array*:



After clicking **OK**, you should see two changes to the Scratch interface. First, there should be a component in the **stage** of the Scratch interface. The component has the name of the list you just created, any information that is stored in the list (which is currently empty), and the length of the list (which is currently 0):



The second effect of creating this list is that there are now more puzzle pieces in the variables blocks group. These puzzle pieces allow us to do a lot of things with our list; for example, we can add values to and remove values from the list. We can even delete the entire list.



### Populating a list with random numbers

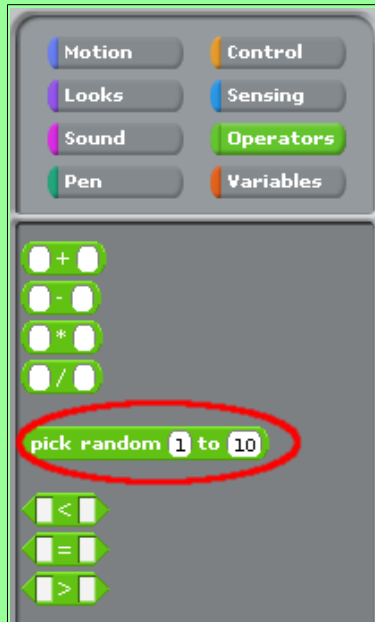
Let's populate the list with 20 random numbers. While it is possible (albeit tedious) to populate the list by adding each value individually using **add thing to array**, we are going to create a short script using other puzzle pieces to make this population process automated.

Because this population process calls for a specific task (addition of an item to the list) to be repeated over and over again, let's use a **repeat-n** construct from the control blocks group (as well as the **add thing to array** instruction from the variables blocks group). Because our task is going to be done 20 times (in order to fill the list with 20 values), we change the value in the repetition block to 20:



Now what “thing” will we be adding to the array? We want to add 20 randomly selected numbers to the list. We find the appropriate puzzle piece for selecting a random number under the operators blocks

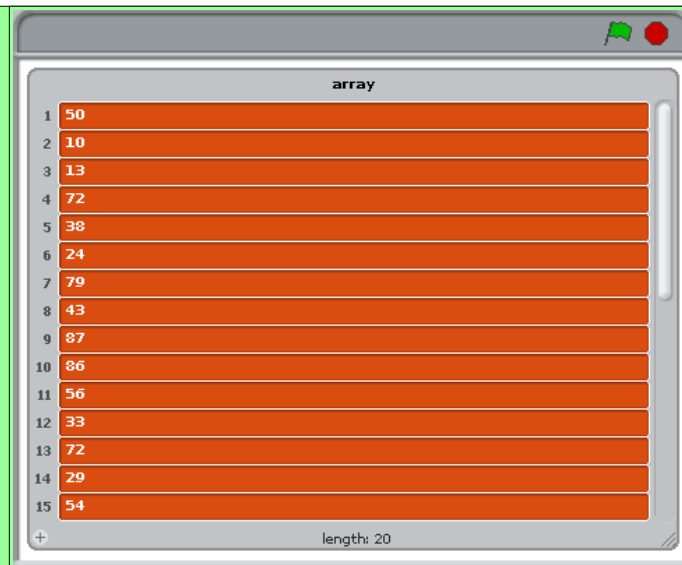
group. This puzzle piece is used to generate a random number whose value is between the two arguments (which are 1 and 10 by default).



Drag this piece and place it in the “thing” argument position of **add thing to array** that is already in the Scripts area. Edit its arguments such that it will pick a random number between 1 and 100.



When you click the repeat block, the script will be executed. This will fill the array with 20 random numbers (from 1 to 100). Feel free to resize the list so that you can see more of the numbers stored in it at once:

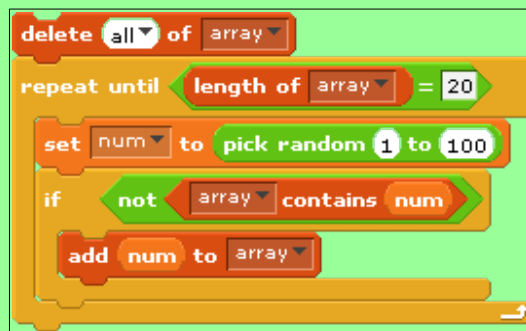


You have now created a list (which we called *array*) and filled it with 20 random numbers. If any of the following activities require a list filled with random numbers, you can easily refer to the above steps and create one. In fact, the rest of the activities in this lesson will require a list filled with numbers. If you require a larger or smaller array, it's just a matter of changing the value in the repeat block. If you require values in a different range (perhaps numbers between 100 and 1000), it's just a matter of changing the values in the **pick random** block.

Note that if you continue to click on the **repeat-n** block, more values will be added to the list (i.e., it will keep growing). To reset the list and add 20 fresh values, clear it first by modifying your script as follows:



Note that the script above can add duplicate values to the list. How could it be modified to ensure that the list only has unique values? A possible solution is:



There are several changes. The first is that the **repeat-n** loop has been changed to a **repeat-until**. This is because of the **if** statement inside the loop. If a duplicate value is found (via the **not array contains num** block) then it is not added to the list. It is possible that a duplicate value is selected that does not get added to the list and takes up one of the 20 iterations of the original **repeat-n** loop. This could result in the list having less than 20 values. To deal with this, we repeat the loop statements until the length of the list is 20 (i.e., 20 unique values have been added to the list).

Another change is that the random number is stored in the variable **num**. This is because we need to access the random number up to two times (first to check if it is already in the list, and second to add it to the list if it is not a duplicate). Suppose that the Boolean expression were instead, **not array contains pick random 1 to 100**. If the result of this expression is *true* (i.e., the list does not contain the random number), then an **add pick random 1 to 100 to array** block (as in the first script) would simply add a different random number to the list. To ensure that the one checked in the Boolean expression is the same as the one added in the *true* part of the **if** statement, we store the random number to the variable **num** and use it in both blocks.

Now that we have an array populated with random values, we can implement the sequential search to find the largest value in the array!

## Activity 2: Sequential search of the largest value in an array

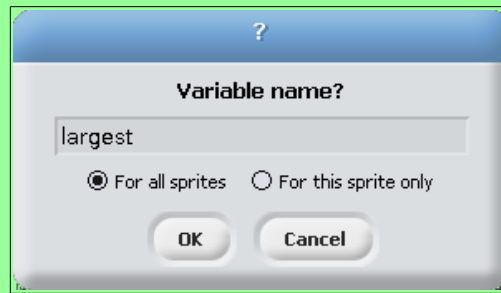
This activity involves implementing a sequential search on the list that was created in the first activity. We will use the sequential search to find the **largest** value in the list. We need to use our knowledge of the sequential search algorithm to implement it using the tools available to us in Scratch.

Just to jog your memory, the sequential search algorithm begins by assuming that the largest value is the value stored in the first position of the list. We then check through every position, one by one, to see if there is a value greater than what is currently stored as largest. If a larger value is found, then the largest is replaced with the value in the current position. Otherwise, we just move on to the next position. This process is repeated until the end of the list is reached.

The first thing to do is to create a variable called **largest** and assign the value in the first position of the list to it. Create the variable in the variables blocks group:



You should then be presented with a window in which you can type in the variable name (**largest** in this case):



At this point, a component identifying the new variable should be in the upper-left of the stage. It is initialized with a default value of 0:



There are also new puzzle pieces under the variables blocks group that allow you to set, change, show, and hide **largest**.





We now need to store the value in the first position of array to **largest**. To set the value of a variable, we use the **set largest to 0** block in the variables blocks group. Drag it to the scripts area and combine it with the **item 1 of array** block. This way, the value stored in position 1 of the list will be stored in **largest**:



When this puzzle piece is clicked, the value of **largest** shown in the stage should change from 0 to whatever value happens to be in the first position of your list (32 in the following example):



Now that we have initialized **largest**, we need to go through each value in the list and compare it with the variable largest. This sounds like we'll need the **repeat-n** block from the control blocks group (to do the comparison 20 times: one comparison for each of the values in the array). Technically, we really only need to do the comparison 19 times: one comparison for each of the remaining values in the array). We will also need the **greater than** block from the operators blocks group (to determine if a value from the list is greater than **largest**) and the **if** block from the control blocks group (to selectively update **largest** if a larger value is found):



Recall that the sequential search algorithm dictates that if the value we are comparing with in the list is greater than **largest**, then we change the value of **largest** to store the current value in the list. Otherwise we keep on comparing with the remaining values in the list.

To compare **largest** with individual elements in array, we will need to know the indices of each of the values in the array. That is, we will need to start with the second index (why not the first?), compare it with largest, go to the third, and so on. So we will need another variable in our algorithm which will just store the index (or position) of whichever value we happen to be comparing **largest** with at the time. Let's call this variable **counter**. Create it in the same way that you created **largest**.

You'll notice that the set, change, show, and hide blocks in the variables blocks group now have drop down menus in them which you can click to change between all the different variables that you have created. At this point, the drop down menu has just two variables: **counter** and **largest**:



We can now use **counter** to refer to specific elements in the list. For example, we already have a block that sets **largest** to **item 1 of array**. We can use the variable **counter** instead of explicitly using the number 1. The benefit of using a variable instead of an explicit number is that a variable can change at any point in our algorithm.

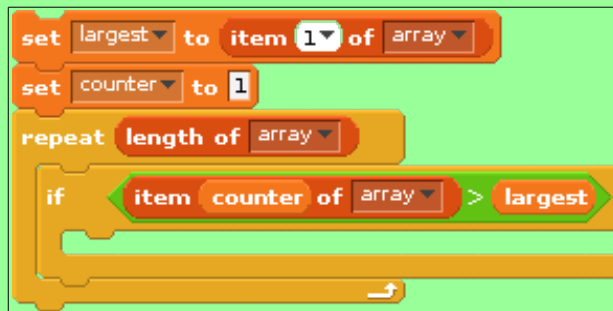
Start the sequential search algorithm as follows:



```
set largest to item 1 of array
set counter to 1
repeat 20
  if <> > <>
```

So far, we've initialized **largest** to the first value in the array, initialized **counter** to 1, setup the algorithm to repeat some number of times a comparison (undefined so far).

The next step is to tune the repeat-n construct to repeat the blocks in the loop 20 times (or the same number of times as there are items in the array). We will also need to setup the Boolean expression so that it compares the current value in the array (at the **counter**) with **largest**:



```
set largest to item 1 of array
set counter to 1
repeat length of array
  if item counter of array > largest
```

Since we want to compare **largest** to every item in the array (there are exactly **length of array** of them!), we repeat **length of array** times. Each time we do so, we compare the current item in the list (**item counter of array**) with **largest**.

There are only two more things left to do: first, if we have found a larger value in the list, then we must update **largest** to reflect this; second, we must increment **counter** at the end of each repetition:



```
set largest to item 1 of array
set counter to 1
repeat length of array
  if item counter of array > largest
    set largest to item counter of array
  change counter by 1
```

Note that the **set largest to item counter of array** in the loop will only be executed if **item counter of array** is greater than **largest**. This is because that particular block is within the **if** statement. Statements within the **if** block are only executed when the **if** block Boolean expression (which is **item counter of array > largest** in this case) is **true**. If the **if** block Boolean expression is **false**, then all the puzzle pieces within the **if** block are skipped.

Note where the increment of **counter** (via the **change counter by 1** block) is placed in the script. We want **counter** to change at the end of every loop (and not just when the **if** block is executed and a larger value in the list has been found). This means we need to put it within the **repeat-n** block, but outside of the **if** block.

The sequential search algorithm is now complete. If you run the program (by clicking the top puzzle piece), it should execute and change the values of **largest** and **counter**. At the end of its execution, **largest** should have the value of the largest item in the list, and counter should be 21 (why 21 and not 20?). The figures below shows what the stage looks like before and after you execute the program:



### Did you know?

While it is technically referred to as a list in the Scratch programming language, there is a distinction between a *list* and an *array* in other programming languages that will become more apparent later on. For our purposes right now, a list in Scratch behaves in a similar manner to a traditional array, and the terms will be used interchangeably for now.

### Value vs. index

The sequential search is an algorithm that has very many applications. With just a few adjustments to the algorithm, we can search for a specific number as opposed to searching for the largest or smallest number. It is an algorithm that computer scientists often make use of in one form or another.

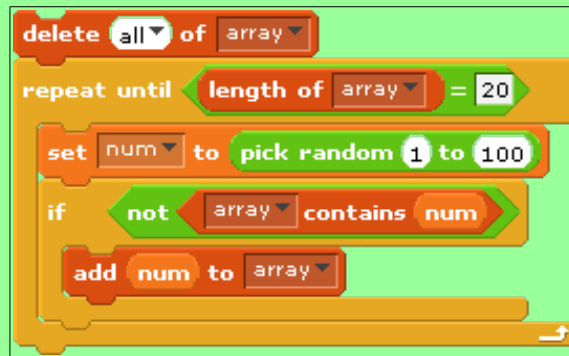
The distinction between a value and its index is one that must be emphasized. In this context, a value refers to a piece of data stored in an array, and its index is the position in the array where that value is stored. The index represents *where* an element is, and the value represents *what* the element is. While

the two are related, each of them will be of different importance to us depending on the scenario we are trying to solve. For example, if you misplaced your favorite jacket at home, its location would be more important than its value (i.e., its index would be more important than the fact that it is a jacket). In contrast, when you get feedback on a test you did in class, the value of your score is more important. While dealing with lists (and later on with arrays), it is important to understand the distinction between index and value.

One category of algorithms that we have already covered and relies heavily on the distinction between index and value is sorting. Our next activity will involve sorting an array of randomly assigned values. More specifically, we will implement the selection sort to order the list of values.

### Activity 3: Selection sort of an array

For this activity, you will make use of the script that randomly populates a list with unique values from 1 to 100. Here it is again for reference:



Recall that this script requires a list to be declared first (called **array** above).

Now we will implement the selection sort. First, let's look back at the pseudocode for the algorithm:

```
1:  $n \leftarrow$  length of the list
2: for  $i \leftarrow 1..n-1$ 
3:    $minPosition \leftarrow i$ 
4:   for  $j \leftarrow i+1..n$ 
5:     if item at  $j <$  item at  $minPosition$ 
6:     then
7:        $minPosition \leftarrow j$ 
8:     end
9:   next
10:  swap items at  $i$  and  $minPosition$ 
11: next
```

The selection sort works by building a sorted list from left-to-right. Initially, the smallest value is located in the list and swapped with the first item in the list. The sort repeats this process with the next unsorted element (i.e., the first item in the unsorted portion of the list). Each iteration, a *minPosition* is updated that reflects the position (or index) of the smallest value in the list so far. Once the entire list has been searched through, a swap is made (swapping this minimum value with the first value in the unsorted portion of the list).

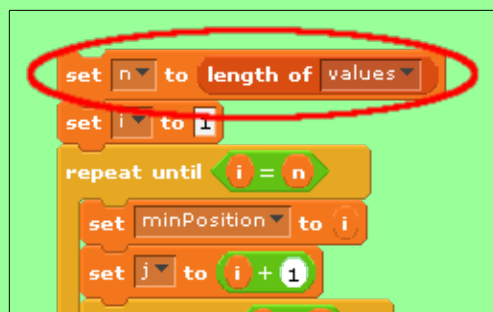
Here is the algorithm in Scratch:



```
set n to length of values
set i to 1
repeat until i = n
  set minPosition to i
  set j to i + 1
  repeat until j > n
    if item j of values < item minPosition of values
      set minPosition to j
    change j by 1
  set temp to item i of values
  replace item i of values with item minPosition of values
  replace item minPosition of values with temp
  change i by 1
```

Note that the list is called **values** in the script above; however, you can call it what you want. Let's break the algorithm down, step-by-step. The first statement is pretty evident and is easily translated to Scratch blocks:

1:  $n \leftarrow$  length of the list



```
set n to length of values
set i to 1
repeat until i = n
  set minPosition to i
  set j to i + 1
```

The selection sort makes use of two loops, one inside the other. The outer loop controls the number of passes made through the list, each time placing the next smallest value in the list. The inner loop finds the next smallest value by comparing each value in the list to the current minimum.

```
2: for  $i \leftarrow 1..n-1$ 
3:    $minPosition \leftarrow i$ 
4:   for  $j \leftarrow i+1..n$ 
5:     if item at  $j <$  item at  $minPosition$ 
6:       then
7:          $minPosition \leftarrow j$ 
8:       end
9:   next
10:  swap items at  $i$  and  $minPosition$ 
11: next
```

Since Scratch doesn't have a **for-next** loop, we have to use a repetition construct that can function in a similar way. A **for-next** loop is similar to a **repeat-n** loop; however, the **repeat-n** loop doesn't permit us to know the number of the current iteration. We can use a **repeat-until** loop instead, as discussed in a previous lesson. We can rewrite the selection sort so that it uses **repeat-until** loops instead of **for-next** loops as follows:

```
1:  $n \leftarrow$  length of the list
2:  $i \leftarrow 1$ 
3: repeat
4:    $minPosition \leftarrow i$ 
5:    $j \leftarrow i + 1$ 
6:   repeat
7:     if item at  $j <$  item at  $minPosition$ 
8:       then
9:          $minPosition \leftarrow j$ 
10:    end
11:     $j \leftarrow j + 1$ 
12:  until  $j > n$ 
13:  swap items at  $i$  and  $minPosition$ 
14:   $i \leftarrow i + 1$ 
15: until  $i = n$ 
```

The outer loop, **for**  $i \leftarrow 1..n-1$ , in the original pseudocode is replaced with the following:

```
 $i \leftarrow 1$ 
repeat
  ...
   $i \leftarrow i + 1$ 
until  $i = n$ 
```

The **for** loop internally initializes  $i$  to 1 and (also) internally increments it by 1 at the end of the loop statements. By switching to a **repeat-until**, however, we must manually initialize  $i$  and increment it at the end of the loop statements. In addition, we must specify the exit condition that breaks out of the loop. Since we increment  $i$  at the end of the loop statements and we want the loop statements to execute through  $i = n-1$ , we can then set the exit condition to  $i = n$ . So when  $i$  is equal to  $n$ , we will break out of the loop.

Step through the new pseudocode and convince yourself that the loop statements are executed the same number of times as the original pseudocode. Suppose that  $n=10$ . In the original pseudocode, the loop statements are executed so long as  $i=1..n-1$ , or 9 times. In the revised pseudocode, the loop statements are also executed 9 times.

The inner loop, **for**  $j \leftarrow i+1..n$ , in the original pseudocode is replaced with the following:

```
j ← i + 1
repeat
  ...
  j ← j + 1
until j > n
```

The **for** loop internally initializes  $j$  to  $i+1$  and (also) internally increments it by 1 at the end of the loop statements. Again, we must now manually initialize  $j$  and increment it at the end of the loop statements. We must also specify the exit condition that breaks out of the loop. Since we want the loop statements to execute through  $j = n$ , we can then set the exit condition to  $j > n$ . So when  $j$  is equal to  $n+1$ , we will break out of the loop.

Step through the new pseudocode and convince yourself that the loop statements are executed the same number of times as the original pseudocode. Suppose that  $n=10$  and  $i=5$ . In the original pseudocode, the loop statements are executed 5 times. In the revised pseudocode, the loop statements are also executed 5 times.

And now you see why the Scratch version of the algorithm utilizes **repeat-until** constructs:



Also note that variable initializations (above each **repeat-until** construct), and the variable increments (at the end of each set of loop statements).



The statements in the inner loop are almost a direct translation of the pseudocode to Scratch blocks. The only thing left to discuss is the swap: swap items at  $i$  and  $minPosition$ . There is no swap block in Scratch (nor is there such an instruction in most programming languages). We must therefore manually swap the two values in the list.

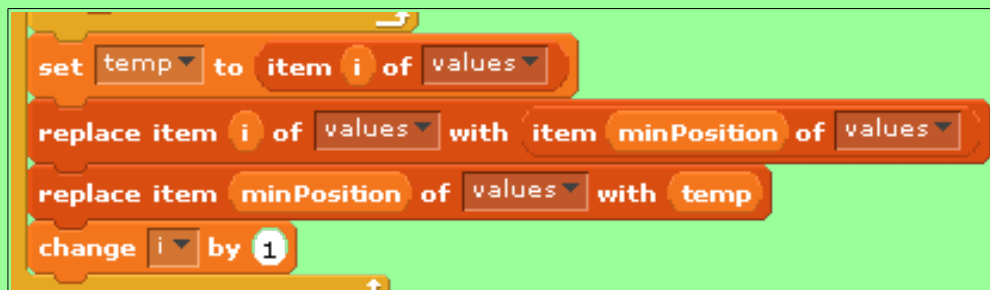
Swapping is typically done by declaring a temporary variable, assigning it one of the two values to be swapped, and then performing successive assignments. This is best described with an example.

Suppose that we wish to swap the values of two variables,  $num1$  and  $num2$ . A simple algorithm to do this is:

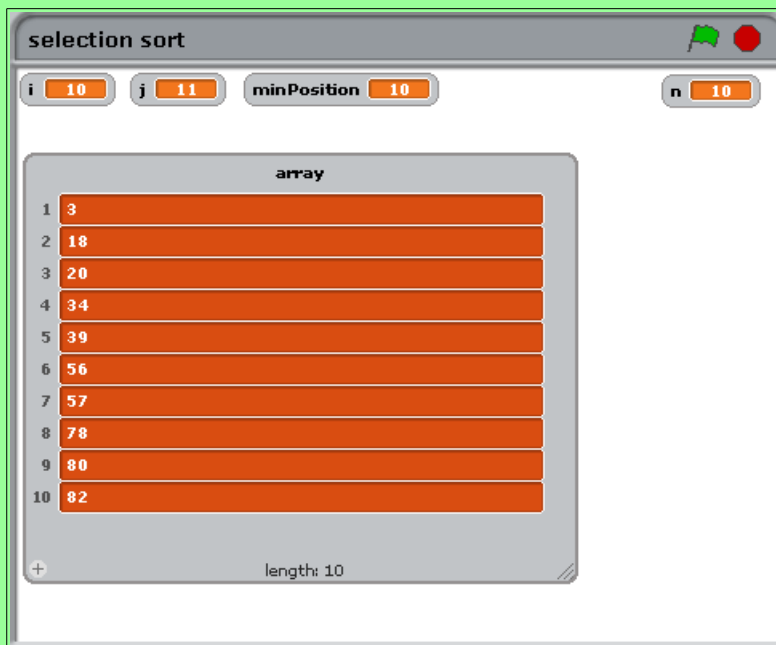
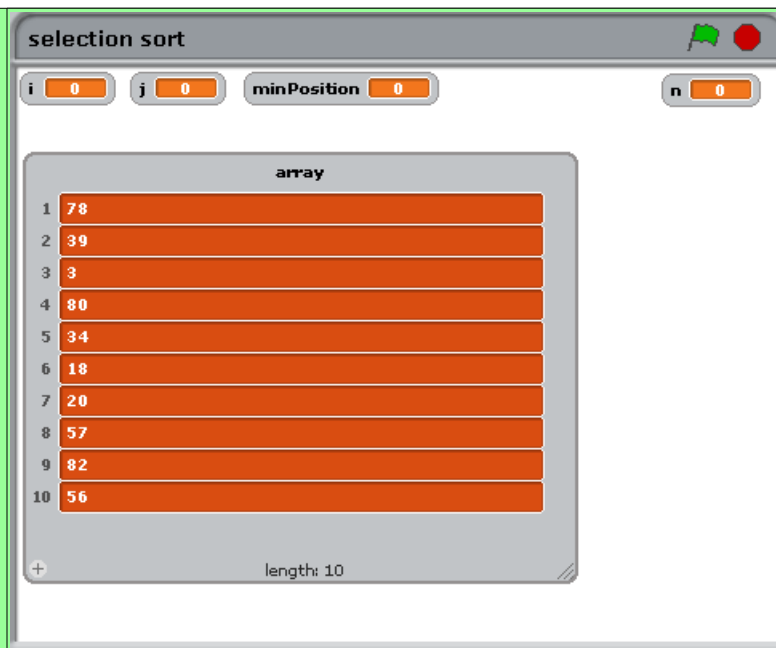
```
temp ← num1
num1 ← num2
num2 ← temp
```

The process begins with  $temp$  storing the value of  $num1$ . This now allows  $num1$  to be changed to the value stored in the variable  $num2$ . Finally, we can assign  $num1$ 's old value (now stored in  $temp$ ) to  $num2$ , completing the swap.

The selection sort algorithm in Scratch makes use of this by storing the value at index  $i$  of the list in a temporary variable ( $temp$ ). It then replaces the item at index  $i$  of the list with the item at  $minPosition$ . Finally, it replaces the item at index  $minPosition$  of the list with  $temp$  (the item that used to be at index  $i$  of the list).



To test the entire algorithm, first create a list of 10 or 20 values and randomly populate it with unique values using the appropriate script above. Then run the selection sort script and watch the list become sorted! Here's some sample output of this:



Now that we have implemented the selection sort, we can sort any array of values! And now that we have a sorted array, we can implement a more efficient search than the sequential search. Recall that there exists a more efficient search that only works on sorted data: the binary search. Let's try to implement it now in the next activity.

#### Activity 4: Binary search for a specific value in an array

For this activity, we assume that you have declared and randomly populated a list of values (called **array**), and that you have also sorted this array using the selection sort implemented in the previous activity.

Let's begin by recalling the pseudocode for the binary search as shown in a previous lesson:

```
1: repeat
2:      $n \leftarrow$  number of items in the current portion of the list
3:      $mid \leftarrow$  floor( $n / 2$ ) + 1
4:     guess  $mid$ 
5:     if response is HIGHER
6:     then
7:         discard the left half of the list
8:     else if response is LOWER
9:     then
10:        discard the right half of the list
11:    end
12: until guess is correct
```

Note that the repeat-until loop terminates when the guess is correct. This is because we tailored the binary search to the number guessing game (which means that, eventually, the number will be found). We need to generalize the algorithm so that it can work on an arbitrary list of values, whether the specified value is found in it or not. Try to rewrite the algorithm so that it works for an arbitrary list:

Here, we store that value to search for in the variable *num*, and the length of the list in the variable *n*. The search will continue so long as the list has at least one item (i.e., until its length *n* is 0). At each iteration, its middle is calculated and stored in the variable *mid*. Recall that the middle is calculated as:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1$$

At this point, the algorithm compares the middle value with *num*. If the values are equal, an appropriate message is displayed and the algorithm terminates. If *num* is greater than the middle value, then the left half of the list is discarded (i.e., items 1 through *mid* are removed). If *num* is less than the middle value, then the right half of the list is discarded (i.e., items *mid* through *n* are removed). Of course, the length of the list is recomputed after one half is potentially discarded. If the value is never found, then at some point the list will become empty (i.e., all values will be removed). The loop will then terminate, and an appropriate “not found” message will be displayed.

Take a look at how we can implement this version of the binary search in Scratch:

```

set n to length of array
repeat until n = 0
  set mid to n / 2 + 1
  if not mid mod 1 = 0
    set mid to round mid - 1
  if num = item mid of array
    say join num was found! for 2 secs
    stop script
  else
    if num > item mid of array
      repeat mid
        delete 1 of array
      else
        repeat n - mid + 1
          delete last of array
    set n to length of array
  say join num was not found! for 2 secs

```

We obtain the requested value *num* by prompting the user for it:

```

ask What value do you want to search for? and wait
set num to answer

```

Note that Scratch has no *floor* function. To replicate this function, we simply calculate  $mid$  as  $n / 2 + 1$ , and then check to see if  $mid$  is not a whole number (i.e., if it is not evenly divisible by 1). If so, then we round down (well, we technically subtract 1 from  $mid$  and round up – which has the same effect):



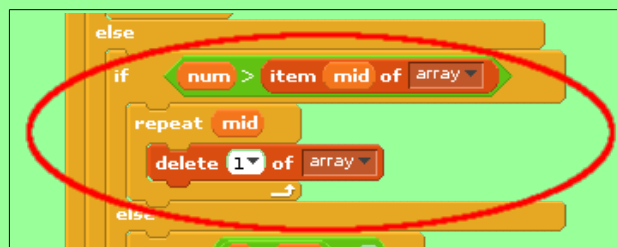
```
set n to length of array
repeat until n = 0
  set mid to n / 2 + 1
  if not mid mod 1 = 0
    set mid to round mid - 1
  if num = item mid of array
```

If the specified value is found, then we display an appropriate message and terminate the script:



```
set mid to round mid - 1
if num = item mid of array
  say join num was found! for 2 secs
  stop script
else
  if num > item mid of array
```

If the specified value is greater than the value at index  $mid$  in the array, we discard the left half of the list. This is accomplished by repeatedly removing the first item in the list,  $mid$  times:



```
else
  if num > item mid of array
    repeat mid
      delete 1 of array
    else
```

The **delete 1 of array** block removes item 1 of the array. Suppose the length of the list,  $n$ , is 10 and  $mid$  is 6:  $\text{floor}(10 / 2) + 1 = 6$ . Further suppose that  $num$  is greater than the value at  $mid$ . So we discard the left half of the list by repeatedly removing the first item in the list 6 times:

List	Action	Removal Count
1 2 3 4 5 <u>6</u> 7 8 9 10	original list	
2 3 4 5 <u>6</u> 7 8 9 10	remove first value	1
3 4 5 <u>6</u> 7 8 9 10	remove first value	2
4 5 <u>6</u> 7 8 9 10	remove first value	3
5 <u>6</u> 7 8 9 10	remove first value	4
<u>6</u> 7 8 9 10	remove first value	5
7 8 9 10	remove first value	6

Notice how this process effectively discards the left half of the list.

If the specified value is less than the value at index  $mid$  in the array, we discard the right half of the list. This is accomplished by repeatedly removing the last item in the list,  $n - mid + 1$  times:



The **delete last of array** block removes item  $n$  (the last item) of the array. Again, suppose the length of the list,  $n$ , is 10 and  $mid$  is 6. Further suppose that  $num$  is less than the value at  $mid$ . So we discard the right half of the list by repeatedly removing the last item in the list 5 times ( $10 - 6 + 1 = 5$ ):

List	Action	Removal Count
1 2 3 4 5 <u>6</u> 7 8 9 10	original list	
1 2 3 4 5 <u>6</u> 7 8 9	remove last value	1
1 2 3 4 5 <u>6</u> 7 8	remove last value	2
1 2 3 4 5 <u>6</u> 7	remove last value	3
1 2 3 4 5 <u>6</u>	remove last value	4
1 2 3 4 5	remove last value	5

Notice how this process effectively discards the right half of the list.

All that's left to do is to recompute the length of the list and store that into the variable  $n$  – and that's it! To test the entire algorithm, first create a list of 10 or 20 values and randomly populate it with unique values using the appropriate script above. Then run the selection sort script to sort the list. Finally, assign a desired value to  $num$ , and run the binary search script.

Here's some sample output of this for *num*=74 on the list 14 27 28 34 39 41 52 61 64 74:

binary search

mid 0 num 74

array	
1	14
2	27
3	28
4	34
5	39
6	41
7	52
8	61
9	64
10	74

length: 10

binary search

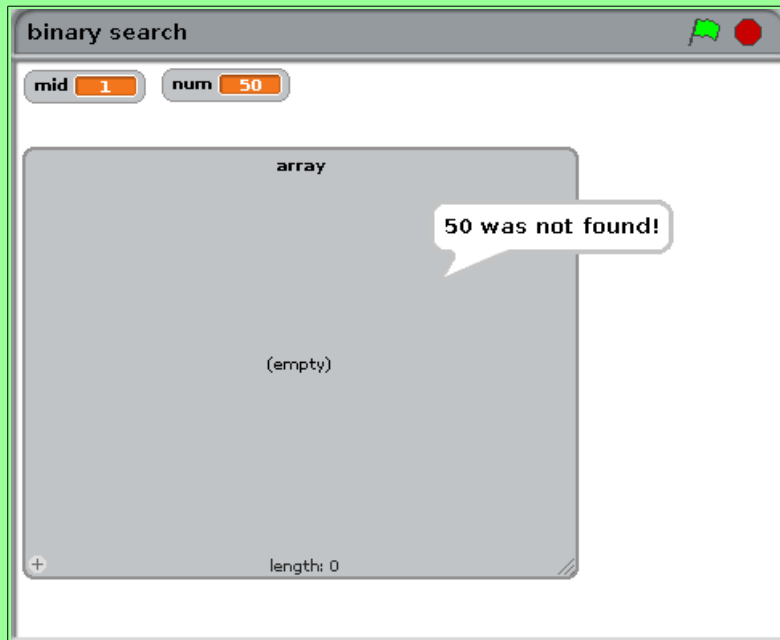
mid 1 num 74

array	
1	74

74 was found!

length: 1

And for  $num=50$  on the same list:



And finally for  $num=28$  on the same list:

