# Preliminary

**CS 2233 Data Structures and Algorithms Sorting**

Using the framework discussed in class, implement the following sorting algorithms:

1. SelectionSort
2. InsertionSort
3. BubbleSort
4. MergeSort
5. QuickSort

In addition to these, you must implement **two** of the following additional sorting algorithms:

* ShellSort
* RadixSort
* HeapSort
* BucketSort
* BinSort
* CombSort

# Exercise 1: Randomized array time performance

Have your program record the sorting time performance of each sorting algorithm for the given array sizes of n. If each algorithm does not complete in less than 10 minutes for a given strategy, record ‘inf’ or ‘∞’ for that n and all larger n for that strategy. Record a ‘0’ when the sort is too fast to measure. Be sure to indicate the name of your additional sorting algorithms in the space below the five required strategies.

Your program should produce a text file with .CSV (Comma Separated Value) extension so that you can import it into a spreadsheet program easily. Please look at the sample .CSV file and how it is constructed. Note that your data will be different and there will be data for each strategy. Import the file into your spreadsheet and **create a report with a chart (or charts)** showing the performance of each sort strategy as n increases. Use a table similar to this one from your CSV file to create your chart(s).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n= | 128 | 256 | 512 | 1024 | … | … | … | 2M | 4M | 8M |
| 1. Selection |  |  |  |  |  |  |  |  |  |  |
| 2. Insertion |  |  |  |  |  |  |  |  |  |  |
| 3. Bubble |  |  |  |  |  |  |  |  |  |  |
| 4. Merge |  |  |  |  |  |  |  |  |  |  |
| 5. Quick |  |  |  |  |  |  |  |  |  |  |
| 6. |  |  |  |  |  |  |  |  |  |  |
| 7. |  |  |  |  |  |  |  |  |  |  |

# Exercise 2: Corner-case time performance

Modify your program to execute the sorting algorithms on an array that is already sorted. Record the time performance of each sorting algorithm for the given array sizes as you did in Exercise 1 and include the chart(s) in the report. Again, indicate the name of your additional sorting algorithms.

Finally, include answers to the following questions in your report:

# Questions:

**Q1:** Assume you were given a task at your next programming job to sort a collection of objects. What are the first three questions you should answer before starting to write the code?

How many items do I need to sort?

How close to inorder is the data going to be?

How much space on the computer am I working with?

**Q2:** Having answered those three questions, is that enough to make a definite decision as to which sorting algorithm you will use?

Yes, if there are less items, I would use Bubble, Bin, or Selection.

Bubble sort, if the list was nearly in order with a few changes, no matter the size.

If there are many items, random order, I am working with less space. I would use merge, quick, or heap. More than likely quick because it is simpler to code. I didn’t code heap as an in place algorithm but I think I could’ve making take up less space.

If there are many items, random order, and more space to work with. I would use bin sort.

**Q3:** Describe a hybrid sorting algorithm that is good at both very large and small collection sizes. Give the pidgin- or pseodo-code for this hybrid algorithm.

First the algorithm would do bubble sort without the swapping to check if the array is already in order. If the array is out of order**,** then the algorithm would check if there are less than 131071. If there is less than 131071 then it would use quick sort, if not it would use bin sort.

hybridSort(array, left, size)

{

For i from 0 to size:

For j from 0 to size-i-1

If array[j] > array[j+1]:

Swap=true

Break

If Swap = false return

If size >= 131071:

BinSort(array, size)

Else:

QuickSort(array, left, right)

}

BinSort(array, size)

{

Counts[10001]

For from 0 to size:

Counts[array[i]]++

Index = 0

For from 0 to 10001:

If counts[i] != 0:

For from counts[i] to 0:

Array[index+1] = i

}

QuickSort(array, left, right)

{

If left >= right return;

Pivot = arrayMedian

Int index = partition(array, left, right, pivot)

QuickSort(array, left, index-1)

QuickSort(array, index, right);

}

Partition(array, left, right, pivot)

{

While left < right

While(array[left] < pivot) left++

While(array[right > pivot) right--

If left < right

Swap(array[right], array[left])

Left++

Right--

}