FACTFILE: GCSE DIGITAL TECHNOLOGY

Unit 4

DIGITAL AUTHORING CONCEPTS

Rewarding Learning

Data Programming Constructs 2

Learning Outcomes

Students should be able to:

Demonstrate understanding of and use the functionality of the following constructs in a programming language:

- simple sorting techniques such as the bubble sort and the insertion sort;
- simple searching techniques such as linear and binary searching; and
- string manipulating functions, including splitting, concatenating, character searching and substring searching.

Content

- Sorting Algorithms
  – Selection Sort
  – Insertion Sort
  – Bubble Sort
- Searching Algorithms
  – Linear Search
  – Binary Search
- Python Scripts
  – String Searching on Python
  – Bubble Sort in Python

Sorting Algorithms

Imagine that you have three, different sized blocks and that you are required to arrange them in order of size – small to large. You could probably do this without much thought, and with no particular method. If you had five or ten blocks instead of three it would still be a relatively straightforward task. But what if you had fifty, or a hundred, or five hundred blocks? With more blocks you would need to be more systematic in how you approach the task – you would need a *sorting algorithm*.

Sorting Algorithms

"[Sorting algorithms] are the processes implemented in a computer program to arrange the data in order."

BCS Glossary of Computing, 13th edition, p 346.

In computer systems we are concerned with sorting *data* rather than blocks. We might, for example, want to sort a collection of customer records by surname.

Surname	First Name	Telephone Number
Smith	Maria	02970814566
Jones	Rachel	02970814567
O'Hara	Robert	02970814568
McSherry	James	02970814569

Sort by surname

Surname	First Name	Telephone Number
Jones	Rachel	02970814567
McSherry	James	02970814569
O'Hara	Robert	02970814568
Smith	Maria	02970814566

Or we might, for example, want to sort a collection of medical records by patient age.

Surname	First Name	Age
Smith	Maria	49
Jones	Rachel	24
O'Hara	Robert	68
McSherry	James	53

Sort by age

Surname	First Name	Age
Jones	Rachel	24
Smith	Maria	49
McSherry	James	53
O'Hara	Robert	68

Data may be sorted in *ascending* order (low values to high) or *descending* order (high values to low). Fortunately, an algorithm for an ascending sort is easily adapted to perform a descending sort (and vice-versa).

Sorting and searching are very common and important operations in computing. Efficient database retrieval relies upon efficient *searching algorithms*, which, in turn, rely upon efficient *sorting algorithms*. Imagine, for example, searching for a telephone number in a telephone directory that was not in alphabetical order (i.e. had not been sorted).

Computer scientists have developed many different algorithms for sorting, some of which are explained below. In each case we assume that the data to be sorted is stored in an input array and the sorted data is returned in an output array.

Insertion Sort

The insertion sort algorithm works in the following way.

- Take the *first* item from the input array, and put it in the output array. At this stage it will be the only item in the output.
- Take the *second* item from the input array, and *insert* it into the output array. It must be inserted in the correct position to ensure that the output array remains sorted.
- Take the *third* item from the input array, and *insert* it into the output array. As before, the insertion operation must respect the sorted order of the output array.
- etc…until the input array is empty…

The example below shows the steps taken during the execution of an insertion sort.
The input array – [1 3 4 2 5] – is to be sorted in ascending order.

Input Array	Output Array	Comment
[1 3 4 2 5]	[ ]	Initially the input array is unsorted and the output array is empty.
[3 4 2 5]	[**1**]	The first item (1) is selected from the input and placed in the output.
[4 2 5]	[1 **3**]	The next item (3) is selected from the input and *inserted* into the output. The insertion operation places it *after* the 1.
[2 5]	[1 3 **4**]	The next item (4) is selected from the input and *inserted* into the output. The insertion operation places it *after* the 3.
[5]	[1 **2** 3 4]	The next item (2) is selected from the input and *inserted* into the output. The insertion operation places it *between* the 1 and the 3.
[ ]	[1 2 3 4 **5**]	The next item (5) is selected from the input and *inserted* into the output. The insertion operation places it *after* the 4. The input is now empty and the algorithm terminates. The output now contains the data sorted in ascending order, as required.

Bubble Sort

The bubble sort algorithm works in the following way.

- Scan the input array by comparing each pair of adjacent items.
  – Compare the first two items in the input array – swap them if they are not in the required order.
  – Compare the next pair of items in the input array – swap them if they are not in the required order.
  – etc...until you reach the end of the input array...
- Repeat this until a complete scan of the input array is completed, during which no items need to be swapped.

The example below shows the steps taken during the execution of an insertion sort.

The input array – [1 3 4 2 5] – is to be sorted in ascending order.

Input Array	Output Array	Comment
[1 3 4 2 5]		Initially the input array is unsorted. Bubble sort does not use a separate output array. It works by making changes to the input array. When execution terminates the input array will be sorted.
[**1 3** 4 2 5]		The first pair of items (1&3) are compared. As they are already in the correct order (i.e. 1<3) no changes are made.
[1 **3 4** 2 5]		The next pair of items (3&4) are compared. As they are already in the correct order (i.e. 3<4) no changes are made.
[1 3 **4 2** 5] Update: [1 3 **2 4** 5]		The next pair of items (4&2) are compared. This time the order is incorrect (i.e. 4>2) so the input array is updated.
[1 3 2 **4 5**]		The next pair of items (4&5) are compared. As they are already in the correct order (i.e. 4<5) no changes are made.
This completes the *first* scan or pass of the input array. As some items were swapped during this scan it is necessary to carry out another scan.		

[**1 3** 2 4 5]		The first pair of items (1&3) are compared. As they are already in the correct order (i.e. 1<3) no changes are made.
[1 **3 2** 4 5] Update: [1 **2 3** 4 5]		The next pair of items (3&2) are compared. This time the order is incorrect (i.e. 3>2) so the input array is updated.
[1 2 **3 4** 5]		The next pair of items (3&4) are compared. As they are already in the correct order (i.e. 3<4) no changes are made.
[1 2 3 **4 5**]		The next pair of items (4&5) are compared. As they are already in the correct order (i.e. 4<5) no changes are made.
This completes the *second* scan or pass of the input array. As some items were swapped during this scan it is necessary to carry out another scan.		
[**1 2** 3 4 5]		The first pair of items (1&2) are compared. As they are already in the correct order (i.e. 1<2) no changes are made.
[1 **2 3** 4 5]		The next pair of items (2&3) are compared. As they are already in the correct order (i.e. 2<3) no changes are made.
[1 2 **3 4** 5]		The next pair of items (3&4) are compared. As they are already in the correct order (i.e. 3<4) no changes are made.
[1 2 3 **4 5**]		The next pair of items (4&5) are compared. As they are already in the correct order (i.e. 4<5) no changes are made.
This completes the *third* scan or pass of the input array. As no items were swapped during this scan no further scans are required, and the algorithm terminates. The input array is now correctly sorted and is returned as the output.		

Here is an algorithm for bubble sort.

```
begin bubbleSort( array )
    swapped = True
        while swapped
            swapped = False
                for each item in array
                    if item > next item
                        then swap item with next item
                            and set swapped = True
            end for
        end while
    return = array
end bubbleSort
```

Searching Algorithms

The purpose of a search algorithm is to locate a specified item, which we call the *search term*, in a list or array. A search algorithm might be used, to find, for example, the telephone number for a given individual in a telephone directory. In this case we search for a name and then return the corresponding number. In the examples below we assume that we are searching an array of characters and return the corresponding array index – i.e. the position in the array where the search term is found. If the term is not in the array then the value –1 is returned.

Linear Search

One version of a linear search works in the following way:

- Compare the *first* item in the array with the search term. If they match, terminate and return the position in the array.
- Compare the *second* item in the array with the search term. If they match, terminate and return the position in the array.
- Compare the *third* item in the array with the search term. If they match, terminate and return the position in the array.

- etc…until you reach the end of the array…
- If you have reached the end of the array then return –1.

It is possible, however, to improve upon this algorithm providing we know that the array has been sorted. In the version above, which we will call *version 1*, we must continue to search until either the search term has been matched or else we have reached the end of the array. If we know, however, that the array is sorted we may be able to stop sooner than this.

- Let’s say we are searching for the character d in the array [a, b, c, e, g, k, m].
- Linear search will compare the search term (d) with the array item a, then b, then c, then e.
- At this stage we know that there is no point in searching further because if *d* was in the array it would have been before *e*.
- Consequently we do not need to continue to the end of the list.

A revised version of a linear search works like this.

- Compare the *first* item in the array with the search term.
  – If they match, terminate and return the position in the array.
  – If search item > array item, terminate and return –1
- Compare the *second* item in the array with the search term.
  – If they match, terminate and return the position in the array.
  – If search item > array item, terminate and return –1
- Compare the *third* item in the array with the search term.
  – If they match, terminate and return the position in the array.
  – If search item > array item, terminate and return –1
- etc…until you reach the end of the array…

Remember that this version of linear search, which we will call *version 2*, only works if the array is sorted. We say that version 2 of the algorithm is more *efficient* than the first, because it has the potential to solve the search problem in fewer steps.

Here is an algorithm for a linear search (version 1).

```
begin linearSearch (array, term)
    set i = 0
    set found = False
    while i < length of array AND
not found
        if ith array item = term
        then set found = True
        i = i + 1
    end while
    if not found i = -1
    return = i
end linearSearch
```

Exercise: Convert this algorithm for version 2 of the linear sort.

Binary Search

Binary search also requires the array to be sorted, but it is more efficient than either version of the linear search above. Instead of starting at the beginning of the array and working sequentially through it abinary search starts in the middle of the array. To see how this enables a binary search to be more efficient, imagine that we are searching an array of length 100, and consider what has been achieved after the first comparison has been made.

- Linear search starts by examining the first item in the array. If this doesn’t match the search term then the algorithm proceeds to the next item. At this stage the algorithm has, at most, a further 99 items left to process.
- Binary search starts by examining the 50th item in the array. This effectively divides the array into two halves: items 1–49 and items 51–100. If the selected item does not match the search term then, because the array is sorted, it is possible to identify one half of the array and say that it definitely does not contain the search term. At this stage the algorithm has, at most, a further 49 items left to process.

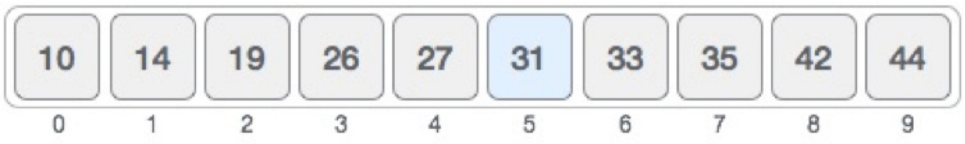
Binary search works like this.

- Compare the *middle* item in the array with the search term. This divides the array into two smaller arrays, which we will call *left* and *right*.
  – If the middle item matches the search term, terminate and return the position in the array.
  – If middle item < search term, do a binary search on the right array.
  – If middle term > search term, do a binary

search on the left array.

You can see that at each step the binary search algorithm reduces the effective size of the array to be processed by a factor of 0.5. This is why it is much more efficient than linear search. These last two processes are repeated until the search term can be found.

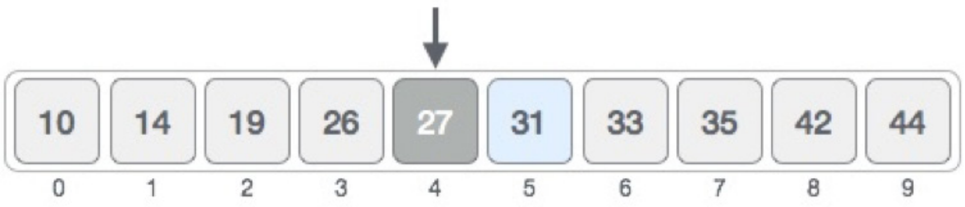
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



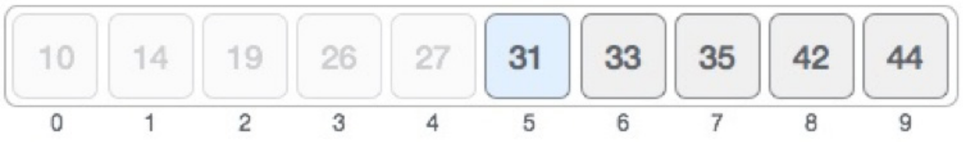
First, we shall determine half of the array by using this formula -

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 – 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



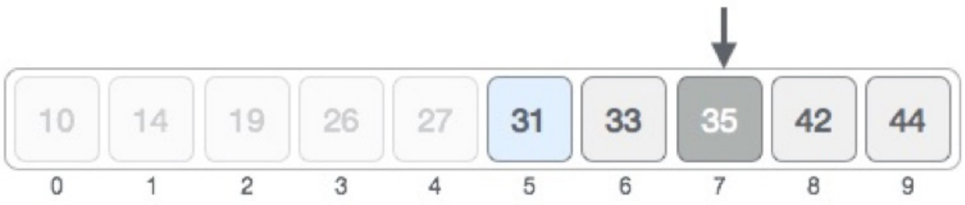
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, now we know that the target value must be in the upper portion of the array.



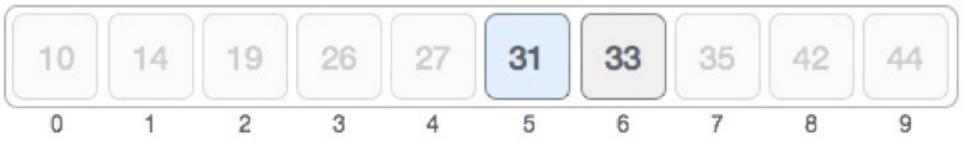
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

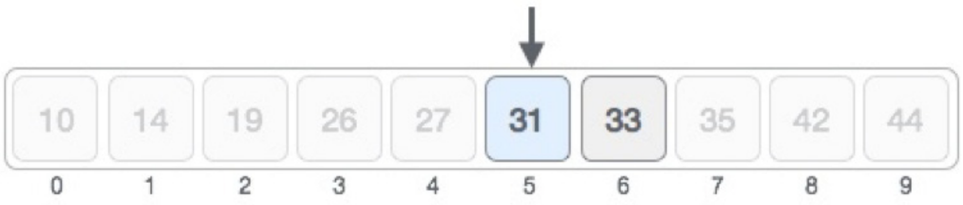
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



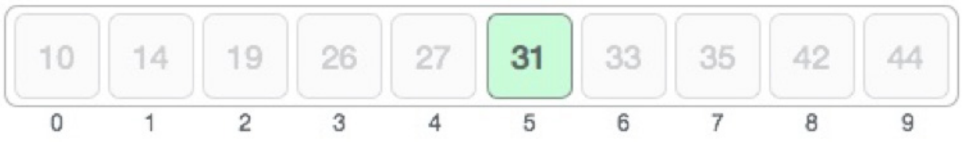
The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

```
begin binarySearch (array, arraySize, term )
   set lo = 0
   set hi = arraySize-1
   set found = FALSE
   repeat
      set mid = 0.5 * ( lo + hi )
      rounded to nearest whole number
      if array[mid] = term set found = TRUE
      if array[mid] < term set lo = mid + 1
      if array[mid] > term set hi = mid - 1
   until found OR hi < lo or
   if found
   then return = mid
   else return = -1
end binarySearch
```

Some more explanation will be helpful here.

- The variables *hi* and *lo* are used to store the upper and lower bounds of the array. As the algorithm converges on the solution the values are adjusted to focus on ever smaller sections of the array.
- The variable *mid* is the index to the middle item in the array. As the algorithm converges on the solution, its value is adjusted to correspond to smaller sections of the array.
- The variable *found* is a flag to indicate whether or not the search term has been found. It is initially set to FALSE, and then updated if and when the search term is found.
- The repeat loop terminates when either the search term is found or the condition *hi*<*lo* is met. If *hi*<*lo* this indicates that the array cannot be subdivided any further and consequently the

search term is not present.

Python Scripts

Further introductory material on the Python language, including string manipulation (U4FF3) and control structures (U4FF6), can be found in the following Fact Files.

- U4FF3: Digital Data
- U4FF5: Data Programming Constructs 1
- U4FF6: Data Programming Constructs 2

The scripts below provide further illustration of string manipulation in Python, and implement some of the sorting and searching algorithms developed above.

String Searching in Python

The following Python script implements both versions of our linear search algorithm, by searching for character *char* in string *string*. Recall that the algorithms differ in that version 2 is more efficient but it requires the input string to be sorted.

```
def linearSearch1( string, char ):
  i = 0
  found = False
  while i < len(string) and not found:
    if string[i] == char: found = True
    i = i + 1
  if not found: i = -1
  return(i)

def linearSearch2( string, char ):
  i = 0
  found = False
  while i < len(string) and not found
and string[i] <= char:
    if string[i] == char: found = True
    i = i + 1
  if not found: i = -1
  return(i)
```

The following interaction with Python shows version 1 of the algorithm in action.

```
>>> linearSearch1('bca', 'a')
3
>>> linearSearch1('bca', 'b')
1
>>> linearSearch1('bca', 'c')
2
>>> linearSearch1('bca', 'z')
-1
>>>
```

The following interactions with Python show version 2 of the algorithm in action.

```
>>> linearSearch2('bca', 'a')
-1
>>> linearSearch2('abc', 'a')
1
>>>
```

You will notice that the first function call to version 2 gives an incorrect response – this is because the string is not sorted.

Bubble Sort in Python

The following Python script implements the bubble sort algorithm.

```
def bubbleSort( array ):
  swapped = True
  while swapped:
    swapped = False
    for i in range(0,len(array)-1):
      if array[i]>array[i+1]:
        temp = array[i]
        array[i] = array[i+1]
        array[i+1] = temp
        swapped = True
return(array)
```

The following Python interactions illustrate this script in action.

```
>>> bubbleSort([1,2,3,5,4])
[1, 2, 3, 4, 5]
>>> bubbleSort([7,2,3,5,4])
[2, 3, 4, 5, 7]
>>>
bubbleSort(['q','w','e','r','t','y'])
['e', 'q', 'r', 't', 'w', 'y']
>>>
```

Resources

In addition to an excellent Python tutorial, there are very clear and accessible explanations and illustrations of sorting and searching algorithms at the TutorialsPoint website.

- TutorialsPoint, Python 3 Tutorial: https://www.tutorialspoint.com/python3/index.htm
- TutorialsPoint, Sorting Algorithms: https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm
- TutorialsPoint, Linear Search Algorithm: https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm
- TutorialsPoint, Binary Search Algorithm: https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm