Unit 2 | 2.3.1 | Algorithms

Specification & learning objectives

	A Level	Specification point description
	2.3.1a	Analysis and design of algorithms for a given situation
	2.3.1f	Standard algorithms (bubble sort, insertion sort, binary search and linear search)
	2.3.1f	Implement bubble sort, insertion sort
	2.3.1f	Implement binary and linear search
	2.3.1e	Representing, adding data to and removing data from queues and stacks
	2.3.1b	Compare the suitability of different algorithms for a given task and data set

Resources

PG Online textbook page ref: 184-188,200-203,334-337,340-344
Hodder textbook page ref: 49-59
CraignDave videos for SLR 25

Key question: How does the bubble sort work?

1. Set a pointer to the start of the list
2. Take the first pair of items in the list and compare them
3. If the first is bigger than the second, swap them
4. Now increment the pair pointer and repeat the compare, swap action
5. Repeat the action through the list and start again from the first pair
6. Stop when no swaps have occurred over the whole list

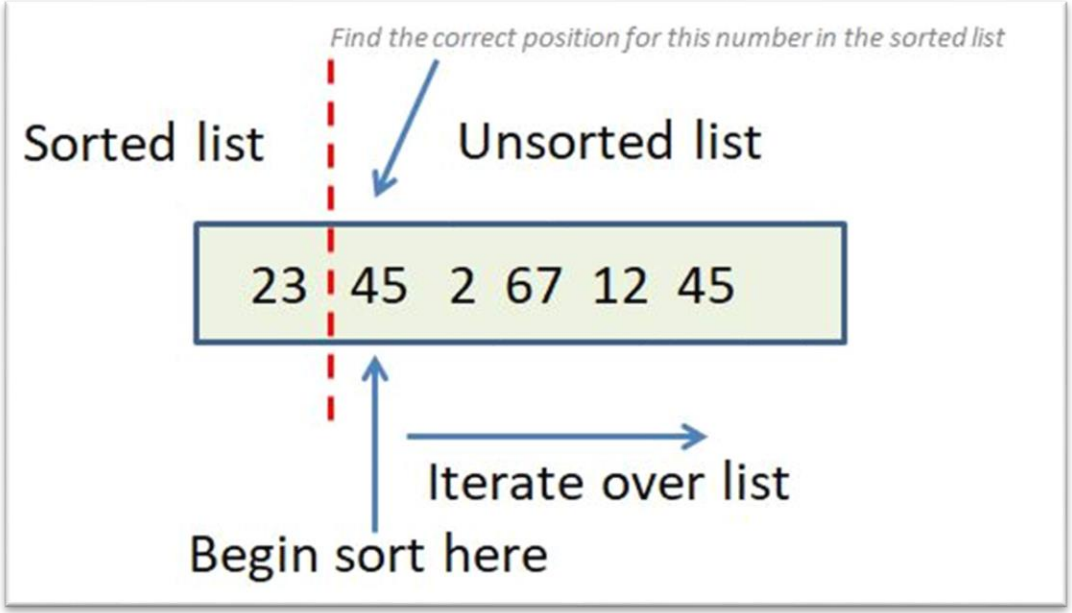
```
item = a defined array or list of values
Repeat

   i = start position of the array called item
   swapsmade = false
   Repeat
      if item(i) > item(i+1) then
         swap values
         swapsmade = true
      end if
      increment i

  Until end of the array

Until swapsmade = false
```

Key question: How does the insertion sort work?




```
Let A be an unsorted list
pointer = 1   # second position in the list
while pointer < length(A) # iterate over the list
     p = A[pointer]  # temporary copy of current value
     q = pointer - 1  # point one to the left of current
     while A[q] >= 0 AND A[q] > p
           A[q+1] = A[q] # push up elements by one
           q = q - 1 # move down the sorted list
     end while
     A[q+1] = p # insert operation
     pointer = pointer + 1 # iterate up the list
end while
```

Key question: How does the linear search work?

For a list of ***n*** items

The algorithm is:

1. Specify the item to be searched **T**

2. Set a list pointer to its beginning i = 0

2. Compare **T** with the first element in the list (L_0) as the first element is position zero

3 If $T = L_0$ then stop search and return 0

4. Else increment i

5. Compare to next element L_i

6. If $T = L_i$ then stop search and return i

7. Repeat until the end of the list L_{n-1} is reached and return a 'not found' signal

Key question: How does the binary search work?

List has to be in order (smallest to largest)

1. Let the item to be searched for be T
2. Examine the midmost item in the list to see if it is T
3. If it was, then it has been found and return its position
4. If it is higher than the middle,
    1. then examine the next one up.
    2. Adjust the bottom of the list to be this one
    3. If it not T then examine the next element and adjust the bottom of the list so the list is becoming shorter each time
    4. Repeat until found or the end of list is reached and return as not found
5. If it is lower than the middle
    1. Examine the next one down and adjust the top of the list to be this item
    2. If it is not this one, then examine the next lower one and again adjust the top each time
    3. Repeat until it is found or the beginning of the list is reached and it is not found

Comparisons

Bubble sort

Algorithm	Best case	Worst case	Average case	Space complexity
Bubble sort	$O(n)$ comparisons $O(1)$ swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	$O(1)$

Insertion sort

Algorithm	Best case	Worst case	Average case	Space complexity
Insertion sort	$O(n)$ comparisons $O(1)$ swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	$O(1)$

Linear Search

Algorithm	Best case	Worst case	Average case	Space complexity
Linear search	1	$O(n)$	$O(\frac{(N+1)}{2})$	$O(1)$

Binary Search

Algorithm	Best case	Worst case	Average case	Space complexity
Binary search	O(1)	$O(log_2\ n)$	$O(log_2\ n)$	$O(1)$

Typical exam questions

Consider this pseudocode:

```
Do
   swap = false
   For position = 0 to listlength-2
      If list[position]>list[position+1] then
         list.swap[position,position+1]
         swap = false
      EndIf
   Next
Until swap = false
```

1. The code contains a logic error. Explain what the mistake is. **[1]**

swap = false inside the condition should be swap = true

2. Identify the name of the algorithm. **[1]**

Bubble sort

3. Explain why this algorithm is inefficient. **[2]**

It requires significantly more swaps than other algorithms.
It takes more CPU cycles to solve than other algorithms performing the same sort.
It will require $O(n^2)$ swaps in the worst-case.

Typical exam questions

4. Explain an alternative approach to the algorithm. **[6]**

Insertion sort.
Has a sorted and unsorted list.
First item from the unsorted list is compared to the last item in the sorted list.
If the new item comes before the last item in the sorted list it is shuffled left…
…until it's place is found…
…and becomes a new item in the sorted list.
Requires fewer swaps than a bubble sort.

Unit 2 | 2.3.1 | Algorithms

Target: Overall grade:

Minimum expectations & learning outcomes

☐	Terms 234, 236, 237, 242, 432 from your A Level Key Terminology should be included and formatted.
☐	You must provide a worked example and pseudocode for each of the following algorithms and data structures: Bubble sort & Insertion Sort.
☐	Binary Search & Linear Search.
☐	Queue & Stack.
☐	Answer the exam questions.

Feedback

Breadth	Depth	Presentation	Understanding
☐ All	☐ Analysed	☐ Excellent	☐ Excellent
☐ Most	☐ Explained	☐ Good	☐ Good
☐ Some	☐ Described	☐ Fair	☐ Fair
☐ Few	☐ Identified	☐ Poor	☐ Poor

Comment & action required

Unit 2 | 2.3.1 | Algorithms

Reflection & Revision checklist

Confidence	Clarification
☹ 😐 ☺	Candidates need to be able to write algorithms using flow charts, pseudocode and program code.
☹ 😐 ☺	Candidates need to be able to follow the code as shown in the OCR pseudocode guide but are not expected to write code in this syntax.
☹ 😐 ☺	Candidate's code is not expected to be syntactically correct but must use appropriate code structures.
☹ 😐 ☺	Candidates need to understand the need for standard sorting algorithms.
☹ 😐 ☺	Candidates need to understand how the sorting algorithms bubble and insertion work and the situations when each can and cannot be used.
☹ 😐 ☺	Candidates need to be able to use the algorithms to sort data, and complete, write and correct algorithms to perform each sorting algorithm.
☹ 😐 ☺	Candidates need to understand the need for standard searching algorithms.
☹ 😐 ☺	Candidates need to understand how the searching algorithms binary and linear work and the situations when each can and cannot be used.
☹ 😐 ☺	Candidates need to be able to use the algorithms to search data sets for specific values that may or may not exist in the data set.
☹ 😐 ☺	Candidates need to understand when each searching algorithm can and cannot be used.
☹ 😐 ☺	Candidates need to be able to complete, write and correct algorithms to perform each searching algorithm.
☹ 😐 ☺	Candidates should have experience of using the data structures stacks and queues.
☹ 😐 ☺	Candidates need to understand the differences and similarities between stacks and queues.
☹ 😐 ☺	Candidates need to be able to add and remove data from both stacks and queues.
☹ 😐 ☺	Candidates need to understand how pointers are used within stacks and queues.
☹ 😐 ☺	Candidates need to understand how stacks and queues can be implemented in a computer system, for example through the use of an array with pointers.
☹ 😐 ☺	Candidates need to be able to read, correct and write algorithms to add and remove data items, and manipulate data items in a stack and queue.
☹ 😐 ☺	Candidates need to understand how the choice of algorithm can be affected by the data set.
☹ 😐 ☺	Candidates need to understand the impact of specific algorithms on speed and memory use.

Unit 2 | 2.3.1 | Algorithms

Reflection & Revision checklist

Confidence	Clarification
☹ 😐 ☺	Candidates are not expected to know about Big O notation, but should be aware of how and when a program can use more memory, or can take longer to run and be able to compare algorithms to determine which will use more/less memory, and which will run faster/slower.