OCR Computer Science AS Level

2.3.1 Sorting Algorithms
Advanced Notes

Specification:

- Standard algorithms
  - Bubble sort
  - Insertion sort

Sorting Algorithms

Sorting algorithms are designed to take a number of elements in any order and output them in a logical order. This is usually numerical or lexicographic (phonebook style ordering).

Most sorting algorithms will output elements in ascending order, but the algorithms can typically be slightly altered (for example, by switching an inequality from less than to greater than) or their output simply reversed in order to produce an output in descending order.

Bubble Sort
Bubble sort makes comparisons and swaps between pairs of elements. The largest element in the unsorted part of the input is said to "bubble" to the top of the data with each iteration of the algorithm.

The algorithm starts at the first element in an array and compares it to the second. If they are in the wrong order, the algorithm swaps the pair. Otherwise, the algorithm moves on. The process is then repeated for every adjacent pair of elements in the array until the end of the array is reached (at which point the largest element is in the last position of the array).

This is referred to as one pass of the algorithm. For an array with n elements, the algorithm will perform n passes through the data, at which point the input is sorted and can be returned.

```
A = Array of data

    for i = 0 to A.length - 1:
        for j = 0 to A.length - 2:
            if A[j] > A[j+1]:
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp
            endif
    return A
```

The three lines highlighted could be written as "swap A[j] and A[j+1]" and indeed in most situations this would be acceptable. The example illustrates how a swap operation would work, making use of a temporary store.

Example

Use bubble sort to arrange the elements in the array into ascending order.

4	9	1	6	7

The two first elements are compared. They are in the correct order and so the algorithm moves on.

4	9	1	6	7
4	1	9	6	7

The second two elements are in the wrong order, and so they are swapped.

4	1	9	6	7
4	1	6	9	7

Again, the next two elements are in the wrong order, and so they are swapped.

4	1	6	9	7
4	1	6	7	9

The last two elements are also in the wrong order, so are swapped.

This marks the end of the first pass of the algorithm.

The second pass starts at the beginning of the array and compares the first two elements. They are in the wrong order, so are swapped.
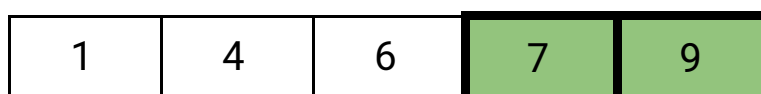
4	1	6	7	9
1	4	6	7	9

The next two elements are in the correct order, so the algorithm moves on.

1	4	6	7	9

Again the comparison reveals that the elements are in the correct order.

1	4	6	7	9

The last pair are also in the correct order.

1	4	6	7	9

This is the end of the second pass, and we can see that the data is sorted. However, the algorithm wouldn't terminate here. It would carry out another two passes before terminating.

You might be thinking that this sounds like a waste of time, and you'd be right. If the algorithm goes on until it terminates, it could possibly make many pointless comparisons. The bubble sort algorithm can be improved to solve this issue, as the pseudocode below shows.

```
A = Array of data

    for i = 0 to A.length - 1:
        noSwap = True
        for j = 0 to A.length - (j + 2):
            if A[j] > A[j+1]:
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp
                noSwap = False
        if noSwap:
            break
    return A
```

The most important modification is the introduction of a flag recording whether a swap has occurred. If a full pass is made without any swaps, then the algorithm terminates earlier than it would have done originally.

The second, less significant, change is the inclusion of j in the second for loop. This means that as j increases, fewer comparisons are made. This works because the largest element is always in place at the end of the first pass, and the second largest element is in place at the end of the second pass and so on. There is no need to check these elements as we know they're in the right place.

Make sure you aren't caught out when using this version of the algorithm though. The algorithm terminates after a complete pass in which no swaps are completed, not at the end of the first pass at the end of which the elements are sorted!

Bubble sort is a fairly slow sorting algorithm, with a time complexity of $O(n^2)$.

Insertion Sort
Another sorting algorithm, insertion sort places elements into a sorted sequence. In the ith iteration of the algorithm, the first i elements of the array are sorted. Be careful though, although i elements are sorted, they are not the i smallest elements in the input!

For example, the first three elements in the sequence 3, 7, 9, 4, 8 are sorted, but are not the three smallest elements in the input.

Insertion sort starts at the second element in the input, and compares it to the element to its left. If the two elements are in the wrong order, the smaller element is placed in the lowest position.

The third element in the input is then selected. It is inserted into the correct position in the sorted portion of the input to its left. This continues until the last element is inserted into the correct position, resulting in a fully sorted array.

```
A = Array of data

for i = 1 to A.length - 1:
    elem = A[i]
    j = i - 1
    while j > 0 and A[j] > elem:
        A[j+1] = A[j]
        j = j - 1
    A[j+1] = elem
```

The pseudocode for insertion sort, shown above, shows how the algorithm starts at the second item and places it into a sorted sequence by performing consecutive swaps (within

the while loop) until every item has been inserted, leaving behind a sorted array. Insertion sort has the same time complexity as bubble sort, $0(n^2)$.

Example

The algorithm can ignore the first element, as a single element is always a sorted sequence. Starting at the second element, we see it is greater than the element to its left and so it remains in place.

4	9	1	6	7

Moving to the third element, which is smaller than the element to its left, the algorithm has to move it into the right place in the sorted sequence that is forming at the start of the data. It moves the element one place to the left, but it is still smaller than the new element to its left, so is moved left again at which point it is in the correct place in the sorted sequence.

4	9	1	6	7
4	1	9	6	7
1	4	9	6	7

The next element to be sorted is also smaller than the element to its left, so is moved left.

1	4	9	6	7
1	4	6	9	7

The same occurs with the next element, which is also moved left into the correct place.

1	4	6	9	7
1	4	6	7	9

Finally, the last element is checked, and is already in the right place.

1	4	6	7	9

1	4	6	7	9