Overview

For the 5th practical we had to write the implementations of a MergeSort and an InsertionSort, once completed, we had to preform various micro benchmark tests on each implementation and compare the results.

Design

I began by making an interface, which implementation, implements. The interface only contains a single method "sort" which takes in a Comparable array.
The reason I made an interface was to allow me to parametrized testing later on and the reason for making the sorting methods take in an array of type "Comparable" was because the Comparable object class contains methods for comparing various different data types, thus meaning each implantation can sort arrays of different types.
I have also added a private attribute "swaps" which keeps track of how many swaps it has taken each different implantation to sort the input array completely. The values are later used as benchmark.

Merge Sort: Worst-case O(n log n), Best-case O(n log n), Average O(n log n)

The first sort algorithm I implemented was the Merge sort. The main two methods in this class are the: "divide" and "merge" methods. The divide "method" is a recursive method, which is used to split the input array. The base case for this method is when the input array is of length one. When first called two values "lowerIndex" and "upperIndex" are passed in. Initially "lowerIndex" is 0 and "upperIndex" is the length of the input array minus 1.
Upon each call the method first calculates the midpoint of the array, if this isn't an integer, the "Math.floor" function is applied to the calculated midpoint, ensuring it is always rounded down. Next are the two recursive calls, the first call hands back in value of "lowerIndex" which is unchanged but the “upperIndex” parameter is set equal to the calculated "midPoint". The next recursive call is similar but hands in the value of the "midpoint" variable as the "lowerIndex" parameter and keeps the value of the "upperIndex". This recursion stops when the original input array has been split into individual elements.
Finally, once this happens, the stack starts calling the "merge" method. This method takes in: an array and the “lowerIndex", "middle" and "upperIndex" values as well as the temporary array. It then copies both sides of the input array to a temporary array. Next the input parameters are copied and the "middle" variable is incremented by 1. Following this is a while loop, which terminates when the “lowerIndex” variable reaches the value of the “middle” variable, or when the “middle” variable reaches the value of the “upperIndex” variable. Once entered, the while loop, uses the “compareTo” method to copy the smallest values from either the left or the right side back to the original array. Finally, the remaining elements in the left part of the array are copied back to the original input array.

Insertion Sort: Worst-case O(n2), Best-case O(n), Average O(n2)

The next algorithm I implemented was the Insertion Sort. This class was much easier to implement and consists of only one substantial method “Sort”. Initially when the method is called a new Comparable object “temp” and an integer “y” are initialized. Next, the first of two for loops, is entered this loop is used to iterate, left to right, over the array. Once entered the variable “temp” is set to the element at the current index in the input array. Next the second loop is entered, here the value of the variable “temp” is compared to the element left of it, if its larger the element to the left is shifted one place left. This continues until the element left to the of the current element is smaller or the loop hits the front the array.

Extensions

For this practical I decided to implement the following sorting algorithms: Bubble Sort, Quick Sort and a Heap Sort as an extension. This has allowed me to compare all 5 sorting algorithms and see where their strengths and weaknesses lye.
I have also used the Junit-Benchmark library, so I can properly analyze each of the implementations and obtaining the following performance benchmarks: Average execution time and memory usage (GC time, GC average time and GC calls), moreover I have also recorded the number of swaps preformed in each algorithm, which is an interesting benchmark accompanied with speed. Finally, I have graphically plotted all my results using excel. In order to record my results, I used the Junit-Benchmark library with the “@BenchmarkHistoryChart” annotation that stores the results to each test in a H2 database and a JSONP file located in the /target directory. The Junit-Benchmark library also automatically creates a HTML plot using the data from the JSON file. However, you cannot change the y axis too logarithmic, hence why I used excel.

Memory footprint:

In order to test how much memory was used by each algorithm I have created a class “SortBenchmarksMemory” which uses parametrized testing to test the memory footprint of each of the sorting algorithms. In this class there is an object “runtime” which allows the application to interface with the environment in which the application is running. Thus meaning I can get the total memory and free memory before and after each test, allowing me to calculate the amount of memory used by each sorting algorithm. To ensure the GC was not obscuring my results, the method “getMemoryUsage” first runs the GC before calculating the result. The Junit Benchmark library does include a feature to record the memory footprint of a method or object, however it precision is not adequate to test methods which only use a small amount of memory.

Bubble Sort: Worst-case $O(n^2)$, Best-case O(n), Average $O(n^2)$

The first extension I decided to implement was the bubble sort. The bubble sort algorithm is an internal exchange sort and was one of the simplest algorithms to implement. The algorithm essentially works by sinking the smallest items to the bottom of the array / rising the largest items to the top of the array. It does this by comparing adjacent pairs of objects in the array. If the element on the left is larger the elements are swapped. This process continues until the largest of the objects, eventually "bubbles" up to the highest position in the array. After this occurs, the search for the next largest object begins. The swapping continues until the whole array is in the correct order. When the function is called a Boolean variable “swapsMade” is set true. Following a while loop is entered which only terminates when the “swapsMade” variable is set false. Next the “swapsMade” variable is set false and a for

loop is entered which iterates over the input list. In the for loop there is a single if statement, which is used to check adjacent pairs. If a swap is made the "swapsMade" is set true.

Quick Sort: Worst-case $O(n^2)$, Best-case O(n log n), Average O(n log n)

Next I decided to implement the Quicksort algorithm. The quick sort uses divide and conquer to gain the same advantages as the merge sort, however the quick sort algorithm has a smaller memory footprint. This said, there is a trade-off, as it is possible that the list may not be divided in half. When this happens the performance is diminished and degrades to $O(n^2)$. The most substantial method in the "QuickSort" class is the "quickSort" function, which is the recursive function, where the base case is when the input list size contains 1 element. When the function is first called the pivot point set equal to half the input list size, I choose to use the middle value as the pivot (this can lead to poor performance bellow certain thresholds). Next a while loop is entered, which contains two further while loops and terminates when the input array has been completely traversed. The two inner while loops user the "compareTo" function which records how many items need to be reordered. Once calculates the "shift" method is called which shifts elements with values less than the pivot to come before the pivot and all elements with values greater than the pivot to come after it. Finally, if the either side of the pivot has more than one element the recursive function is called.

Heap Sort: Worst-case O(n log n), Best-case O(n log n), Average O(n log n)

Finally, I decided to implement the Heap Sort Algorithm. The heapsort algorithm is made up two steps where the first step is building a max heap from the data with the layout of a partly sorted, complete binary tree. This is stored in an array of type Comparable. The next step is sort the array, this is done by creating a new array and then repeatedly removing the largest element from the heap which is the root the of the heap / binary tree, and inserting it into the array. The heap is then updated after each removal. Once the sort method is called in the class, the "buildHeap" method is called. Here a for loop is entered, which calculates where to start the heapify processes by dividing the list size by 2 and subtracting one, i.e. ignoring any leaf nodes in the input array. Once in the loop the "maxHeap" function is called. This is a recursive function, where the base case is when the there is only a single element in the tree. The function takes in the index of a non-leaf node and the input array. The indexes of the input node's children are then calculated and a new Comparable variable "largest" is created. Next a check is preformed to ensure the left node isn't the last node, if not, the "compareTo" method is used to check if the left child node is greater than its parent, if so the "largest" variable is set to the input nodes left child. Otherwise the "compareTo" method is then used to check if the right child node is greater than the parent if so the "largest" variable is set to the input nodes right child and then the base case is checked. Once the heap has been created execution returns to the "sort" function, here another loop is entered. The root element is removed and placed into the sorted array, the item at the back of the heap is then moved to the front and the heap size is reduced by 1. Because the swap ruined the heap property the "maxHeap" function is called again to restore it.

Testing

I have written unit tests to the basic functionality of every implantation. Because the overall output of each implantation should be the same when using the same input data for each of them, I have used Junit's parametrized testing to test the functionality of all the classes.

Once I was satisfied that each of the implantations works as expected, I went on to preform various different benchmarks. These included:

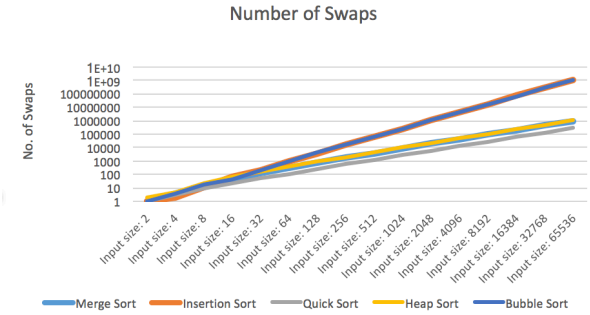
- Recording execution speed for sorting a: randomized input a sorted input and a reversed input.
- Recording number of swaps
- Recording memory footprint

All then micro benchmark tests for speed and swaps are in the "SortBenchmarksSpeedAndSwaps" class which extends the "AbstractBenchmark". This allowed me to me accurately record the performance of all the sorting algorithms.

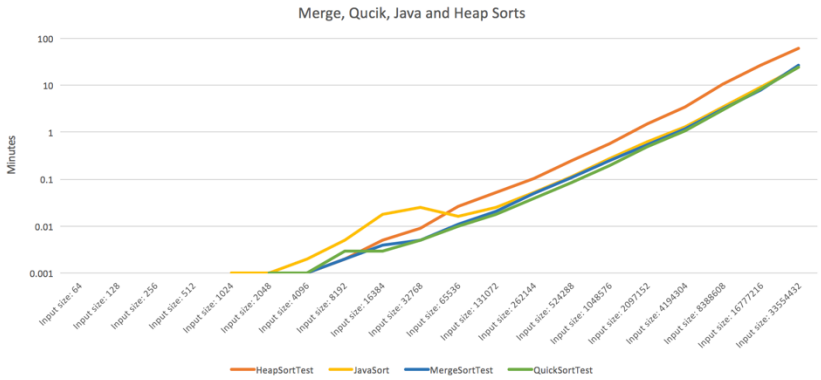
Each micro benchmark is run 23 times, the first 3 times are warm up runs and the performance is not recorded, this is done to allow the JVM to preform optimizations and is ensure its working efficiently. Then the average and standard deviation is recorded for the remaining 20 preform tests. In practice, the methods containing the tests will not be executed with a cleaned heap, so I have disabled Java's garbage collector for the tests and taken the average from multiple test runs. Moreover, to ensure the tests are as accurate as they can be the setup and input for the tests is created once and remains fixed for all tests and making the test code use a singleton object to avoid allocating other memory to anything else. All the memory benchmark tests are located in "SortBenchmarksMemory" class. I have tested each implantation with list of sizes: 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16} and 2^{17}.

I have also tested the Merge sort, Java sort, Quick sort and Bubble sort algorithms with lists sizes: 2^{18}, 2^{19}, 2^{20}, 2^{21}, 2^{22}, 2^{23} and 2^{24}.

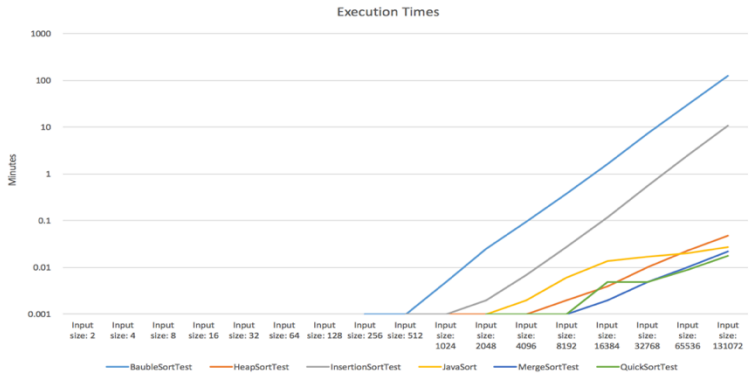
Here is a chart showing the number of swaps each algorithm preformed before the data was sorted. The y axis is in log to base 2. When the size of the input is very low, the bubble sort and insertion sort actually out preform the other sorting algorithm in terms of number of swaps. However, this very quickly changes and the number of swaps massively increases when the data set gets bigger. This graph also shows us that the Merge sort and Heapsort have very similar results and preformed also equal number of swaps for a given data size. However, the clear winner here is the quick sort which massively outperforms the other sorting algorithms.



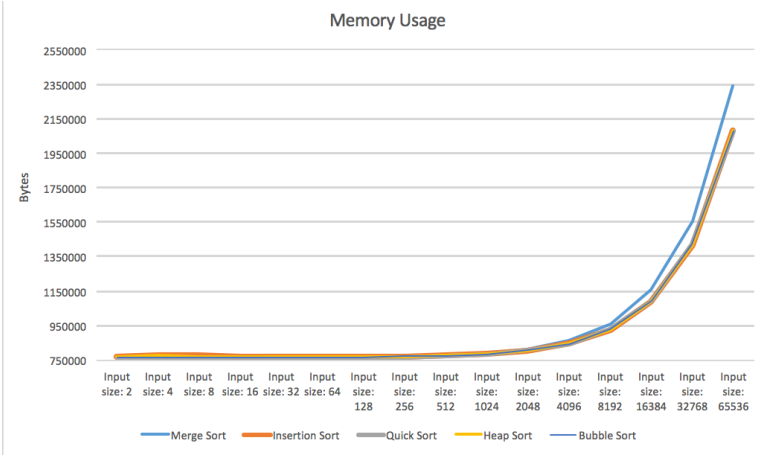
This plot shows the execution times for all the sorting algorithms. The size of input list was increased by powers of 2, all the way to 2^{18}, for each test run. At this point the bubble sort was taking excessive amounts of time to complete and therefore I stopped testing at that point. Again this graph is logarithmic. As you can see the bubble sort has the worst performance taking over 300 minutes to sort a list of size 2^{18}. The insertion sort had the second worst performance taking over 100 minutes to sort a list of size 2^{18}. What is interesting about the insertion and bubble sort is that, for small data sets the bubble sorts performance is much worse than that of the insertion sort. However, this trait switches with larger datasets, as the bubble sort performance increases and the insertion sorts performance decreases. The reaming, Java, Heap, Merge and Quick sorts performances all significantly outdid the Merge and Bubble sorts. Initially on small datasets, Javas own sorting algorithm is out performed by the Heap, Merge and Quick sorts. However, its performance increases will larger datasets and it overtakes the Heap Sort. The thing to notice about the Java sort, unlike any of the other sorts, is that its performance almost Plato's. The merge sort and quick sort appear to have very similar characteristics and although the quick sort is massively outperformed by the merge sort with small datasets, it finishes with the best performance.



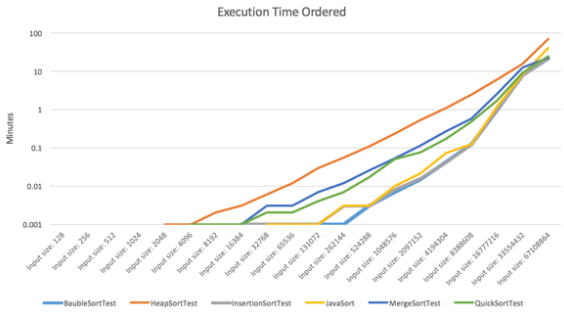
This plot shows the execution times for times for just the merge, quick and heap sorting algorithms. The size of input list was increased by powers of 2^6, all the way to 2^{25}, for each test run. Again this graph is logarithmic. As you can see the heap sort has now got the worst execution time and is clearly being outdone by the other three algorithms. From the plot we can also see the Java sort in fact realigns with the merge and quick sort, in terms of execution times.

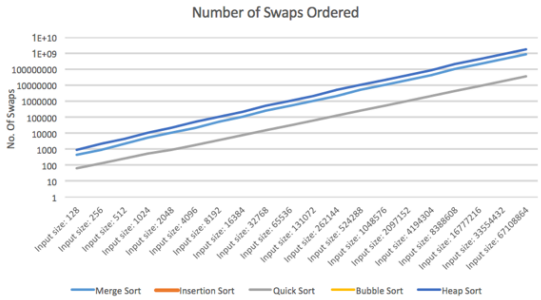


This is plot shows the memory used by each sorting algorithm, the insertion, quick heap and bubble sort all use a similar amount of memory and there's not to split them. However, the merge sort algorithm, consistently consumes more memory when the input list size exceeded the 2048 items threshold. This because the algorithm is not an "in-place" algorithm and a temporary array is used which consumes more memory.

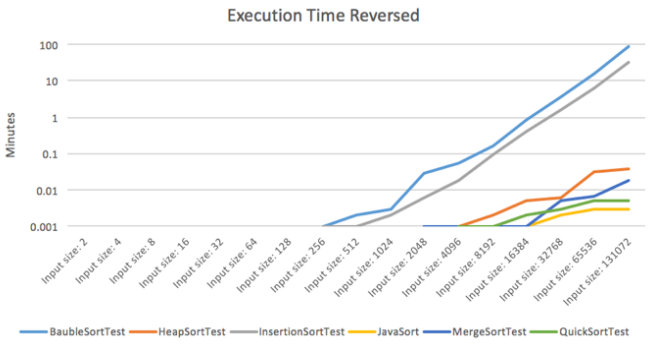


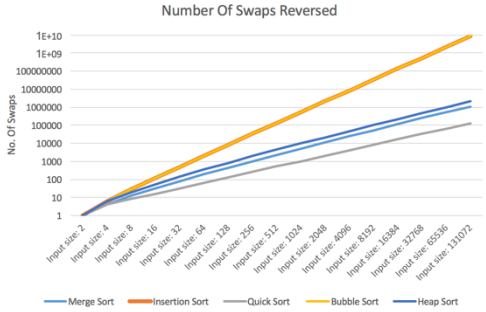
Below show the plots of the number of swaps and execution time, for all the sorting algorithms, when the input list was already sorted, thus finding their best execution time. As you can see the non-recursive algorithms performance start of much better than the recursive algorithms, moreover the number of swaps was 0 for the non-recursive algorithms is constantly 0. Because the recursive algorithms still split the input list, the number of swaps don't stay at 0.



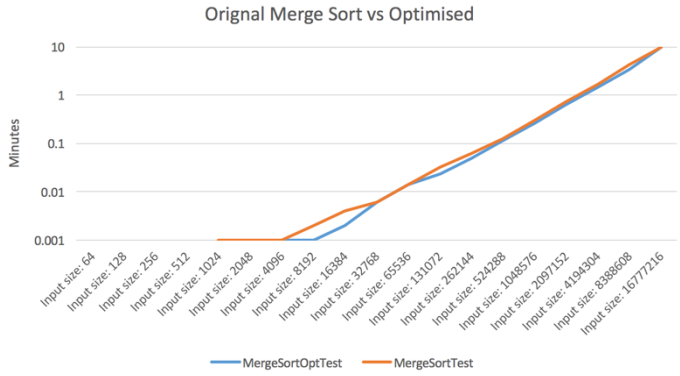


Next I tested all the algorithms will an input list which was completely revered, thus finding each algorithms worst execution time. The plots bellow again show the number of swaps and execution time, for all the sorting algorithms. As you can see the recursive algorithms have a pretty similar performance to when the input list was already sorted. The major difference is the that now the non-recursive algorithms start taking a lot longer to complete and this happens much earlier on. Moreover, the number of swaps also rapidly increases much earlier on compared to the recursive algorithms. As you can see the insertion and bubble sort preform a very similar number of swaps, although the insertion sort has a quicker execution time.

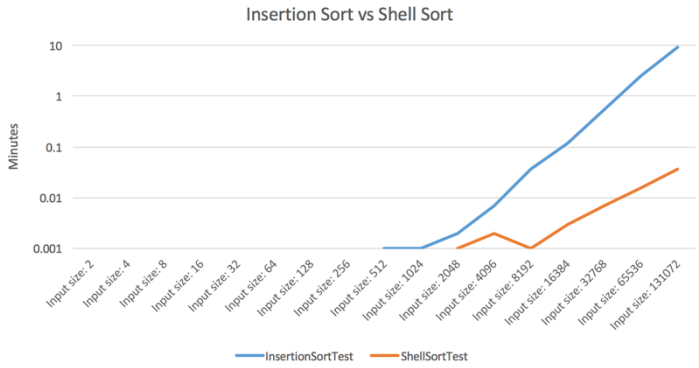


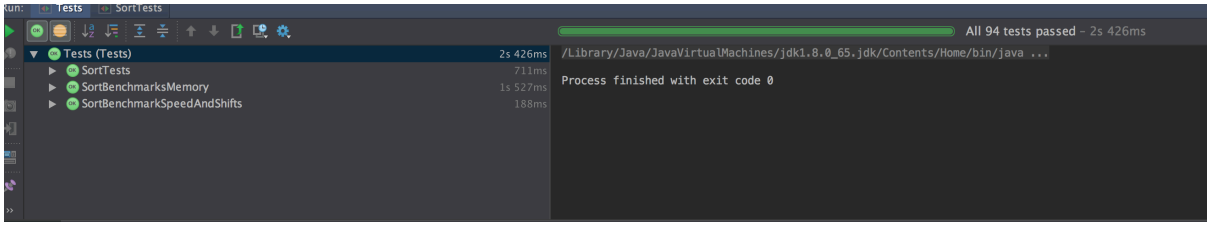


I have also decided to implement an optimized version of the Merge sort. The optimizations where quite basic and include:
The use of an insertion sort for small subarrays. This is because a Merge sort algorithm has too much overhead for tiny subarrays. Therefore, the threshold at which the merge sort uses an insertion sort is a list of 10 items or less. I have also added another optimization which stops the sorting processes if the input already sorted this is done by checking if the largest item in first half of the array is less than the smallest item in second half. As you can see the optimized implantation of the merge sort preforms much better, however as the data set increases both algorithms performance start becoming very similar.



Finally, I have decided to implement an optimized version of the Insertion sort, known as a shell sort. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left. The algorithm works by breaking the original list into a number of smaller sub lists, each of which is sorted using an insertion sort. The way that these sub lists are is by first defining the gap, then creating the sub lists by choosing all items that are x(the gap) items apart. In my implantation the gap was set equal to the list size divided by 3. Here you can see the performance of the shell sort massively out does the original insertion sort. With a far shallower execution time cure, moreover the shell sort takes a much bigger input list size before the speed of the algorithm can be measured.





I have added in the design section the worst-case, best-case and average-case in terms of the Big 0 notation, for all the algorithms, based on my recordings.

Evaluation

After testing my program, I pleased to say it works as expected and all the tests pass. I am also pleased with the micro bench results I was able to record for all of the different implantations of a sorting algorithm. The results clearly show where the strengths of each algorithm lie. The results clearly show that the recursive algorithms, on average, have far better characteristics than the non-recursive algorithms. From my results I can conclude the quick sort has the best overall characteristics with the least swap and the fastest execution time for a list containing random numbers. I'm also pleased with my extensions, as they have added depth in to the research and have allowed me to make better comparisons. Moreover, the optimizations for the merge sort and insertion sort

do increase their performance even further. If I had longer I would have liked to implement am in place version of the merge sort. I would have also liked to have offloaded my tests to the CS servers so they could run far more extensive tests, however this was not possible, so all the tests where preformed on a MacBook Pro, 2.8 GHz Intel Core i7 with 16GB Ram.