## Task 4: Comparing bubble sort and quick sort

The bubble sort is a well-established and relatively simple algorithm for sorting data by repeated comparisons of adjacent elements in an array. ‘Quick sort makes use of a technique called recursion, splitting an array in half again and again until each array only has one element in it, and then ‘unwinding’. Recursion is a programming technique where a module (a group of instructions referred to by name) calls itself, the most important aspect of which is to ensure that it ‘unwinds’ or at least stops. Failure to correctly stop recursion can lead to stack overflow, the mechanisms of which you can investigate at your leisure. This makes it efficient for large data sets, but less so for smaller ones, since there is a large programming overhead.

In this task you will apply both sorting techniques on large integer arrays.

The program steps are:

* Create a large array filled with random integers.
* Record the current time as precisely as possible using code.
* Run the algorithm (don’t display before capturing the end time!).
* Capture the end time and subtract the start time to give the elapsed time.
* Display the results and the elapsed time.

Your program will need to provide the following features and selections:

* it must be able to select the size of the array
* it must be able to select which algorithm to use
* it must be easily resettable to allow re-running with different parameters.

Gather and record results for both algorithms with array sizes of 100, 1000, 10000 and 100000 integers. Following is a suggested representation of the Graphical User Interface (GUI). If you are working without a GUI, such as in Python, you will need to prompt the user for the data and print out the sorted results, if possible.



The code for each algorithm and the random integer generating code **must** each be in their own separate function/method.

' Quick sort algorithm (divide and conquere in-place) recursively

' 1. Choose any element in the array array[p..r]. Call this element the pivot q

' 2. Rearrange the elements in array[p..r] so that all elements in array[p..r]

' that are less than Or equal (<=) To the pivot q are To its left And all

' elements that are greater (<) than the pivot q are To its right

' 3. call quicksort on subarray array[p..q-1] and call quicksort on subarray

' array[q+1..r] Note item q Is Not part of either of these subarrays

' 4. these subarrays are then joined together recursively as each call

' to quicksort ends

Private Sub quickSort(intSubList() As Int32, ByVal intFirst As Int16, ByVal intLast As Int16)

Dim Low As Int16, High As Int16

Dim Pivot As String

Low = intFirst

High = intLast

Pivot = intSubList((intFirst + intLast) \ 2)

Do

While intSubList(Low) < Pivot

Low = Low + 1

End While

While intSubList(High) > Pivot

High = High - 1

End While

intCompCount += 1

If Low <= High Then

swap(Low, High)

Low = Low + 1

High = High - 1

End If

Loop While Low <= High

If intFirst < High Then quickSort(intSubList, intFirst, High)

If Low < intLast Then quickSort(intSubList, Low, intLast)

End Sub

' The selection sort algorithm sorts an array by repeatedly finding the

' minimum element(considering ascending order) from unsorted part and

' putting it at the beginning. The algorithm maintains two subarrays

' In a given array.

' 1) The subarray which Is already sorted.

' 2) Remaining subarray which Is unsorted.

Private Sub selectionSort()

Dim minDex, intLoop1, intLoop2 As Int32

For intLoop2 = 0 To arrSampleNums.Count - 2

minDex = intLoop1

For intLoop1 = intLoop2 + 1 To arrSampleNums.Count - 2

If arrSampleNums(intLoop1) < arrSampleNums(minDex) Then minDex = intLoop1

Next

swap(minDex, intLoop2)

Next

End Sub

Sub swap(intNdxA As Int32, intNdxB As Int32)

Dim intTemp As Int32

intTemp = arrSampleNums(intNdxA)

arrSampleNums(intNdxA) = arrSampleNums(intNdxB)

arrSampleNums(intNdxB) = intTemp

End Sub