**IT6033 Practical Task 2: Implement Sorting Algorithms.**

# Part 1. Bubble Sort:

Fully implemented working code to be assessed.

not ideal for sorting through large datasets but simple and effective. Can be easily implemented to get some code working in early stages of development.

**Best case:** O(n) (if checking the arrangement to check if already sorted, a bubble sort algorithm can cover a sorted array in one pass).
**Average case**: O(n/2 x n)
**Worst Case:** O(n2)

# Part 2. Quick Sort:

Demonstrate the efficiency of an algorithm.

## Counters Used:

**CountComparisons:**
increments each time a comparison is made to check if an assessed value is less or greater than the pivot.

**CountExchanges.**
Increments each time array values are exchanged.
**note:** Due to the way this quicksort is designed, at least one exchange will happen when QuickSort() is called plus and an additional exchange for each recursion, because when i == j there is an exchange even though there is no impact on the array arrangement.

## Timers:

**TimerSort**
each time the Sort is called, A timer will start to keep track of how much time it takes to complete that Sort call.

**Average:**
Each time a timer is complete it is added to the TimerList so that the running average can be calculated at any point. By dividing the sum of the timers by the count.

**TimerFull**
An additional timer has been added in the Main() section to time the entire exercise.
(this will be greater than the sum of sortTimer's because it tracks All the code)

**Analysis:**

**Best and Worst cases:**

Timer results are printed in the Console and demonstrate that organizing an un-sorted array with QuickSort takes far longer than sorting a pre-sorted or a pre-sorted-reversed array.

**The best-case** scenario is clearly when the array is already sorted as the least amount of exchanges are required. Also the time to sort an ordered-reversed array using quick sort is not much slower than the best-case scenario because each exchange puts the items in question directly in the correct position in one iteration.
**O(log-n) Logarithmic-Time** is the best case/average time Big O notation for quick sort. Each Iteration has the potential to discard huge chunks of data and only sorts through what is in the remaining defined range. In the best case, the database gets split evenly (in half) with each recursion.

**The Worse-case (in this exercise)** is when the array is unsorted because a few of the elements can each require multiple exchanges before they are set into the correct position.

**The Worst-case O(n2) Quadratic-time** the worst possible case for a quicksort is when the pivot is consistently at the extreme (smallest or largest element) end of the of the array. This happens usually when the array is sorted and either the first or last element is consistently picked as the pivot.

**Which Algorithm to use?** (from the 3 used in this exercise)
Depending on the application, usually Quicksort would be the Algorithm of choice provided that it is not essential to preserve the order of items that contain equal values.

**QuickSort:**
The name gives is away as this uses the divide and conquer strategy that is effective at breaking down large tasks into small simple steps almost wherever applied. By comparing the number of comparisons and swaps with bubble sort, QuickSort is the clear winner.
One of the main concerns with Quicksort is that it is not considered a stable algorithm because it can swap 2 elements of the same value.
**Best case:** 2 or 3 times faster than merge or heap sort
**Average case**: O(log n)
**Worst Case:** O(n2) - when implemented very badly.

**Built in sort:**
the easiest to implement but as seen by the Timer output, this sort takes up more computing power than the quicksort by a factor of about 30.
Very easy to implement but not very efficient

# Part 3. Built in Sort:

The use of OrderBy() => combined with a lambda allowed me to return the sorted array using one line of code.

I implemented a timer to return the results which I found returned between 90,000 to 125,000 ticks to sort the array.