GCST

2.1 ALGORITHMS

SEARCH & SORTING ALGORITHMS

GCSE OCR

code is written in OCR Exam Reference Language

GCST

LINEAR SEARCH

```
01 function LinearSearch(array, value)
02   index = 0
03   while index <= array.length - 1
04        if array[index] == value then
05             return true
06        endif
07        index = index + 1
08   endwhile
09   return false
10 endfunction
```

In this example, a function is created that takes two parameters:

- array: An array of items to be searched through.
- value: The item to be searched for.

How does it work?
A linear search is carried out by inspecting each item of the list in turn to check if it is the desired value. If so, we have found the item; if not, the next item in the list must be checked. If the algorithm gets to the end of the list without finding the item, then it is not in the list

- Line 01: The function is defined with two parameters.
- Line 02: A loop counter index is set to 0.
- Line 03: The WHILE loop starts at index 0, and repeats until the end of the list. Using array.length -1 allows any array size to be used.
- Line 04 compares the element in the array with the item we are searching for.
    - If the element matches the value then true is returned.
    - Because a return is used here, the function then terminates by default.
- Line 07: If there is no match, index increments by 1. The WHILE loop iterates (back to Line 03) using the new index value. The WHILE loop ceases to run if the end of the array is reached.
- Line 09: If the WHILE loop exits due to reaching the end of the array then it returns false.

GCST

BINARY SEARCH

pre-requisite is that the list/array must be sorted

```
01 function BinarySearch(array, value)
02      start = 0
03      end = array.length - 1
04      while start != end
05        midPoint = (start + end) DIV 2
06        if array[midPoint] < value then
07             start = midPoint + 1
08        elseif array[midPoint] > value then
09             end = midPoint - 1
10        elseif array[midPoint] == value then
11             return true
12        endif
13      endwhile
14      return false
15 endfunction
```

In this example, a function is created that takes two parameters:

- array: An array of items to be searched through.
- value: The item to be searched for.

How does it work?
The algorithm works by continually splitting the list in half until it finds the value it is looking for or has eliminated all the options.

- Line 02/03: An initial value for start and end are set. On the first pass, the start index is 0 and the end index is the final index in the array.
- Line 05 finds the middle of the array.
- Line 06 checks to see if the element in the array at the midpoint is greater than the item we are looking for.
    - If so, then Line 07 sets the index, one to the right of this position, as the new start.
- Line 08 checks to see if the element in the array at the midpoint is less than the item we are looking for.
    - If so, then Line 09 sets the index, one to the left of this position, as the new end.
- Line 10 checks if the item in the array and the item we are looking for are the same.
    - If so, then Line 11 returns true and the function exits.
- If either Line 06 or Line 08 are true, the while LOOP repeats with either the new start or end values.
- Line 14: By default, if the value is not found, false is returned.

GCST

BUBBLE SORT

Compare each item with the one next to it, if it is greater, swap them

9 5 10, 4, 15
5, 9, 10, 4, 15
5, 9, 10, 4, 15
5, 9, 4, 10, 15 1st pass

5, 9, 4, 10, 15
5, 4, 9, 10, 15 2nd pass

4, 5, 9, 10, 15 3rd pass

4, 5, 9, 10, 15 4th pass ✓

```
01 function BubbleSort(array)
02    sorted = false
03    while sorted == false
04          sorted = true
05          for i = 0 to array.length - 2
06                if array[i] > array[i + 1] then
07                temp = array [i]
08                    array[i] = array[i + 1]
09                    array[i + 1] = temp
10                sorted = false
11               endif
12          next i
13 endwhile
14 return array
15 endfunction
```

In this example, a function is created that takes one parameter:

- array: An array of items to be sorted.

How does it work?

The algorithm works by using a FOR loop, nested within a WHILE loop.

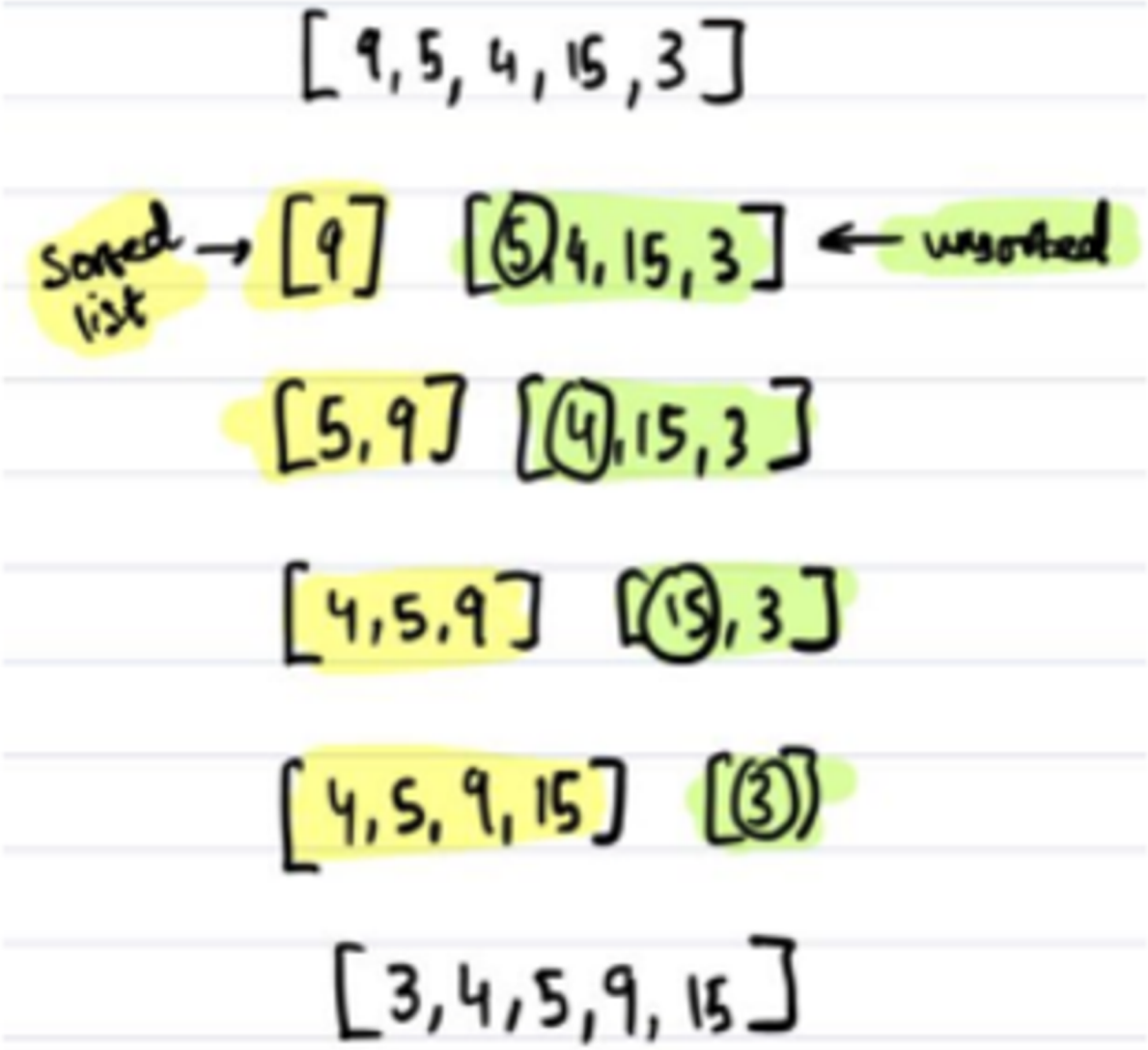
- The WHILE loop keeps running until the array is sorted.
- The FOR loop is used to sort the elements within the array.

- Line 01: The function is defined with one parameter.
- Line 02: We create a 'flag' and set it to false – assuming the array is not sorted at the start of the algorithm.
- Line 03: We start the WHILE loop, which keeps running whilst the array is not sorted (sorted == false).
- Line 04: We assume that this next iteration will sort the array.
- Line 05: We use a FOR loop to iterate through the array.
- Lines 06–09: If the current element is greater than the one to the right, then it swaps these elements around.
- Line 10: Because a swap was made, the array is 'unsorted' and so the flag is set back to false.
- Line 12: The index is increased by 1, i.e. we move to the next index and Lines 06–09 are repeated.
- Line 13: Once the FOR loop exits, the WHILE loop iterates.
- Line 03: The WHILE loop checks to see if sorted is false and executes again if so. If sorted is true, we know the array is sorted.
- Line 14: Once sorted == true, we return the sorted list.

GCST

INSERTION SORT

First item is in sorted list, the rest in unsorted, compare the first of the unsorted to the sorted and insert in correct position



```
01 function InsertionSort(array)
02   for i = 1 to array.length - 1
03        j = i
04        while j > 0 and array[j] < array[j - 1]
05             temp = array [j]
06            array[j] = array[j - 1]
07            array[j - 1] = temp
08            j = j - 1
09        endwhile
10   next i
11 return array
12 endfunction
```

In this example, a function is created that takes one parameter:

- array: An array of items to be sorted.

How does it work?
The algorithm works by using a FOR loop with a WHILE loop nested inside.

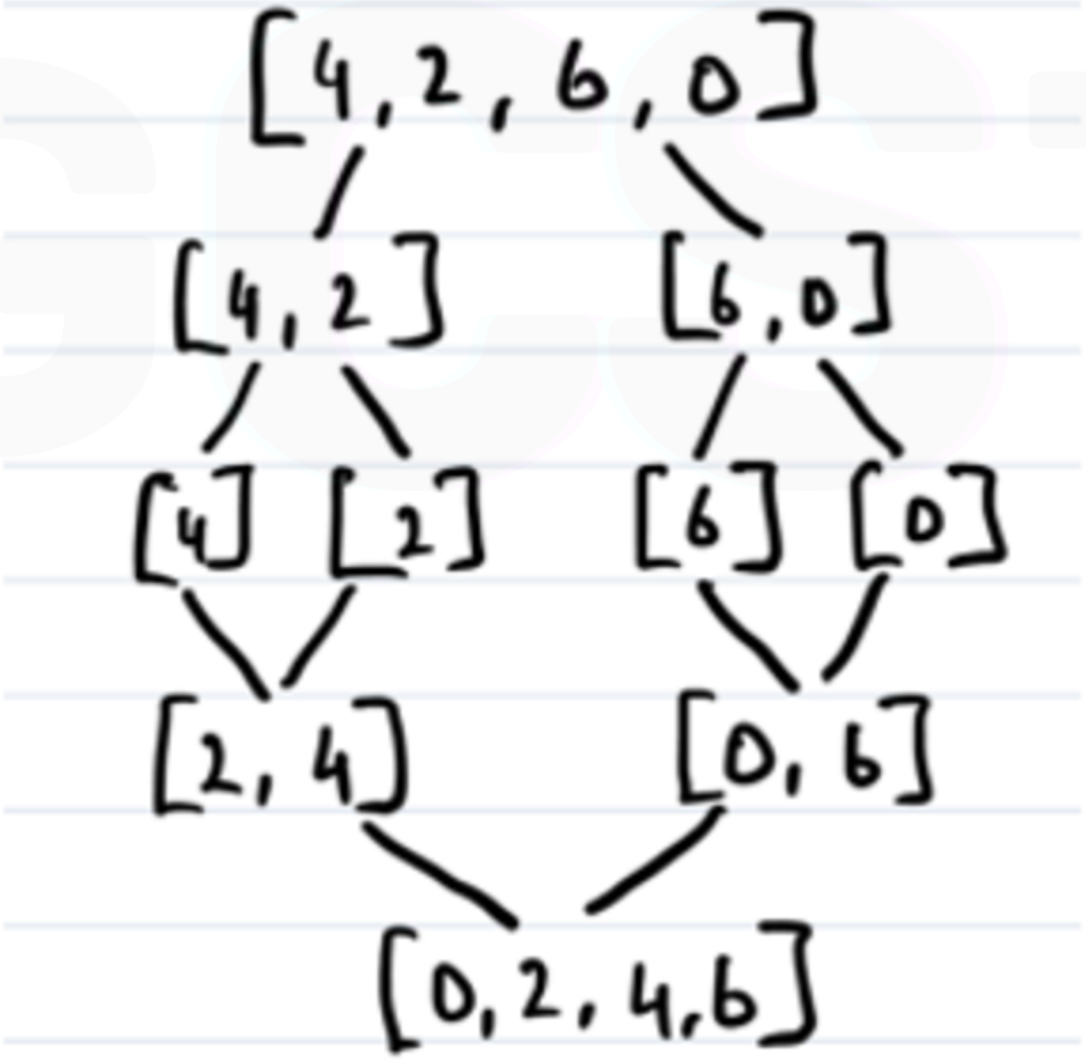
- The FOR loop is used to move element by element through the array from the second element to end.
- The WHILE loop checks to see if an element in the array is greater than the one to the left of it. It then repeats this working backwards through the list, until the element is in the correct place.
- Line 02/03: The FOR loop is created to run through the array. Each time i is assigned to the j variable. This means that later (line 8) we can decrement the j variable without affecting the FOR-loop pointer.
- Line 04: The WHILE loop checks to see if the value in the current index is less than the one to the left.
    - Line 05/06/07: If so, we swap the values.
    - Line 08: We now need to check that the swapped value is in the correct place in the sorted sub list, so we reverse down the array, checking and swapping if needed.
- Line 10: Once one element is sorted, we increase the FOR-loop counter to check the next element in the array.
- Line 11: Finally, we return the sorted array.

GCST

MERGE SORT

you do not need to know the code for merge sort

Divide the list into halves, till each list is one item long, then merge



GCST

COMPARISON OF THE SORTING ALGORITHMS

Bubble Sort:

- The bubble sort algorithm works by comparing pairs of values.
- If the two values are in the wrong order with respect to each other, they are swapped over.
- This is then repeated for each further pair of values. When the last pair of values has been compared, the first pass of the algorithm is complete.
- The algorithm will repeat until a pass has been completed with no swaps occurring.
- Once this happens, the list is guaranteed to be in order.
- After the first pass the highest element in the list bubbles towards the end

Pros

- Easy to implement
- Does not use much memory

Cons

- Poor for efficiency, especially if the list is scrambled up. It would take too many iterations to sort the list.

Insertion Sort:

- The insertion sort algorithm splits the list to be sorted in two parts: a sorted side and an unsorted side.
- Initially, the sorted side contains just the first item in a list, with everything else on the unsorted side.
- Each item on the unsorted side is then taken and inserted into the correct place on the sorted side, one by one.
- This process is repeated for the next item in the unsorted list till all items have been sorted
- Unlike a bubble sort, an insertion sort does not require multiple passes to check that the values are in order; once each value has been inserted into the sorted list and the unsorted list is empty, the list as a whole will be in order

Pros

- More efficient than bubble sort because fewer iterations are needed on average
- Little memory used
- Easy to implement

Cons

- Not very efficient compared to merge sort

GCST

Merge Sort:

- The merge sort algorithm uses a ‘divide and conquer’ approach to split data up into individual lists and then merge it back together in order.
- First, in the ‘divide’ stage, the original list is split into two separate sub lists. And each of those sublists are themselves each split into two sublists till each list Is one item long.
- Then each pair of lists are then merged together in the ‘conquer’ stage

Pros

- Very efficient compared with the rest, especially on large lists

Cons

- Inefficient for small lists or lists that are nearly sorted

To conclude, all of the above algorithms will result in a sorted list, but they will do it in very different ways. Bubble sort is generally thought of as a simple but slow algorithm; as the size of the list of values increases, it slows down significantly because it requires multiple passes over the same data. An insertion sort can be more efficient, but a merge sort is much more efficient than both of these for large lists of values. However, a merge sort may not be the best sorting method for nearly-sorted or small lists.

GCST

COMPARISON OF THE SEARCHING ALGORITHMS

Linear Search

- A linear search is carried out by inspecting each item of the list in turn to check if it is the desired value. If so, we have found the item; if not, the next item in the list must be checked. If the algorithm gets to the end of the list without finding the item, then it is not in the list.

Pros

- Easy to implement as it can work on any list (doesn't have to be ordered)

Cons

- Every single value in the list needs to be checked before you can be certain that a value is not present in a list which is time consuming.

Binary Search

- The middle value in the sorted list is picked. If there are an even number of values the value to the left of the middle is chosen.
- If the middle value is the one we are searching for then the algorithm finishes.
- However, if not, we can discard the bottom half of the list if the middle value is smaller than the one we are searching for, or discard the top half of the list if the middle number is larger than the one we are searching for.
- Either way, we always discard the middle value.
- If we get to a situation where the list only has one item and it is not the one that we are searching for, then the value is not in the list.

Pros

- Binary search is highly efficient.
- If an ordered list of one million numbers is used, the binary search could find a number in the list with no more than 21 comparisons. Linear Search would take up to one million comparisons lol.

Cons

- Only works for a sorted list. Therefore, it cannot always be used

If you found this useful, drop a follow to help me out!

THANK YOU!

GCST

GCSECOMPUTERSCIENCETUTOR.COM