INSTITIÚID TEICNEOLAÍOCHTA NA GAILLIMHE-MAIGH EO
GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Project: Benchmarking Sorting Algorithms

Jan Hazincak
G00376399@gmit.ie

Contents

Introduction

An algorithm describes the steps which need to be followed to accomplish a specific task (Rosen, 2012, p.191). Each computer software consists almost entirely of algorithms which are through well-defined steps processing input to desired output. One of the first problems which was a computer scientist dealing with was how to sort data in the most efficient way. To highlight the importance of the sorting, Rosen says that an amazingly large percentage of computing resources used for this data organising activity (Rosen, 2012, p.196). The problem of sorting is solved by sorting algorithms and as can be seen in this project, their correct implementation and use can save both time and computer computing resources. Throughout history of computing there were many sorting algorithms analysed, starting with a simple comparison-based Bubble sort from 1956 and ending with efficient sorts such as Timsort dating to 2002 or the library sort being published in 2006 (Rosen, 2012, p.196). After doing some research I can say that no sorting algorithm can be considered the best and this can be proven by their usage diversity between programming languages. For instance, Java (SE 14) itself embeds more than one sorting algorithm. A java.util.Arrays class implements Dual Pivot Quick sort and java.util.Collections implements algorithm called Merge sort. This project is benchmarking and analysing following algorithms:

- Bubble sort
- Selection sort
- Merge sort
- Quick sort
- Counting sort

In addition, there is Dual Pivot Quick Sort from the *java.util.Arrays* class included in the benchmarking application.

Time and space complexity

Comparing an efficiency of the sorting algorithm, by comparing their running time is not always the best practice because of computers hardware. The hardware influences the running time of all processes done by computers, including the running time of the sorting processes performed by sorting algorithms (Downey, 2017, p.7). If we were to run an implementation of any sorting algorithm on a desktop computer that was built in 2020 and compare that to the running time of the exact same implementation on a computer that was built 30 years ago, obviously the implementation is going to run slower on the older hardware. To demonstrate this, I attached the screenshot of the algorithms running times when my laptop was unplugged therefore all laptop components including CPU were operating in power safe mode and the running time increasement of each algorithm was around 30%. That's the reason why there are a more objective measures used in the computer science – time and memory complexity.

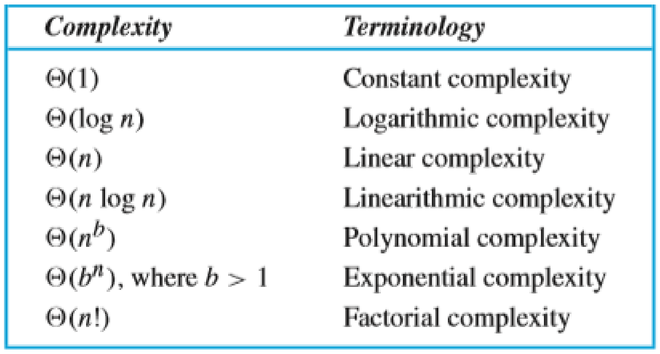
The time complexity is the number of steps involved to run an algorithm (Rosen ,2012 ,p. 219). On the other side the memory complexity is the amount of memory it takes to run an algorithm. Describing the memory complexity, there are two important concepts to mention which are in-place and non-in-place sorting. The in-place sorting performs whole sorting process in one field and the implementation doesn't create any other temporary data structures for storing data.

However, the in-place sorting algorithms still require a small amount of extra storage for variables like tracking indexes or variables of primitive types. If the extra memory being used doesn't increase as the number of items in the array increases. Then it is still and in-place algorithm (Downey, 2017, p.138).

On the other side the non-in-place sorting algorithms, require extra space which size depends on the size of elements being sorted. However, nowadays, even low-end computers come with at least 4GB of RAM so memory complexity shouldn't be an issue. This is the reason why the time complexity is mainly concerned by programmers while implementing code. Nevertheless, the in-place sorting algorithms still have their use in embedded system which are running in limited memory.

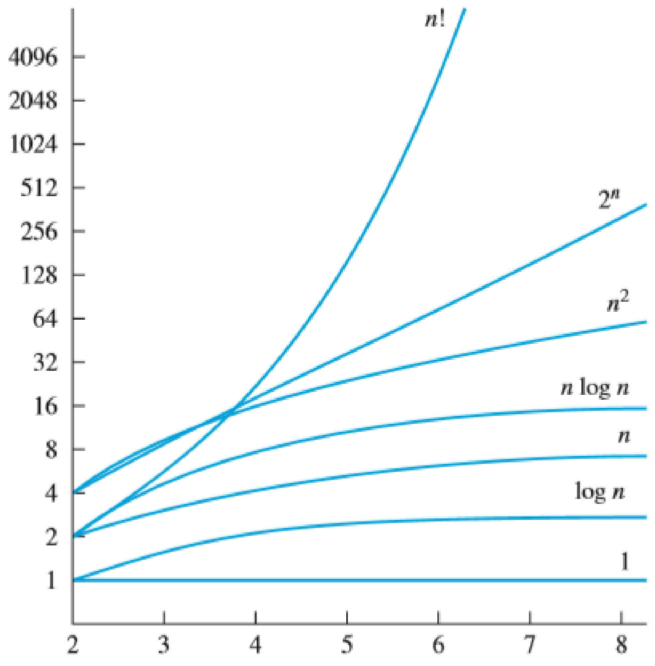
When the time complexity is considered, the worst-case scenario is always looked at and how the algorithm will perform if the number of items being sorted grows. The time is represented by a rising curve on the graph and this growth of functions is also known as Big O notations which are very useful when analysing algorithms for their time complexity (Downey, 2017, p.10).

The following table of the Big-O terms and graph are showing the most common Big-O notations, their meaning and their curves: The curves represent their time growth toward input size increasement.



Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Commonly Used Terminology for the Complexity of Algorithms (Rosen, 2012, p.226)



A display of the Growth of Functions Commonly Used in Big-O notations (Rosen, 2012, p. 211)

Stable and unstable sort algorithms

When it comes to sorting algorithms, there are stable and unstable sorts. Whether the algorithm is stable or unstable can be seen if there are any duplicated values in the array of values being sorted. If the original ordering of the two duplicates is preserved then the algorithm is stable, otherwise, it is the unstable algorithm. For better illustration see simple field sorted firstly with stable and then with unstable algorithm.

1. Unsorted array of random integers with two duplicates of value “10”.

1	10	11	10	18	9	5	4

2.a. The field sorted by stable algorithm.

1	4	5	9	10	10	11	18

2.b. The same field sorted by unstable algorithm.

1	4	5	9	10	10	11	18

Sorting Algorithms

Bubble sort

Bubble sort is a simple sorting algorithm. The algorithm iterates over an input, in every step it compares two elements and if the lower value is on the right side (the side depends on the implementation) from the element with higher value, bubble sort swaps them. (Rosen, 2012, p.196)

This algorithm's performance degrades quickly as the number of items being sorted grows. The bubble sort is an in-place algorithm as the array is logically partitioned into a sorted and unsorted part and no other array needs to be created to store temporary values.

The bubble sort has O(n2) time complexity therefore it is a quadratic algorithm so that means that in the worst case it will take 100 steps to sort 10 items, 10 000 steps to sort 100 items, 1 000 000 steps to sort 1000 items.

Selection sort

Selection sort is another simple comparison-based sort. This algorithm divides the array into the sorted and unsorted partition. The selection sort is traversing through the array and is looking for the largest (or smallest if sorting in descending order) element in the unsorted partition. If the largest element is found it is selected and swapped with the last element in the unsorted partition. And at that point, that swapped largest element will be at its correct sorted position.

The same as bubble sort, the selection sort is in-place algorithm as no other data structure is created to temporarily store sorted values and it has the same time complexity as the bubble sort, which is the O(n2) time complexity.

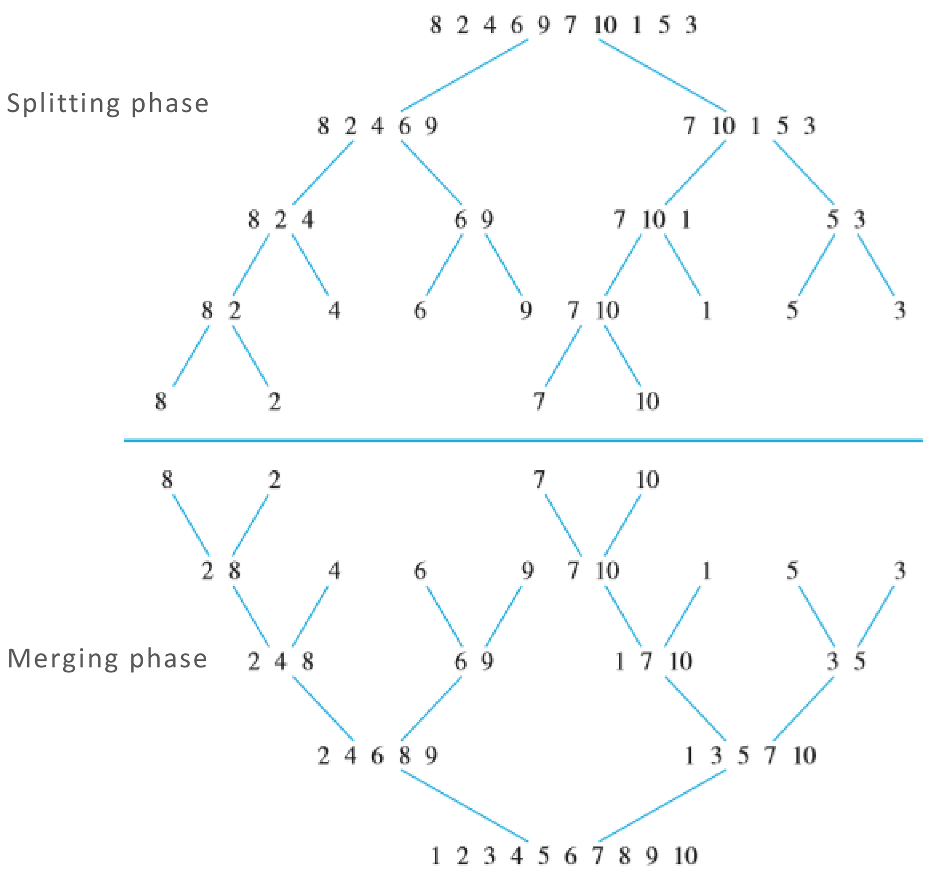
The selection sort is unstable algorithm because if there are any duplicate elements, there is no guarantee that their original order relative to each other will be preserved because on each iteration, the algorithm swaps the largest element, with the element occupying the last position in the unsorted partition regardless its value.

Merge sort

Merge sort is an efficient comparison-based sort. The main idea behind merge sort is to merge already sorted subarrays using additional temporary array. The Merge sort can be split into two main phases: splitting and merging phase (Rosen, 2012, p.367).

The splitting phase starts with an unsorted array of elements which is divided into two sub arrays. The first array is the left array, and the second array is the right array. Merge sort then splits the left and right arrays into two arrays each and this process is repeated using the recursion until all the arrays have only one element each.

After splitting phase comes merging phase. The merging process merge every left or right pair of sibling arrays into a sorted array. After the first merge, there is a couple of 2-element sorted arrays. After the second merge of those sorted arrays the merge produces a couple of 4-element sorted arrays. This merging process is repeated until there is a single sorted array. Splitting phase



The Merge Sort of 8, 2, 4, 6, 9, 7, 10, 1, 5, 3 (Rosen, 2012, p.367)

The merge sort is not an in place algorithm. Splitting phase is in place as all splits are done on one array but that's not true about the merging phase as a temporary array is used to merge each pair of sibling arrays.

The merge sort is stable algorithm because when there are two duplicate elements, first in left the subarray and the second in the right, then the element in the left will be the one that's copied into the temporary array first and the second duplicate from the right subarray will be copied second and the relative ordering of duplicate values is preserved.

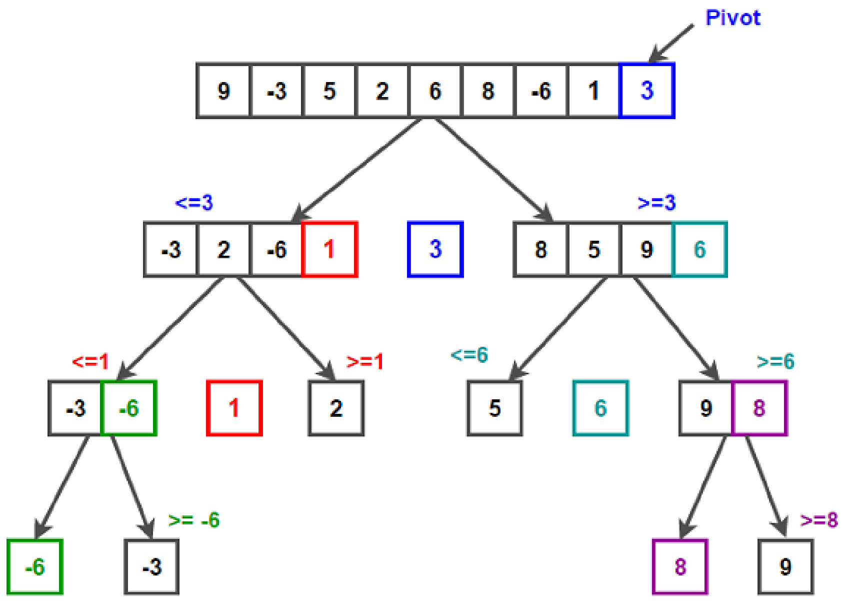
The merge sort is efficient algorithm with O (n log n) time complexity as a set of data is repeatedly divided into half.

Quick sort

Quick sort is another divide and conquer, comparison-based sort algorithm. The Quick sort uses a pivot element which divides the array into two halves. The first phase, called partitioning phase places the pivot into its correct position. Once the partitioning phase is finished everything less than the pivot element is on the left and everything greater than the pivot is on the right.

Partitioning phase

This implementation selects the last element in the array for its pivot.



https://www.techiedelight.com/quicksort/

After the initial partitioning phase, the left and right subarrays aren't sorted, just the first pivot element is in its correct position. Both subarrays, left and right are sorted through the recursion as these subarrays are partitioned into several subarrays until there are one element arrays and eventually, every element is chosen as the pivot so every element is in its correct position.

This process is very similar to the splitting and merging process of the merge sort and that's why both algorithms are named as divide and conquer algorithms. However, unlike merge sort, whole partitioning phase of the quick sort is done in-place so the quick sort is the in-place algorithm.

The Quick sort is unstable algorithm with O (n log n) time complexity. However, under very unlucky circumstances, the Quick sort might perform in quadratic time complexity. If the largest or smallest element is selected for the pivot in each partitioning step, a lot of steps would have to be taken for them to be centred.

Counting sort

Counting sort is non-comparison sorting algorithm. Instead of using less than "<", equal to "==" or greater than ">" symbols, the sort works with assumptions about the data being sorted and it uses the knowledge of the smallest and the largest element in the array.

An implementation of the counting sort creates a temporary array where are the number of occurrences of each value counted. The length of the temporary array depends on the range of values being sorted. That is the reason why the counting sort needs to know some information about sorted data and sometimes it can lead to strange scenarios. For instance, if the counting sort is used on an array of 10 elements, and the range of these 10 elements is between 1 to 10 000, the temporary array would have to be of length 10 000 to sort 10 elements. As this array needs to be created it is not in-place algorithm.

The Counting sort is unstable algorithm because when the sort is traversing over unsorted values, their occurrence number is increased in the temporary array and when the counting phase is over, the values are inserted back to the default array by decreasing occurrences number and at that moment duplicates position can be swapped toward their default position.

Although, the counting sort is one of the fastest sorting algorithm and in my opinion with very interesting sorting logic, it also has several disadvantages. As already mentioned, it is not in place algorithm and it needs to make assumptions about data being sorted which is probably not always possible. Nevertheless, the major disadvantage of the counting sort is that it only works with non-negative discrete values therefore it cannot work with floats, strings etc.

Implementation & Benchmarking

The Java benchmarking application has been developed in order to compare all described sorting algorithms. This application creates a random unsorted data of preselected size and range, applies each implemented sorting algorithm, calculates a running time of sorting process, stores calculated values and appropriately displays them in the IDE's console.

Input

Input is generated by *int[]randomArray* method with two int parameters in its signature. First parameter defines a size of input, and the second defines the range between which the "random" numbers are generated. The method returns an array of random integers values of specified size and range.

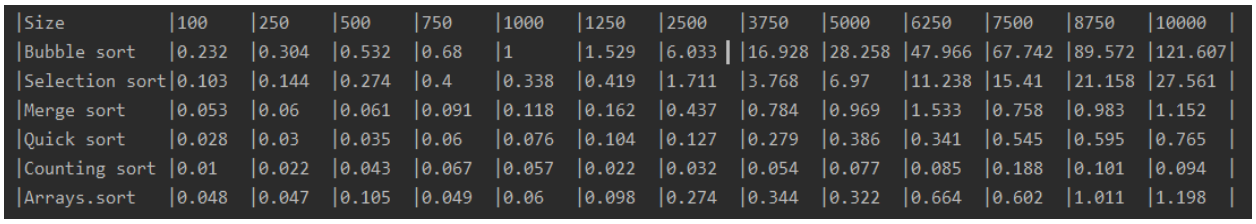
Input consists of 13 different values ranged from 100 to 10 000. This spread of values creates an input diversity and gives an opportunity to analyse the sorting algorithms performance either on small or big input size.

Calculation

As mentioned in the previous paragraph, each sorting algorithm is tested on 13 different sizes of input. Input of each size is sorted by the different algorithm 10 times and the average value of these 10 runs is then stored in data structure which is eventually displayed in the console when the program finishes its execution. It is important to mention that the *randomArray* method is called in each possible iteration therefore one randomly generated field is never stored more than once but there is different field sorted every time.

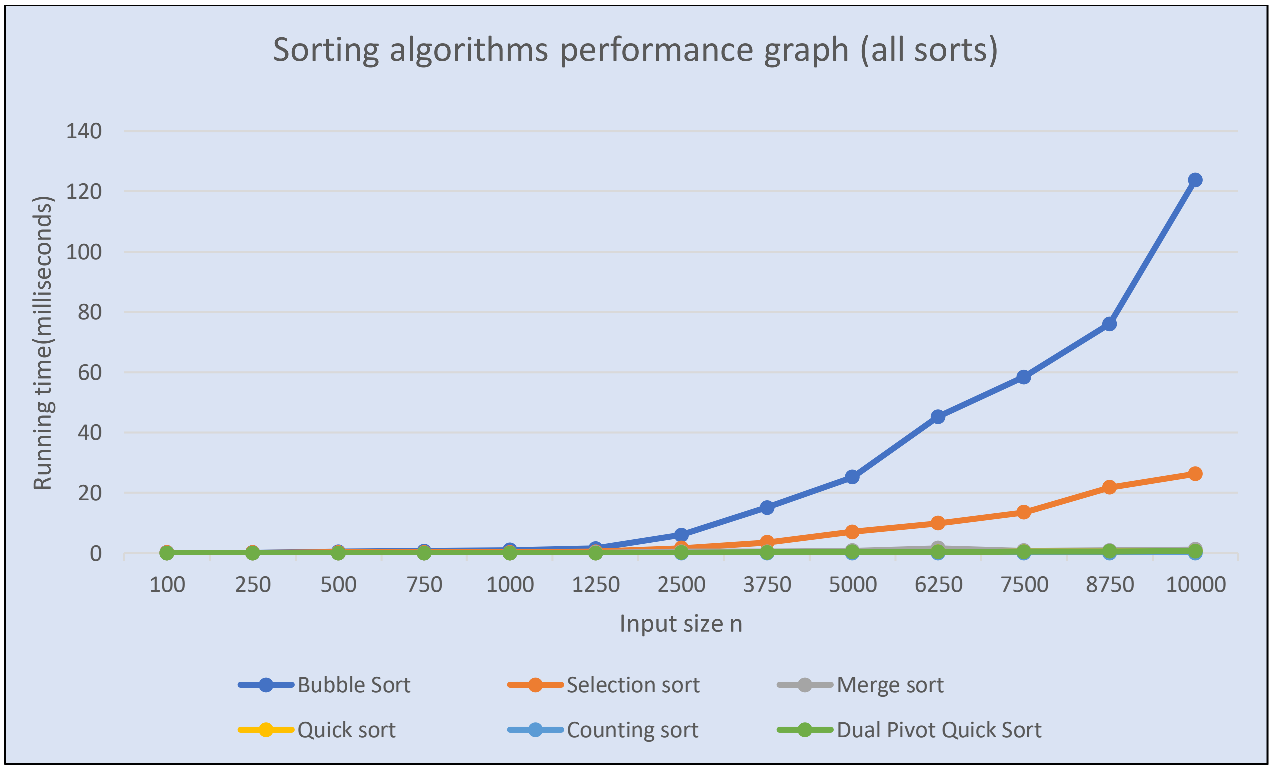
Results

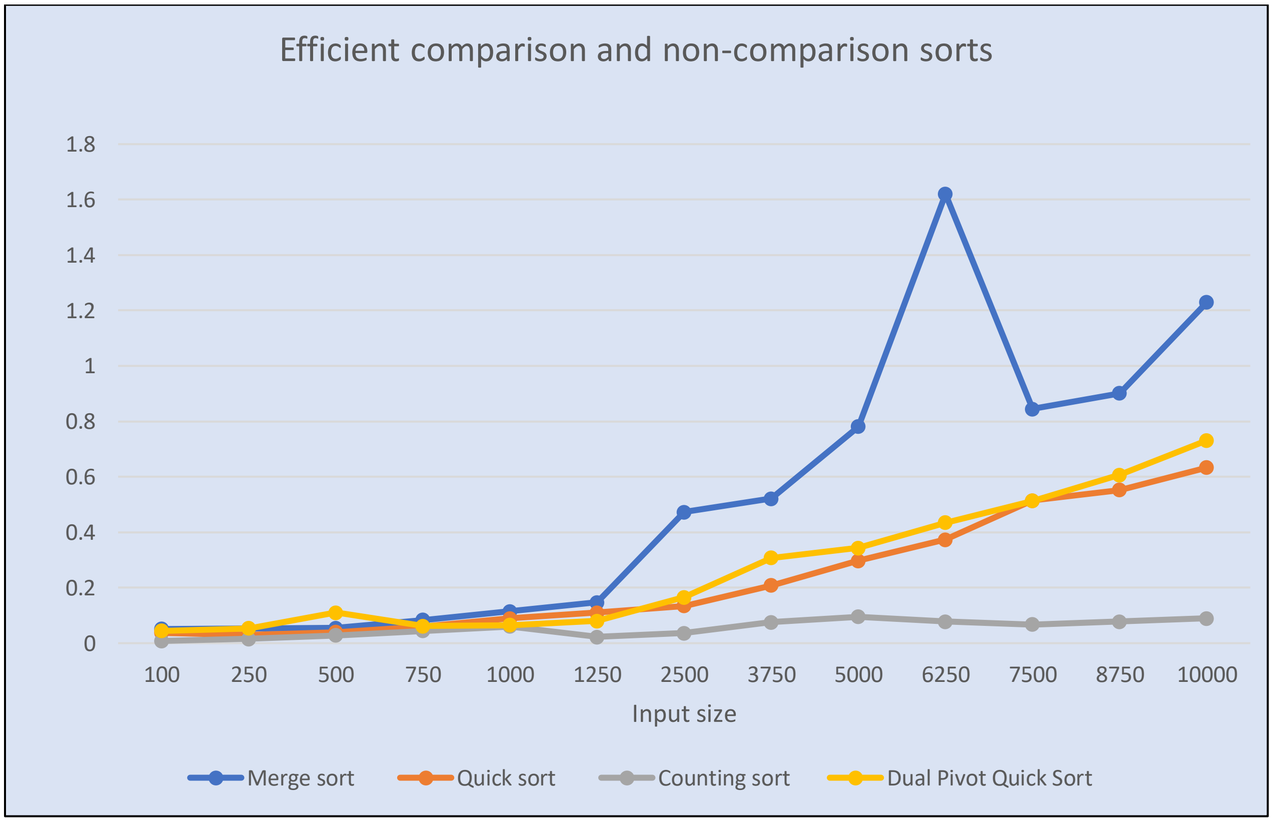
Console output – plugged laptop. Full performance



```
|Size          |100    |250    |500    |750    |1000   |1250   |2500   |3750   |5000   |6250   |7500   |8750   |10000  |
|Bubble sort   |0.232  |0.304  |0.532  |0.68   |1      |1.529  |6.033 ||16.928 |28.258 |47.966 |67.742 |89.572 |121.607|
|Selection sort|0.103  |0.144  |0.274  |0.4    |0.338  |0.419  |1.711  |3.768  |6.97   |11.238 |15.41  |21.158 |27.561 |
|Merge sort    |0.053  |0.06   |0.061  |0.091  |0.118  |0.162  |0.437  |0.784  |0.969  |1.533  |0.758  |0.983  |1.152  |
|Quick sort    |0.028  |0.03   |0.035  |0.06   |0.076  |0.104  |0.127  |0.279  |0.386  |0.341  |0.545  |0.595  |0.765  |
|Counting sort |0.01   |0.022  |0.043  |0.067  |0.057  |0.022  |0.032  |0.054  |0.077  |0.085  |0.188  |0.101  |0.094  |
|Arrays.sort   |0.048  |0.047  |0.105  |0.049  |0.06   |0.098  |0.274  |0.344  |0.322  |0.664  |0.602  |1.011  |1.198  |
```

Graphs


Sorting algorithms performance graph (all sorts)
Running time(milliseconds)
140
120
100
80
60
40
20
0
100
250
500
750
1000
1250
2500
3750
5000
6250
7500
8750
10000
Input size n
Bubble Sort
Selection sort
Merge sort
Quick sort
Counting sort
Dual Pivot Quick Sort


Efficient comparison and non-comparison sorts
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
100
250
500
750
1000
1250
2500
3750
5000
6250
7500
8750
10000
Input size
Merge sort
Quick sort
Counting sort
Dual Pivot Quick Sort

Analysis of algorithms

Simple comparison sorts vs. efficient comparison sorts.

Bubble and Insertion sort were outdone by Merge, Quick and Dual Pivot Quick Sort which is implemented in the *java.util.Arrays* class. The first graph is not even showing any signs that the curves representing efficient comparison sorts are rising. I was expecting this kind of behaviour as this comparison is comparing quadratic time complexity against log base two time complexity of efficient comparison sorts.

Efficient comparison sorts vs. efficient non-comparison sort

The winner of this benchmarking project is the Counting sort which is performing even better than three efficient comparison sorts and its curve is constant regardless of the input size. It is amazing that such a simple logic can sort data so quickly. However, everything comes with a price and as mentioned in the counting sort description, it has some restrictions when it can be used.

Bubble sort vs. Insertion sort

Honestly, this result surprised me the most because as both sorts have quadratic time complexity, I expected that their performance will be similar. Until the input size of 750 values their curves stayed together but eventually the selection sort outdone the bubble sort. This probably depends on the attributes of the field that is being sorted. If it is partially sorted, the insertion sort doesn't have to do as much compared to the bubble sort because many elements might be on their correct or close to correct positions already. On the other side the bubble sort needs to always compare every element to its neighbouring element and even if the elements are swapped, it doesn't mean that they are in their correct position so eventually a lot of shifting needs to be done to sort all elements.

Efficient comparison sorts

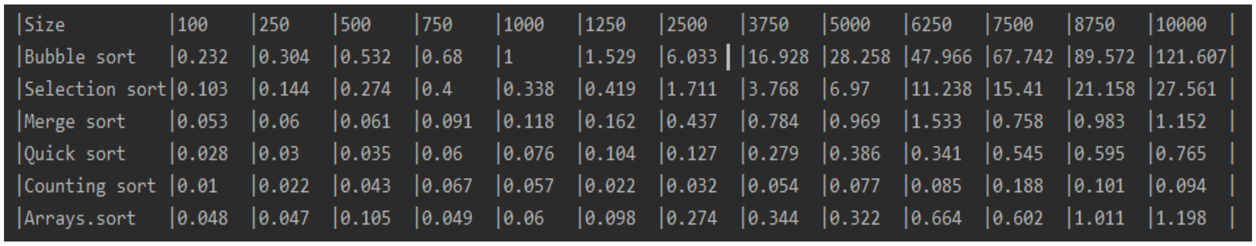
All efficient comparison sorts selected for this project are performing as expected. The Quick sort implementation had almost identical results with Dual Pivot Quick sort from the *java.util.Arrays* class. The reason why Java is using Dual Pivot quick sort instead of regular Quick sort is because of its ability to avoid the worst case of the single pivot Quick sort quadratic behaviour. The Merge sort slightly stayed behind. According to multiple sources, the Quick sort performs better on the small input of primitive types whereas the Merge sort performs better on the larger input of the objects type. This explains why there is a difference between the sorting algorithms embedded in these two Java classes (*java.util.Array and java.util.Collections*).

Reference List

1. Buchalka, T. (n.d.). Sarah Ettrich: Data Structures and Algorithms: Deep Dive Using Java – Counting Sort(Implementation) [video file]. Retrieved from https://www.udemy.com/course/data-structur es-and-algorithms-deep-dive-using-java/learn/lecture/8435794?start=15#overview
2. Buchalka, T. (n.d.). Sarah Ettrich: Data Structures and Algorithms: Deep Dive Using Java –Merge Sort(Implementation) [video file]. Retrieved from https://www.udemy.com/course/data-structures-and-algorithms-deep-dive-using-java/learn/lecture/8435794?start=15#overview
3. Buchalka, T. (n.d.). Sarah Ettrich: Data Structures and Algorithms: Deep Dive Using Java –Quick Sort(Implementation) [video file]. Retrieved from https://www.udemy.com/course/data-structures-and-algorithms-deep-dive-using-java/learn/lecture/8435790#overview
4. Downey, A. B., (2017). Think Data Structure – Algorithms and information retrieval in Java. Sebastopol: O'Reily Media.
5. Mannion, P. (n. d.) Sorting Algorithms Part 1 [PowerPoint presentation]. Retrieved from:https://learnonline.gmit.ie/pluginfile.php/191466/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf
6. Mannion, P. (n. d.) Sorting Algorithms Part 2 [PowerPoint presentation]. Retrieved from: https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf
7. Mannion, P. (n. d.) Sorting Algorithms Part 3 [PowerPoint presentation]. Retrieved https://learnonline.gmit.ie/pluginfile.php/191472/mod_resource/content/0/09%20Sorting%20Algorithms%20Part%203.pdf
8. Rosen, K. H., (2012). Discrete mathematics and its applications, Seventh edition. New York: McGraw-Hill.

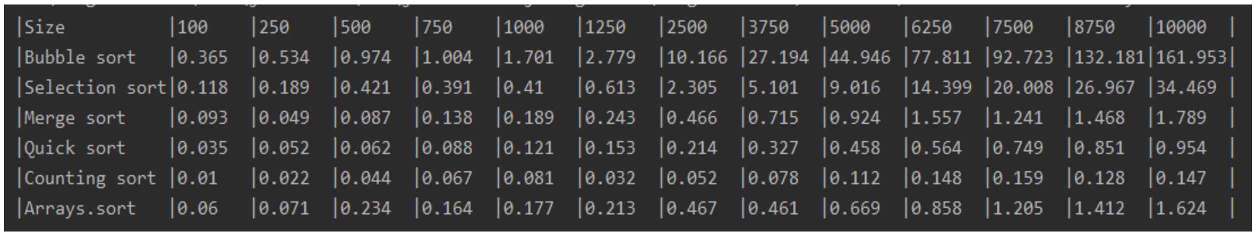
Extras

Console output – plugged laptop. Full performance



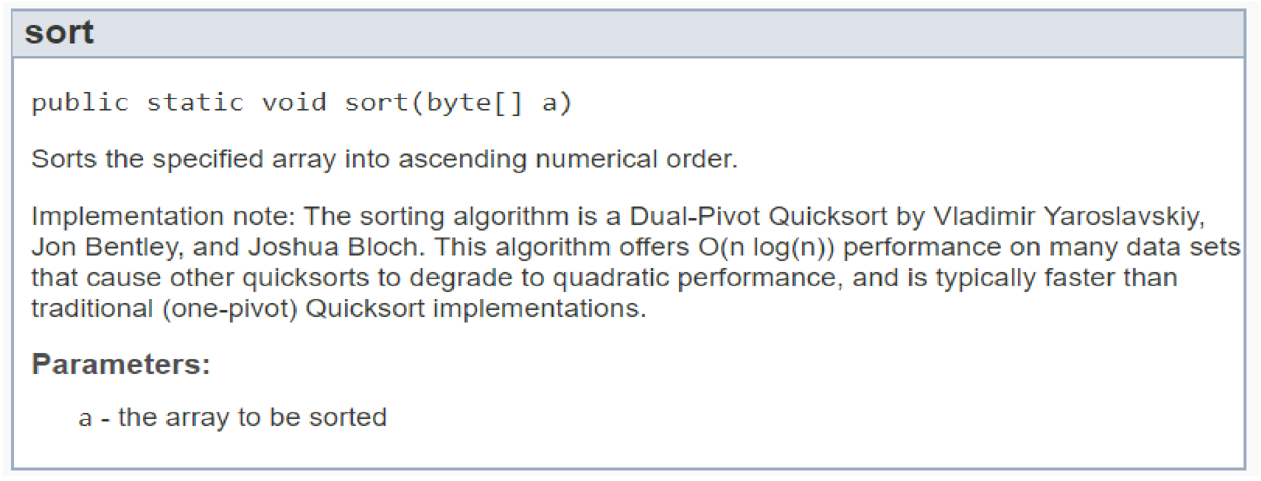
```
|Size          |100   |250   |500   |750   |1000  |1250  |2500  |3750   |5000   |6250   |7500   |8750   |10000  |
|Bubble sort   |0.232 |0.304 |0.532 |0.68  |1     |1.529 |6.033 |16.928 |28.258 |47.966 |67.742 |89.572 |121.607|
|Selection sort|0.103 |0.144 |0.274 |0.4   |0.338 |0.419 |1.711 |3.768  |6.97   |11.238 |15.41  |21.158 |27.561 |
|Merge sort    |0.053 |0.06  |0.061 |0.091 |0.118 |0.162 |0.437 |0.784  |0.969  |1.533  |0.758  |0.983  |1.152  |
|Quick sort    |0.028 |0.03  |0.035 |0.06  |0.076 |0.104 |0.127 |0.279  |0.386  |0.341  |0.545  |0.595  |0.765  |
|Counting sort |0.01  |0.022 |0.043 |0.067 |0.057 |0.022 |0.032 |0.054  |0.077  |0.085  |0.188  |0.101  |0.094  |
|Arrays.sort   |0.048 |0.047 |0.105 |0.049 |0.06  |0.098 |0.274 |0.344  |0.322  |0.664  |0.602  |1.011  |1.198  |
```

Console output – unplugged laptop. In Power mode (Win10)



```
|Size          |100   |250   |500   |750   |1000  |1250  |2500   |3750   |5000   |6250   |7500   |8750   |10000  |
|Bubble sort   |0.365 |0.534 |0.974 |1.004 |1.701 |2.779 |10.166 |27.194 |44.946 |77.811 |92.723 |132.181|161.953|
|Selection sort|0.118 |0.189 |0.421 |0.391 |0.41  |0.613 |2.305  |5.101  |9.016  |14.399 |20.008 |26.967 |34.469 |
|Merge sort    |0.093 |0.049 |0.087 |0.138 |0.189 |0.243 |0.466  |0.715  |0.924  |1.557  |1.241  |1.468  |1.789  |
|Quick sort    |0.035 |0.052 |0.062 |0.088 |0.121 |0.153 |0.214  |0.327  |0.458  |0.564  |0.749  |0.851  |0.954  |
|Counting sort |0.01  |0.022 |0.044 |0.067 |0.081 |0.032 |0.052  |0.078  |0.112  |0.148  |0.159  |0.128  |0.147  |
|Arrays.sort   |0.06  |0.071 |0.234 |0.164 |0.177 |0.213 |0.467  |0.461  |0.669  |0.858  |1.205  |1.412  |1.624  |
```

Dual Pivot Quicksort description in java.util.Arrays class from Javadocs.



sort

```
public static void sort(byte[] a)
```

Sorts the specified array into ascending numerical order.

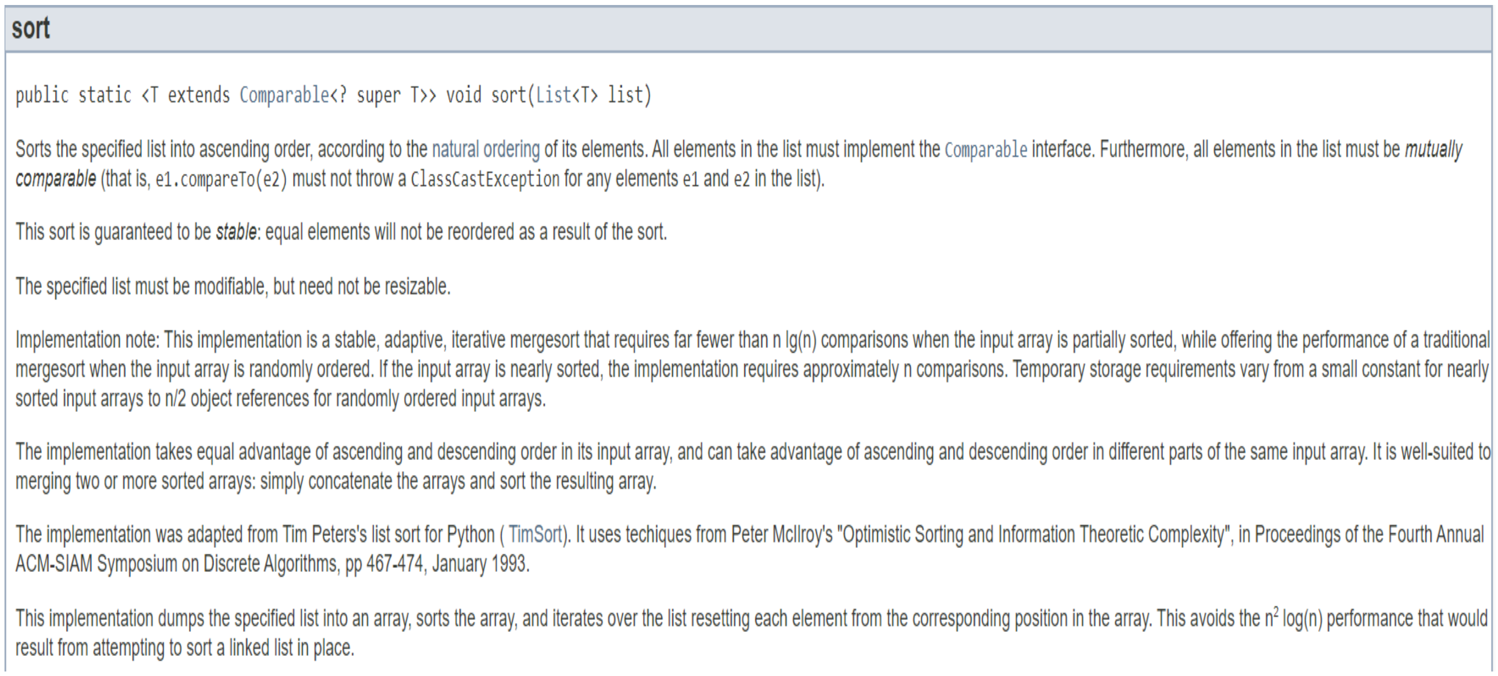
Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Mergesort description in java.util.Collections class from Javadocs.



sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be ***mutually comparable*** (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

This sort is guaranteed to be ***stable***: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the n^2 log(n) performance that would result from attempting to sort a linked list in place.

https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html