Algorithms and Data Structures

Wk7 – exercise –Names sorter

In this exercise you will practice the implementation of three variations of well-known sorting algorithms and explore their performances by measuring elapsed times of sorting a list of names of increasing size.

You can use the starter project provided, such that you can focus your effort on the implementation of the algorithms in the Sorter class.

Complete the sorter methods in the Sorter class.

`public List<E> bubbleSortUntilDone(List<E> items, Comparator<E> comparator)`

Complete this variant of the bubble sort which detects in the inner loop whether any swap of two items has been made. If nothing has changed in the inner loop, you know that the complete list has been sorted and further iteration of the outer loop will not be required anymore.

`public List<E> mergeSortViaAuxiliary(List<E> items, Comparator<E> comparator)`

Complete this variant of a merge sort, which declares one auxiliary array in the wrapper method which gets passed into the recursive calls and the merge for reuse. This reduces the overheads of repeatedly declaring and cleaning up auxiliary arrays in the merge calls. Although not essential, it is recommended that you adhere to the 'from' and 'to' boundaries of the partial merge also for accessing the auxiliary array (clean code.)

Basic use of the auxiliary array eliminates some memory management overheads, but still requires copying of the merged result back into the original items array at the end of each merge. If you want to eliminate those as well, then you should return the max recursion depth from the mergeSortPart method and use that value to alternate merging from items into auxiliary or auxiliary into items. For that the auxiliary and items parameters should have the same List type. This is a complex optimization beyond the scope of our course (but a nice programming challenge if you aspire the academic track later on…)

`public List<E> batchersOddEvenMergeSort(List<E> items, Comparator<E> comparator)`

is a cross-over of merge sort and bubble sort, with very interesting iterative pseudo code of FOUR nested loops and a time complexity of Big-O($N*(\log N)^2$). That is very close to N.logN!

At https://en.wikipedia.org/wiki/Batcher_odd%E2%80%93even_mergesort you find an explanation and further references. Unfortunately, the pseudo-code there is wrong… The correct pseudo code is:

```
for p = 1, 2, 4, 8, ... # as long as p < n
 for k = p, p/2, p/4, p/8, ... # as long as k >= 1
  for j = mod(k,p) to (n-1-k) with a step size of 2k
   for i = 0 to min(k-1, n-j-k-1) with a step size of 1
    if floor((i+j) / (p*2)) == floor((i+j+k) / (p*2))
     compare and sort elements (i+j) and (i+j+k)
```

Run the main program for execution time analysis.

The main program runs your sorter methods on lists of names of different sizes and gathers the execution time results. It also compares systematically all outcomes with the outcome of collection sort and reports any differences in sorting order.

Don’t forget the -Xint JVM option in the run config.

Hope you can produce a nice picture of the execution times like below.

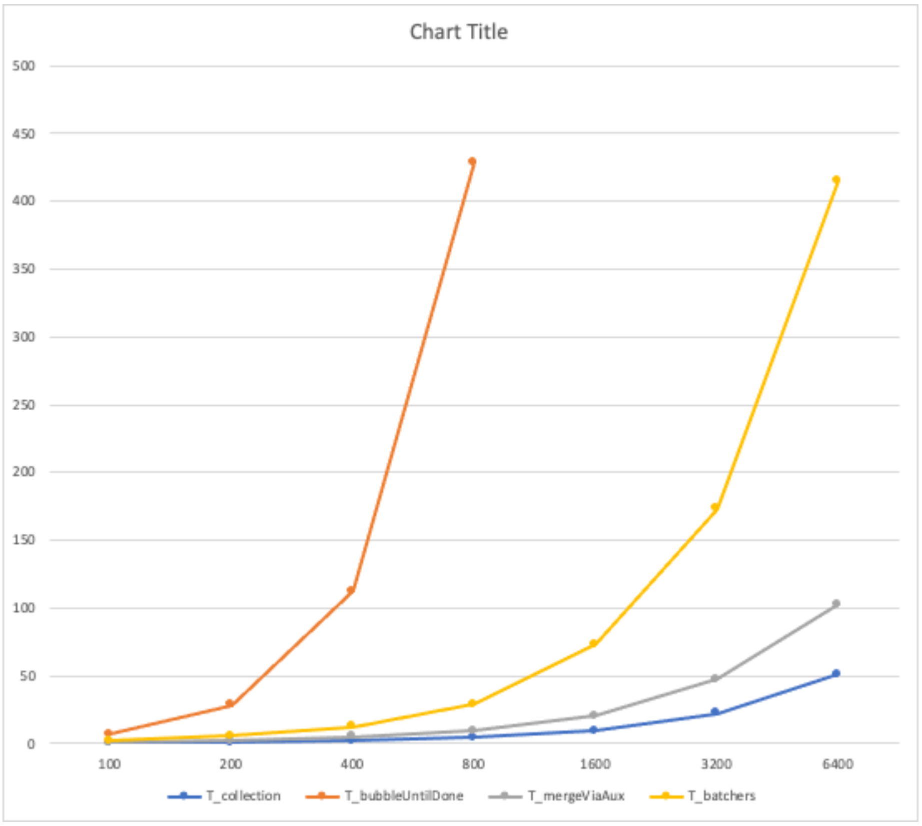
Chart Title
T_collection
T_bubbleUntilDone
T_mergeViaAux
T_batchers