Computational Methods Assignment

Task 2

Bubble Sort										
PASS 0	10	15	12	12	25	16	20	14	28	17
PASS 1	10	12	12	15	16	20	14	25	17	28
PASS 2	10	12	12	15	16	14	17	20	25	28
PASS 3	10	12	12	15	14	16	17	20	25	28
PASS 4	10	12	12	14	15	16	17	20	25	28

In the first pass, we swap 12 and 15 twice, then swap 16 and 25, then 20 and 25, then 14 and 25 and finally swap 17 and 25.
In the second pass we swap 14 and 20, 17 and 25.
In the third pass we swap 17 and 20.
In the fourth pass we swap the 14 and 16
In the fifth pass we swap the 14 and 15.

To demonstrate how BubbleSort works I have recreated the sorting algorithm in both C++ and PsuedoCode

C++

```
1.  int main()
2.  {
3.      int arr[] = { 10, 15, 12, 12, 25, 16, 20, 14, 28, 17 };
4.      int size = sizeof(arr) / sizeof(arr[0]);
5.
6.      for (int i = 1; i < size; i++)
7.      {
8.          for (int j = 0; j < size - 1; j++)
9.          {
10.             if (arr[j] > arr[j + 1])
11.             {
12.                 int temp = arr[j + 1];
13.                 arr[j + 1] = arr[j];
14.                 arr[j] = temp;
15.             }
16.         }
17.     }
18. }
```

PsuedoCode

```
1.  function bubbleSort()
2.  {
3.      int array[] = 10, 15, 12, 12, 25, 16, 20, 14, 28, 17
4.      for i = 1 to array.size
5.          for j = 0 to array.size - 1
6.              if(array[j] > array[j + 1]
7.                  int temp = array[j + 1]
8.                  array[j + 1] = array[j]
9.                  array[j] = temp
10.         endfor
11.     endfor
12. }
```

Bubble sort is a simple sorting algorithm where you repeatedly step through an array, comparing the next element with itself and swapping them if they are in the wrong order. This is repeated until no more swaps are needed and the list is sorted.

Worst Case Time Complexity:
In the worst case each element needs to compare itself with its adjacent neighbours and swap itself multiple times. Which gives you worst case time complexity $O(n^2)$.

Best Case Time Complexity:
In the best case the list would already be sorted, the algorithm would make one pass over the array comparing each number but making no swaps. Which gives you best case time complexity O(n).

Average Case Time Complexity
In the average the algorithm would have to perform roughly half the comparisons from worst case. Which still gives you $O(n^2)$ as the comparisons and swaps increased exponentially with the size of the list.