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.

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

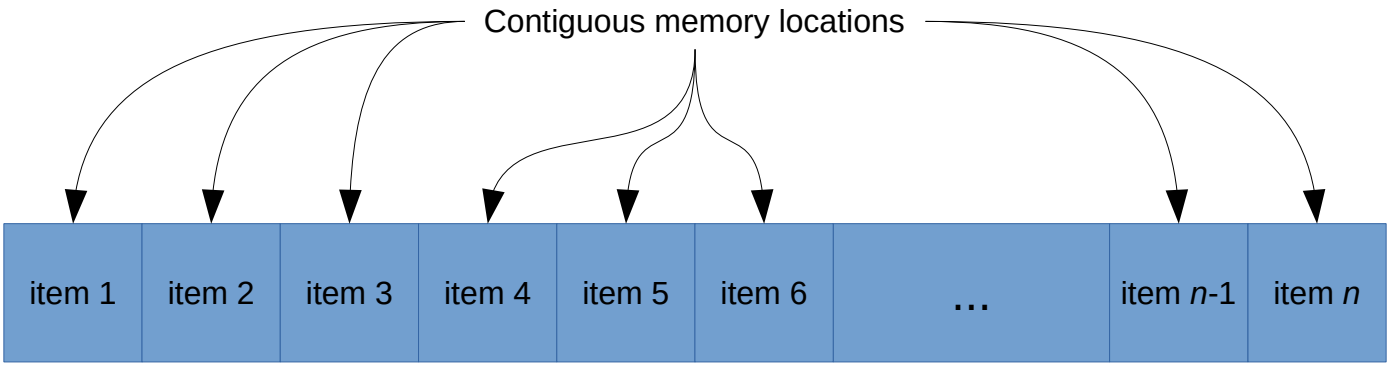
Data structures allow a programmer to arrange pieces of information (data) in a way that makes sense and allows the computer to manipulate the data easily for the programmer's task. Many times this data is made up of multiple instances of similar pieces of data, and other times it involves different kinds of data that are otherwise 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.

The 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*. We say that a *value* is stored at an *index* of the array.

The distinction between a **value** and its **index** is one that must be emphasized. In this context, the following definition holds:

Definition: *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. This means the **index** represents where an element is, whereas 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 arrays, it is important to understand the distinction between index and value.

The Python sequence

The most basic data structure in Python is the sequence. A **sequence** is composed of (related) elements. Each element in a sequence is assigned an index (or position). A sequence with n elements has indexes 0 to n–1. Pay special attention to this! The first value in a Python sequence is located at index 0, and the n^{th} (i.e., last) value is located at index n–1 . Python has many built-in types of sequences; however, the most popular is called the list. It effectively functions as an array!

The **list** in Python is quite versatile and is declared using square brackets; for example:

```
grades = [ 94, 78, 100, 86 ]
```

The statement above declares the list `grades` with four integers: 94, 78, 100, and 86. The list can be displayed in its entirety (e.g., with the statement `print(grades)`); however, we can access each

element individually by its index (specified within brackets). Accessing can mean to read a value in the list, or it can mean to change a value in the list; for example:

```
grades[0]
grades[3] = 87
```

Here's an example of this:

```
>>> grades = [ 94, 78, 100, 86 ]
>>> grades
[94, 78, 100, 86]
>>> grades[0]
94
>>> grades[3] = 87
>>> grades
[94, 78, 100, 87]
```

Note that, in Python, the values within a list do not need to be of the same data type! This is a bit different than lists in most other general purpose programming languages (usually, those languages call their lists **arrays**), in which all elements must be of the same type. Here's an example of a *heterogeneous* (meaning diverse) list in Python:

```
>>> stuff = [ 3.14, 2.78, 100, "100", "the speed of light!" ]
>>> stuff[0] = 3.14159
>>> stuff[4]
'the speed of light!'
>>> stuff[4] = "the speed of sound!"
>>> stuff
[3.14159, 2.78, 100, '100', 'the speed of sound!']
```

More than one value in a list can be accessed at a time. We can specify a range (or interval) of indexes in the format `[lower:upper+1]` which means the interval [lower, upper) (i.e., closed at lower and open at upper). That is, the lower index in the range is inclusive but the upper is not. For example:

```
stuff[3:4]      # accesses index 3 (the same as stuff[3])
stuff[0:5]      # accesses indexes 0 through 4
stuff[-3]       # accesses the third index from the right
```

Here are some examples:

```
>>> stuff = [ 3.14159, 2.78, 100, "100", "the speed of sound!" ]
>>> stuff[3:4]
['100']
>>> stuff[0:5]
[3.14159, 2.78, 100, '100', 'the speed of sound!']
>>> stuff[-3]
100
>>> stuff[-0]
3.14159
>>> stuff[-1]
'the speed of sound!'
>>> stuff[-2]
```

```
'100'
```

Note the difference between the element `100` (a number) at index 2 and the element `'100'` (a string) at index 3.

List elements can be deleted with the **`del`** keyword as follows:

```
>>> stuff = [ 3.14159, 2.78, 100, "100", "the speed of sound!" ]
>>> del stuff[0]
>>> stuff
[2.78, 100, '100', 'the speed of sound!']
>>> del stuff[2]
>>> stuff
[2.78, 100, 'the speed of sound!']
>>> del stuff[2]
>>> stuff
[2.78, 100]
```

Python provides several built-in operations that can be performed on lists. Here are many of them:

`len(list)`	Returns the length of a list
`max(list)`	Returns the item in the list with the maximum value
`min(list)`	Returns the item in the list with the minimum value
`list.append(item)`	Inserts item at the end of the list
`list.count(item)`	Returns the number of times an item appears in the list
`list.index(item)`	Returns the index of the first occurrence of item
`list.insert(index, item)`	Inserts an item at the specified index in the list
`list.remove(item)`	Removes the first occurrence of item from the list
`list.reverse()`	Reverses the items in the list
`list.sort()`	Sorts a list

Note that the word `list` is just a variable we defined to hold a list and is not a keyword nor anything built-into Python. If, for example, you call your list `numbers`, then you would write `numbers.sort()` to sort the values in your list, `numbers.append(item)` to append the contents of the variable `item` into your list, and `len(numbers)` to get the length of your `numbers` list.

```
>>> list = [ 13, 27, 4, 7, 2, 99, 46, 1 ]
>>> list
[13, 27, 4, 7, 2, 99, 46, 1]
>>> len(list)
8
>>> max(list)
99
>>> min(list)
1
>>> list.append(16)
```

```
>>> list
[13, 27, 4, 7, 2, 99, 46, 1, 16]
>>> list.count(7)
1
>>> list.insert(1, 7)
>>> list
[13, 7, 27, 4, 7, 2, 99, 46, 1, 16]
>>> list.count(7)
2
>>> list.index(7)
1
>>> list.remove(7)
>>> list
[13, 27, 4, 7, 2, 99, 46, 1, 16]
>>> list.sort()
>>> list
[1, 2, 4, 7, 13, 16, 27, 46, 99]
>>> list.reverse()
>>> list
[99, 46, 27, 16, 13, 7, 4, 2, 1]
```

Activity 1: Creating and populating an array (actually, a Python list)

In this activity, you will write a Python program to create a list of 20 random integers.

Creating the list

Create an empty list named *numbers* using a simple assignment statement:

```
numbers = []
```

Currently, the list is empty (i.e., its length is 0). In fact, this can be confirmed using one of the list methods shown above:

```
print(len(numbers))
```

Populating the list with random integers

Let's populate the list with 20 random integers. While it is possible (albeit tedious) to populate the list by adding each value individually, instead we are going to create a short program to make the population process automated.

Because the population process calls for a specific task (i.e., the addition of an item to the list) to be repeated over and over again, let's use the **while** loop with a counter (initialized to 0). Let's now declare the counter. Note that statements already shown above are included, and new statements are highlighted:

```
numbers = []
counter = 0
```

Because our task is going to be done 20 times (in order to fill the list with 20 values), we can structure the while loop as follows:

```
numbers = []
counter = 0
```

```
while (counter < 20):
    # add a random number to the list
    counter += 1
```

Note that the comment (beginning with #) will be replaced later. It is just informative at this point. Notice how the counter is incremented at the end of the while loop. This is necessary because it allows it to progress from 0 to 20, guaranteeing that the condition in the while loop will eventually be false (i.e., *counter* will be greater than or equal to 20). This stops the repetition, allowing control to continue past the while loop.

Now, what number will we be actually adding to the list? Well, we want to add 20 *randomly* selected numbers to the list. To generate random numbers, we can make use of a Python library called **random**; specifically, a function in the library called **randint**. Libraries and how to import them in Python programs will be discussed in more detail in a later lesson. For now, we'll simply show how to import the **randint** function from the **random** library:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    # add a random number to the list
    counter += 1
```

Next, let's generate a random integer from 1 to 99:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    num = randint(1, 99)
    counter += 1
```

Note that the **randint** function takes two parameters: a lower bound and an upper bound. The function returns a random integer from the lower bound to the upper bound, inclusive. So far, we've just generated 20 random numbers from 1 to 99. We're not actually adding them to the list! Let's do this now using the **append** list function shown above:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    num = randint(1, 99)
    numbers.append(num)
    counter += 1
```

You have now created a list (which we called *numbers*) and filled it with 20 random integers. If any of the following activities require a list filled with random numbers, you can easily refer to the above steps and create one. If you require a larger or smaller array, its just a matter of changing the value in the

while loop. If you require values in a different range (perhaps numbers between 100 and 1000), its just a matter of changing the values in the **randint** function.

The generated list can be displayed as follows:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    num = randint(1, 99)
    numbers.append(num)
    counter += 1

print(numbers)
```

The output generated will look something like this (of course, your list will look different):

```
[82, 17, 85, 8, 2, 7, 33, 13, 24, 89, 49, 37, 61, 72, 83, 39, 23, 58, 45, 31]
```

You may wonder why we store a randomly generated integer in the variable *num* in the program above. It's not used for anything other than being added to the list. In fact, the variable *num* is actually not necessary. The algorithm above can be modified without changing its behavior as follows:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    numbers.append(randint(1, 99))
    counter += 1
print(numbers)
```

The randomly generated integer is generated and immediately added to the list!

As a last observation, it is often good practice to use constants to specify target or desired values. For example, using a constant to specify the list's desired size can be useful. Any point in the program that needs the list's target size can refer to the constant. If it needs to be changed at a later time, the constant value simply needs to be changed, and all references to it through the constant won't have to also be updated. Here's a modification of the above program that implements this:

```
from random import randint

SIZE = 20
numbers = []
counter = 0
while (counter < SIZE):
    numbers.append(randint(1, 99))
    counter += 1

print(numbers)
```

Finally, recall that one of the list methods shown above returns the list's size. We can use it instead of a counter to repeatedly insert integers into the list until it has reached the desired size:

```
from random import randint

SIZE = 20
numbers = []
while (len(numbers) < SIZE):
    numbers.append(randint(1, 99))

print(numbers)
```

Note that all statements referring to the variable counter have been removed. As integers are inserted into the list, its length increases. Therefore, `len(numbers)`, initially 0, increases by one each time an integer is inserted into the list. Eventually, `len(numbers)` will be 20, and control will continue past the while loop.

The Python for loop

Python provides one more repetition construct called the **for** loop. Typically, while loops are considered to be sentinel-driven. That is, they require a condition to either be true or false that indicates execution of the statements in the loop. Recall that Scratch also had a **repeat-n** block that repeated a task a set (or fixed) number of times. This kind of loop can be implemented in Python using the for loop construct.

The structure of a for loop in Python is:

```
for iterating_variable in sequence:
    loop_body
```

Here's an example that uses the for loop on a list:

```
>>> list = [ 2, 4, 6, 8 ]
>>> for list_item in list:
        print("Current list item: {}".format(list_item))

Current list item: 2
Current list item: 4
Current list item: 6
Current list item: 8
```

The variable `list_item` is used as an iterator. That is, it takes on the value of each item in the list at each iteration of the for loop. The first time, `list_item` takes on the first item in the list, 2. The second time, it takes on the second item in the list, 4. Eventually, it takes on the last time in the list, 8. In total, the body of the for loop executes once for each item in the list (or four times).

The for loop is actually quite powerful and flexible. It can, for example, iterate through the letters of a string:

```
>>> for letter in "Hello world!":
        print("Current letter: {}".format(letter))
```

```
Current letter: H
Current letter: e
Current letter: l
Current letter: l
Current letter: o
Current letter:
Current letter: w
Current letter: o
Current letter: r
Current letter: l
Current letter: d
Current letter: !
```

Typically, for loops iterate through a counter. The counter can then be used to refer to a variety of things, including the index of the elements of a list. To structure a for loop such that it iterates, say from 0 through 8, we would use the built-in Python function, `range()`:

```
>>> for i in range(0, 9):
        print("Current iteration: {}".format(i))

Current iteration: 0
Current iteration: 1
Current iteration: 2
Current iteration: 3
Current iteration: 4
Current iteration: 5
Current iteration: 6
Current iteration: 7
Current iteration: 8
```

The `range()` function typically takes two parameters: a start value and a stop value. The start value is included in the range; however, the stop value is not. For example, to iterate from -10 to 10:

```
for i in range(-10, 11)
```

Note that an optional third parameter can be specified to indicate a step value. For example, suppose that you wish to iterate from 0 to 100 in increments of 10 (i.e., 0, 10, 20, …, 90, 100):

```
for i in range(0, 101, 10)
```

More on the for loop will be discussed later.

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

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 Python.

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 it the value in the first position of the list:

```
largest = numbers[0]
```

Now that we have initialized the variable **largest**, we need to compare it with each value in the list. This sounds like we'll need another loop construct (to do the comparison 20 times: one comparison for each of the values in the list). Technically, we really only need to do the comparison 19 times: one comparison for each of the remaining values in the list). Let's use a **for** loop for this.

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. We then keep on comparing it with the remaining values in the list.

To compare the variable **largest** with individual elements in the list, we will need to know the indexes of each of the values in the list. 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. The strategy will be to use a counter to iterate through the range of indexes in the list and update the variable largest as necessary. Recall that the first index in the list is 0. Since we assume that the largest value is currently there, we can begin searching at index 1:

```
largest = numbers[0]
for index in range(1, 20):
    if (numbers[index] > largest):
        largest = numbers[index]
```

The function **range(1, 20)** generates the sequence 1 through 19 that the variable **index** will iterate through (i.e., take on the value 1, followed by 2, and so on). We can therefore use the variable **index** to refer to specific locations in the list and access each element in the list. The benefit of using a variable instead of an explicit number is that a variable can change at any point in our algorithm.

The last step is to display the largest value:

```
largest = numbers[0]
for index in range(1, 20):
    if (numbers[index] > largest):
        largest = numbers[index]
print(largest)
```

Perhaps a more meaningful output statement is better:

```
largest = numbers[0]

for index in range(1, 20):
    if (numbers[index] > largest):
        largest = numbers[index]
```

```
print("The largest value is: {}".format(largest))
```

The entire program, including generating the list, is shown here for completeness:

```
from random import randint

numbers = []
while (len(numbers) < 20):
    numbers.append(randint(1, 99))

print(numbers)

largest = numbers[0]

for index in range(1, 20):
    if (numbers[index] > largest):
        largest = numbers[index]

print("The largest value is: {}".format(largest))
```

Activity 3: Selection sort of an array

For this activity, you will need a list of 20 random integers from 1 to 100. Refer to the Python code in the activities above if necessary.

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

```
 1: n ← length of the list
 2: for i ← 0..n-1
 3:     minPosition ← i
 4:     for j ← i+1..n
 5:         if item at j < item at minPosition
 6:         then
 7:             minPosition ← 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 selection sort in Python (using the list **numbers** generated above):

```
n = len(numbers)
```

```
    for i in range(0, n - 1):
        minPosition = i

        for j in range(i + 1, n):
            if (numbers[j] < numbers[minPosition]):
                minPosition = j

        temp = numbers[i]
        numbers[i] = numbers[minPosition]
        numbers[minPosition] = temp
```

Notice how the values at indexes *i* and *minPosition* of the list are swapped. A **temporary** variable is used to store on of the values, thereby allowing that value to be overwritten with the one to be swapped with. In other words, to swap two values, *x* and *y*, we temporarily store *x*, overwrite *x* with *y*, and overwrite *y* with the temporarily stored *x* (i.e., *temp* = *x*, *x* = *y*, *y* = *temp*).

Let's break the algorithm down, step-by-step. The first statement is pretty evident and is easily translated to Python: `n ← length of the list` becomes `n = len(numbers)`.

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 ← 0..n-1
 3:     minPosition ← i
 4:     for j ← i+1..n
 5:         if item at j < item at minPosition
 6:         then
 7:             minPosition ← j
 8:         end
 9:     next
10:     swap items at i and minPosition
11: next
```

It's very clear where both for loops are in the Python code:

```
    for i in range(0, n - 1):
        minPosition = i

        for j in range(i + 1, n):
            if (numbers[j] < numbers[minPosition]):
                minPosition = j

        temp = numbers[i]
        numbers[i] = numbers[minPosition]
```

```
        numbers[minPosition] = temp
```

In fact, the pseudocode and Python code are nearly identical! The initialization of *minPosition* to *i* is almost the same: `minPosition ← i` becomes `minPosition = i`.

The if statement in the inner loop looks slightly different; however, that's just because of how list items are accessed in Python: using an index enclosed in square brackets. The relevant pseudocode is:

```
5:  if item at j < item at minPosition
6:  then
7:      minPosition ← j
8:  end
```

And in Python:

```
if (numbers[j] < numbers[minPosition]):
    minPosition = j
```

As mentioned earlier, swapping is typically done by declaring a temporary variable, assigning it one of the two values to be swapped, and then performing successive assignments. Therefore, although the following pseudocode is nice and meaningful, it is not directly translatable to Python:

```
swap items at i and minPosition
```

Instead, we use successive assignment statements:

```
temp = numbers[i]
numbers[i] = numbers[minPosition]
numbers[minPosition] = temp
```

The selection sort algorithm in Python stores 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).

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 **numbers**), and that you have also sorted this list 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 ← number of items in the current portion of the list
3:     mid ← 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.

This new version works by finding the middle value in the list. It then compares the value to search for with this middle value. If they match, then the search is finished! Otherwise, the left or right half of the list is discarded, depending on the result of the comparison. This continues, either until the value is found, or until the entire list has been searched (and the value was not found).

Generally, it's not a good idea to actually remove values from a list when performing a search. We can tweak the algorithm a bit to maintain the list's integrity by keeping track of the "beginning" and "end" of the **valid** portion of the list. That is, if we wish to "discard" the left half of the list, then the valid portion of the list is the **right half**: the beginning of the valid portion of the list is at the index directly to the right of the middle value, and the end of the valid portion of the list doesn't change. And if we wish to "discard" the right half of the list, then the valid portion of the list is the **left half**: the beginning of

the valid portion of the list doesn't change, and the end of the valid portion of the list is at the index directly to the left of the middle value. Here's a modified algorithm that implements this tweak:

```
num ← number to search for
found ← false
first ← 0
last ← number of items in the list - 1
while first <= last and found != true
    mid ← floor((first + last) / 2)
    if num = item at mid of the list
    then
        found ← true
    else if num > item at mid of the list
    then
        first ← mid + 1
    else
        last ← mid - 1
    end
display found
```

If the desired value is found in the list, the search terminates. This occurs because the variable *found* is set to true when the value is found in the list, and the while loop's condition requires `found != false` to continue the repetition. The while loop also terminates if the variables *first* and *last* shift sides (i.e., *first* represents an index in the list that is greater then or equal to *last*). This indicates the the desired value was not found in the list.

Note how the variable *mid* is calculated: it is the average of the first and last indexes! The **floor** math function guarantees that the middle value is to the left of the center of the valid portion of the list, if the valid portion of the list has an **even** number of items.

Here's a Python implementation of the binary search:

```
num = int(input("What integer would you like to search for? "))
found = False
first = 0
last = len(numbers) - 1


while (first <= last and found != True):
    mid = (first + last) // 2
    if (num == numbers[mid]):
        found = True
    elif (num > numbers[mid]):
        first = mid + 1
    else:
        last = mid - 1

print(found)
```

First, the desired search value is obtained and stored into the variable *num*:

```
num = int(input("What integer would you like to search for? "))
```

Next, *found* is initialized to false, *first* to 0 (the first valid index in the list), and *last* to `len(numbers) - 1` (the last valid index in the list):

```
found = False
first = 0
last = len(numbers) - 1
```

The while loop is structured so that it iterates so long as `first <= last` (i.e., the beginning and end of the valid portion of the list haven't logically swapped) and `found != True` (i.e., the desired value hasn't already been found):

```
while (first <= last and found != True):
```

Next, the middle index of the valid position of the list is calculated as the average of *first* and *last*. Note the use of the `//` operator (which performs a floor division):

```
    mid = (first + last) // 2
```

If the desired value is found at the index stored in *mid*, the *found* is set to true (which will terminate the while loop). If not, then either *first* or *last* is appropriately updated: if the desired value is greater than *mid*, then *first* is updated to *mid* + 1; if the desired value is less than *mid*, then *last* is updated to *mid* – 1.

```
    if (num == numbers[mid]):
        found = True
    elif (num > numbers[mid]):
        first = mid + 1
    else:
        last = mid - 1
```

To assure yourself that the loop will terminate if the desired value is not found by logically swapping the positions of *first* and *last*, let's try a trivial example with a list of one item: `[ 5 ]`.

Running the binary search on this list initially sets *found* to false, *first* to 0, and *last* to 0 (`len(numbers) - 1 = 1 - 1 = 0`). At this point, *first* is indeed less than or equal to *last* (actually *first* == *last*). Moreover, *found* != true. Therefore, *mid* is calculated to be (0 + 0) // 2 = 0 // 2 = 0.
Suppose that we wish to search for the value 6 (which is not in the list). Therefore, *num* != *numbers*[*mid*]; however, *num* > *numbers*[*mid*] (i.e., 6 > 5). This results in *first* being updated to 1: *mid* + 1 = 0 + 1 = 1. When the condition in the while loop is reevaluated, *first* is 1 and *last* is 0! The values have logically swapped (i.e., how can *first* be greater than *last*?). Therefore, the while loop terminates, and the variable *found* is never being changed from its initialized value of false.

How could the algorithm be modified so that the output is more meaningful? That is, if the desired value is found, additionally display its index in the list. A possible solution is:

```
num = int(input("What integer would you like to search for? "))
found = False
first = 0
```

```
    last = len(numbers) - 1

    while (first <= last and found != True):
        mid = (first + last) // 2
        if (num == numbers[mid]):
            found = True
        elif (num > numbers[mid]):
            first = mid + 1
        else:
            last = mid - 1

    if (found):
        print("{} was found at index {}!".format(num, mid))
    else:
        print("{} was not found.".format(num))
```

The reason that *mid* can be used to provide the index of the desired (and found) value in the list, is that it is not recalculated when *found* is set to true. That is, its last calculated value is preserved (which is the last calculated middle of the valid portion of the list – where the desired value was found).