GMIT Higher Diploma in Data Analytics 2021/2022
Caoimhin Vallely
G00398568

Computational Thinking with Algorithms Project

Benchmarking Sorting Algorithms Report

Introduction

In the context of computer science, sorting is the process of ordering and arranging data to a particular set of criteria. In this project I will be concentrating on numeric values but the concepts apply equally to most common types of data including web search engine results, filtering of financial transactions, and finding addresses in a phone book. Much of the early development of computing concerned sorting and even to this day, as sorting still takes up a significant portion of general CPU cycles, research in the area and attempts to further optimise algorithms continues. Understanding sorting in terms of numbers is quite simple in that the numbers should be in order, i.e. a number should be lower (or equal) to the number following it. The sorted collection must be a permutation, i,e, contain the same contents before and after sorting. In terms of text based data, the information may be sorted alphabetically while for time data, chronological sorting may be used.

An inversion is a change in the order of two pieces of data. For example the list [9,11,10] would require one inversion to be sorted (11,10), whereas the list [3,8,1,9,6] would require four - (8,1), (3,1), (9,6) and (8,6). This leads us to one of the most important aspects of sorting – running time. Obviously the more inversions required to sort a list, the longer it will take. For the example lists of 3 and 5 numbers, the time differences are minimal, but when we approach much larger datasets of many thousands, the issue becomes more serious, and is the reason why such time and resources have been invested in the optimisation of sorting algorithms.

The simplest sorting algorithms are comparison based, i.e. each element is compared with its neighbour and switched if necessary. The easiest to understand of these is bubble sort. We will discuss it in more detail later in this report but basically it involves comparing each pair of numbers in turn until all of the elements are in the correct order. All comparison sorting algorithms are a variation of this in that they can only progress by comparing two elements at a time. Thus the best case performance result can only be *n* log *n* as every element has to be compared at least once. Non-comparison sorting algorithms on the other hand can sometimes improve on this for certain types of data. We will look at bucket sort later but another popular example is counting sort which is based on the frequency that unique elements appear in a list. There are also hybrid versions which use combinations of comparison and non-comparison based algorithms.

While some sorting algorithms are certainly better than others, there is no single one that suits all situations. Each has its own strengths and weaknesses and some perform better depending on the circumstance or type and size of data. In general though, there are certain criteria which make a sorting algorithm more desirable – stability, good run-time efficiency, and non-excessive use of extra memory.

Stability refers to the reordering of already sorted elements in an unordered list. A stable sorting algorithm guarantees this property whereas an unstable one may alter elements already considered equal.

Run time efficiency of a sorting algorithm is judged on best-case, worst-case and average case performance. Other factors may also be an influence such as the size and type of the data to be sorted, degree to which it is already pre-sorted, and whether or not the data can be manipulated in internal computer memory or external.

In-place refers to the use of a fixed amount of working space for the algorithm to work irrespective of the input size. Those that use in-place sorting may be more appropriate if memory is an issue when dealing with a large quantity of input data.

The following is an overview of ten common algorithms.

Algorithm	Best case	Worst case	Average case	Space complexity	Stable
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	Yes
Quicksort	$n \log n$	n^2	$n \log n$	n	No
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	n x k	Yes
Timsort	n	$n \log n$	$n \log n$	n	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No

Adapted from 'Sorting Algorithms Part 1' – lecture slides - Patrick Mannion GMIT

The Algorithms

Bubble sort is one of the more simple sorting algorithms, and although it is not very practical in many real world situations, it is a good starting point for understanding a lot of the concepts inherent in sorting. It was first analysed in 1956 and was named for the way the larger values 'bubble up' to the surface. It is a comparison based and in-place sorting algorithm and as can be seen above has a time complexity of n in the best case and n^2 in the worst and average cases. The procedure involves starting with the first item in the list, comparing it to its neighbour to the right, and if it's larger, swap it. We move on to the 2nd and 3rd items then and do the same, and continue until we get to the end of the list with the largest value. The process is then repeated until we arrive at the 2nd last item, and so on until the list is in order. The following diagrams explain the process. The list is: [10, 8, 2, -4, 15, 1, 20, 11]

Pass 1:

	10	8	2	-4	15	1	20	11
1	8	10	2	-4	15	1	20	11
2	8	2	10	-4	15	1	20	11
3	8	2	-4	10	15	1	20	11
4	8	2	-4	10	1	15	20	11
5	8	2	-4	10	1	15	11	20

We begin by comparing 10 and 8 and switching, then 10 and 2 and switching, then 10 and -4 and switching. 10 and 15 are in the correct order so we move on to 15 and 1 and switch. 15 and 20 are in the correct order so we move on to 20 and 11 and switch. This provides the starting point then for pass 2, and subsequently passes 3 and 4 until we arrive at the sorted list.

Pass 2:

	8	2	-4	10	1	15	11	20
1	2	8	-4	10	1	15	11	20
2	2	-4	8	10	1	15	11	20
3	2	-4	8	1	10	15	11	20
4	2	-4	8	1	10	11	15	20

Pass 3:

	2	-4	8	1	10	11	15	20
1	-4	2	8	1	10	11	15	20
2	-4	2	1	8	10	11	15	20

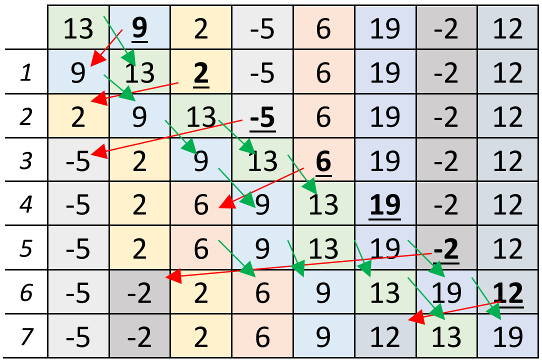
Pass 4:

	-4	2	1	8	10	11	15	20
1	-4	1	2	8	10	11	15	20

Insertion Sort is another relatively simple sorting algorithm which is stable and operates in-place. It works best with small lists and those that are close to being sorted, although it is much less efficient with larger random lists.

It operates in a similar fashion to bubble sort by comparing neighbouring pairs of numbers, but instead of just moving values one space at a time and then comparing the next 2 numbers, we can move elements multiple positions. We begin at index [1] (2nd element in list) – this is designated the **key**. If the number to its left is greater we move it to the right. The next value at index [2] then becomes the key – if the elements to the left are greater, we move them to the right. The number at index [2] now may become the first element. This continues until we reach the value at the last index and the list is sorted.

The diagram below demonstrates a sort on the list [13, 9, 2, -5, 6, 19, -2, 12]



	13	**9**	2	-5	6	19	-2	12
1	9	13	**2**	-5	6	19	-2	12
2	2	9	13	**-5**	6	19	-2	12
3	-5	2	9	13	**6**	19	-2	12
4	-5	2	6	9	13	**19**	-2	12
5	-5	2	6	9	13	19	**-2**	12
6	-5	-2	2	6	9	13	19	**12**
7	-5	-2	2	6	9	12	13	19

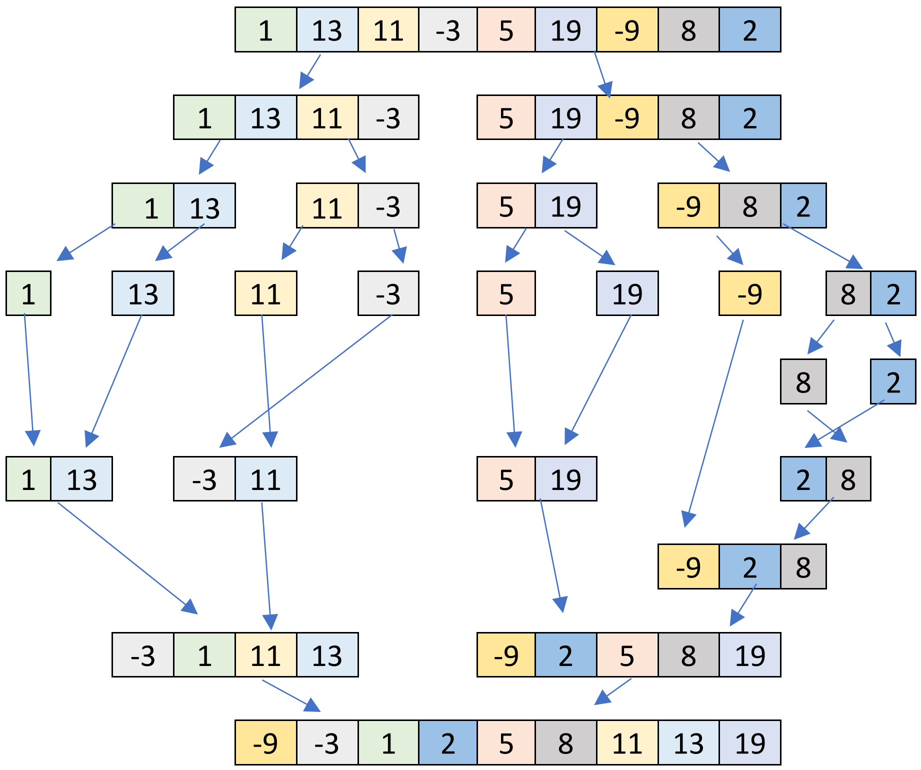
We set the key as the element at index [1] which is **9**. As the number to its left is larger, it moves to the right and 9 becomes the first element. Index[2] – **2** – then becomes the key – as the two values to its left are greater, they both move to the right and 2 becomes the first element. Index[3] then becomes the key – as the 3 values below it are greater, they move to the right and -4 becomes the first element. We continue like this until we reach the element at index [n-1] where n represents the length of the list.

Even visually this appears a much more efficient approach as it only takes 1 'pass' and 6 'inversions' compared to the 4 passes and 12 inversions for bubble sort above. Insertion sort can achieve similar performance to bubble with *n* in the best case and n^2 in the worst and average cases.

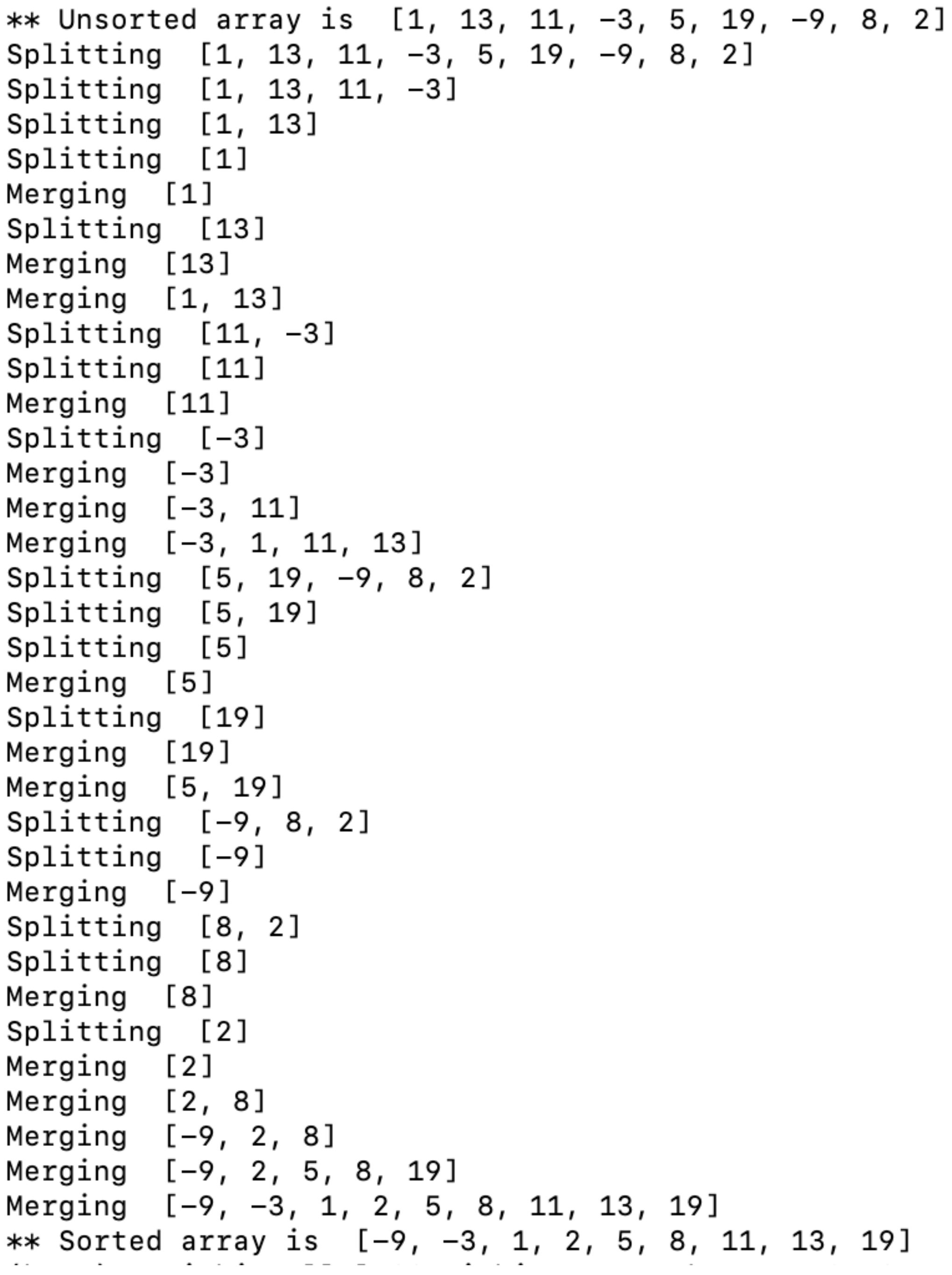
Merge Sort is a much more efficient comparison based sorting algorithm. It utilises a 'divide-and-conquer' approach and its performance in the best, worst, and average cases is quite similar (*n log n*). It was first proposed by John von Neumann in 1945.

The process begins with the dividing. The list is repeatedly divided into 2 equal or near equal sized lists until each list size reaches 1 or 0. These mini-lists' are essentially now sorted and it is now a matter of merging them all back together.

The following is an example using the list: [1, 13, 11, -3, 5, 19, -9, 8, 1]



The list is divided into 2 nearly equal parts of 4 and 5. These are then subdivided into groups of 2 or 3 and finally groups of 1. The groups are gradually merged and sorted until we arrive at the final sorted list. From my implementation and with the help of some strategic print statements we can see the python version in action here:



```
** Unsorted array is  [1, 13, 11, -3, 5, 19, -9, 8, 2]
Splitting  [1, 13, 11, -3, 5, 19, -9, 8, 2]
Splitting  [1, 13, 11, -3]
Splitting  [1, 13]
Splitting  [1]
Merging  [1]
Splitting  [13]
Merging  [13]
Merging  [1, 13]
Splitting  [11, -3]
Splitting  [11]
Merging  [11]
Splitting  [-3]
Merging  [-3]
Merging  [-3, 11]
Merging  [-3, 1, 11, 13]
Splitting  [5, 19, -9, 8, 2]
Splitting  [5, 19]
Splitting  [5]
Merging  [5]
Splitting  [19]
Merging  [19]
Merging  [5, 19]
Splitting  [-9, 8, 2]
Splitting  [-9]
Merging  [-9]
Splitting  [8, 2]
Splitting  [8]
Merging  [8]
Splitting  [2]
Merging  [2]
Merging  [2, 8]
Merging  [-9, 2, 8]
Merging  [-9, 2, 5, 8, 19]
Merging  [-9, -3, 1, 2, 5, 8, 11, 13, 19]
** Sorted array is  [-9, -3, 1, 2, 5, 8, 11, 13, 19]
```

Quicksort is also a recursive ‘divide-and-conquer’ algorithm. It was developed by C. A. R. Hoare in 1959 and is one of the fastest algorithms on average. It runs at *n log n* in the best and average cases but slips to n^2 in the worst case. It can also be unstable although modifications can mitigate against this.

The process begins with the choosing of a pivot. All of the other elements are then arranged in relation to the pivot, i.e. those lower than the pivot move to the left of it and those greater move to the right of it. This results in the pivot being in its final position. The same process is then applied to each of the sections either side of the pivot, and continues until the list is sorted. The efficiency of the algorithm can depend heavily on the choice of pivot. Common pivots include the first or last element, a random element, or the median element. The process by how we find the final position of the pivot, or ‘split point’, is the main focus of the algorithm. If we take the first element to be the pivot, we want to position it, so that all of the elements to its left are lower and those to the right are greater. We start with the start (leftmark) and end point (rightmark) of the remainder of the list. We move the leftmark to the right until we meet a value which is greater than the pivot point. We then move the rightmark to the left until it meets a value which is lower than the pivot point. These two elements now swap. We continue this until leftmark and rightmark cross – this is the split point. The quicksort can now be carried out recursively on both halves until the list is fully sorted. The following diagram demonstrates a quicksort on the list [24, 19, 92, 30, 74, 5, 50, 81, 23]

	24	19	92	30	74	5	50	81	23	
		L							**R**	
Left	**24**	19	92	30	74	5	50	81	23	19<24 move to right
			L						**R**	92>24 stop
Right	**24**	19	92	30	74	5	50	81	23	23<54 stop
			L						**R**	
Swap	**24**	19	23	30	74	5	50	81	92	swap 23 and 92
			L						**R**	
Left	**24**	19	23	30	74	5	50	81	92	23<24 move to right
			L						**R**	30>24 stop
Right	**24**	19	23	30	74	5	50	81	92	92>24 move to left
				L					**R**	81>24 move to left
										50>24 move to left
										5<24 stop
Swap	**24**	19	23	30	74	5	50	81	92	swap 5 and 30
				L		**R**				
Left	**24**	19	23	5	74	30	50	81	92	5<74 stop
				L		**R**				
Right	**24**	19	23	5	74	30	50	81	92	74>24 move to left
					L	**R**				5<24 stop
										23<24 stop
Cross	**24**	19	23	5	74	30	50	81	92	left and right have crossed so split point is where right la
				R	**L**					

Split	19	23	5	**24**	74	30	50	81	92	24 is now in the correct position

19	23	5	**74**	30	50	81	92	Repeat with the two new sub-lists

We choose the first element (24) as the pivot point. The element to its right (19) and the last element (23) will be the leftmark and rightmark respectively. 19 is less than 24 so we move the right. 92 is greater than 24 so we stop. We focus on the rightmark next. 23 is less than 24 so we stop, and exchange 92 and 23. 23 and 92, in their new positions, are now leftmark and rightmark. We start the process again. 23 is less than 24 so we move to the right. 30 is greater than 24 so we stop. On to the right again – 92, 81, and 50 are all greater than 24 but 5 is less so we stop. We swap 5 and 30. These are now our leftmark and rightmark. We repeat the process until leftmark and rightmark cross. At this stage the final position of rightmark becomes the split point to which we move the pivot.

Quicksort gets in to difficulty when a list is already or nearly sorted as its performance deteriorates rapidly if the pivot point chosen ends up being close in value to the minimum or maximum element of the list. To avoid this, rather than choose the first or last index of the list as the pivot, we can choose 3 values – maybe first and last and middle - and then choose the median of the three.

Bucket sort is a (mostly*) non-comparison based sorting algorithm. The basic premise is that a number of buckets representing value ranges are created, into which each of the corresponding elements is placed. These buckets are then sorted using either another sorting algorithm (*possibly comparison-based) or further divided up using bucket sort recursively. They are then merged back together again in the correct order. In the best case the list is distributed uniformly and each bucket contains an even number of elements (*n* + *k*), while in the worst case all of the elements are contained in one bucket and the performance drops to n^{2}.

The running time may also depend on which sorting algorithm is used for the final sort.

The diagram below demonstrates a bucket sort for the array [17, 5, 38, 24, 9, 1, 21, 14, 11, 3, 32, 19]

17	5	38	24	9	1	21	14	11	3	32	19

scatter	5, 1, 3	9	14, 11	17, 19	24, 21		32	38
buckets	1 to 5	6 to 10	11 to 15	16 to 20	21 to 25	26 to 30	31 to 35	36 to 40

sort	1, 3, 5	9	11, 14	17, 19	21, 24		32	38
buckets	1 to 5	6 to 10	11 to 15	16 to 20	21 to 25	26 to 30	31 to 35	36 to 40

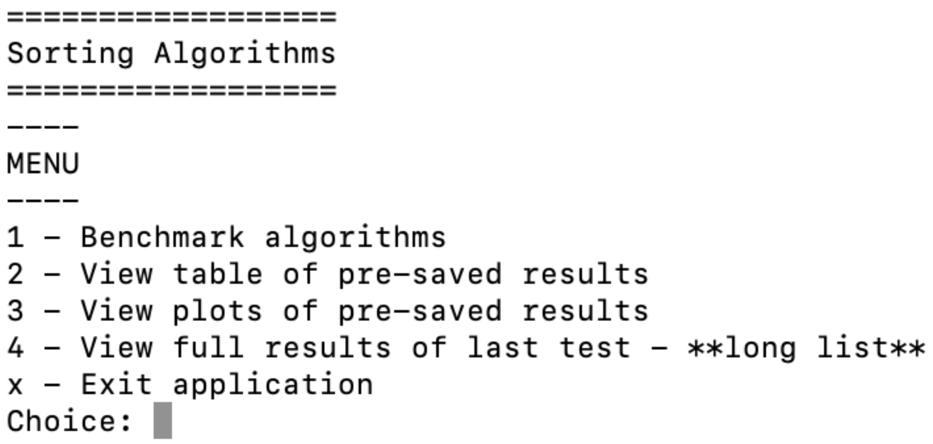
gather	1	3	5	9	11	14	17	19	21	24	32	38

We create 8 buckets with ranges of multiples of 5 and 'scatter' the values into their appropriate buckets. We sort the element within each bucket, and then 'gather' them all together.

Implementation

The Code for each of the algorithms is contained in the module ***algos.py***. There are many versions of the algorithms available online but these particular ones were adapted from https://runestone.academy/ns/books/published/pythonds/SortSearch/sorting.html (bubble, insertion, merge, quicksort) and https://www.geeksforgeeks.org/bucket-sort-2/ (bucket). I've changed variable names and added comments to aid my own understanding of each.
The ***benchmarking.py*** file contains the main program which creates the arrays, runs them through the algorithms, records and benchmarks the results, and plots visualisations of the results. The program imports the ***algos.py*** module and a module for plotting different result options (***plots.py***). To run the program, it is assumed you have a python environment already set up. The folder ***G00398568.zip*** needs to be downloaded and unzipped. On the command prompt, navigate to the folder where it has been saved, and type the command *python benchmarking.py*. From there you can follow the menu options. The code is heavily commented so I won't repeat all of the workings here but rather give an overview of how the program runs and then a discussion on the results.. Sources for individual elements of the code are referenced in the code comments and at the end of this report, but just to mention a few key elements in the code were adapted from:
https://www.angela1c.com/projects/cta_benchmarking/ctabenchmarkingproject#Python-Benchmarking-application-code.
The program is structured around a *for* loop with iterates through each of the five algorithms. Nested within that are two further *for* loops which iterate through each array size, and the specified numbers of runs – which in this case is ten. Each run is timed using the *time* module and the results saved to a *pandas* dataframe. We calculate the average of each of the 10 runs and this is returned in a table form and a plot using the *matplotlib.pyplot* module.

When the program is run, the user is presented with a menu of 4 choices:



```
==================
Sorting Algorithms
==================
----
MENU
----
1 - Benchmark algorithms
2 - View table of pre-saved results
3 - View plots of pre-saved results
4 - View full results of last test - **long list**
x - Exit application
Choice: 
```

Option 1 contains the main bulk of the program. On entering, the user is presented with a submenu which lets them decide between random arrays of relatively low or high values, ordered arrays (ascending and descending), or random arrays with normal distribution. The last few options where included to illustrate the weaknesses of particular algorithms.
Option 1 of the submenu creates random arrays of between 10 and 1,000 values while option 2 creates arrays of between 500 and 20,000. ** N.B. option 2 will take a long time to run as bubble and insertion sort struggle with very high inputs. For each length of array, 10 different versions are created, and each is run through each of the five algorithms. The average is then calculated and that is the figure presented in the results table and used for the plot which both get printed on execution. Options 2, 3, and 4 operate in a similar manner using different types of inputs and different sizes of array.

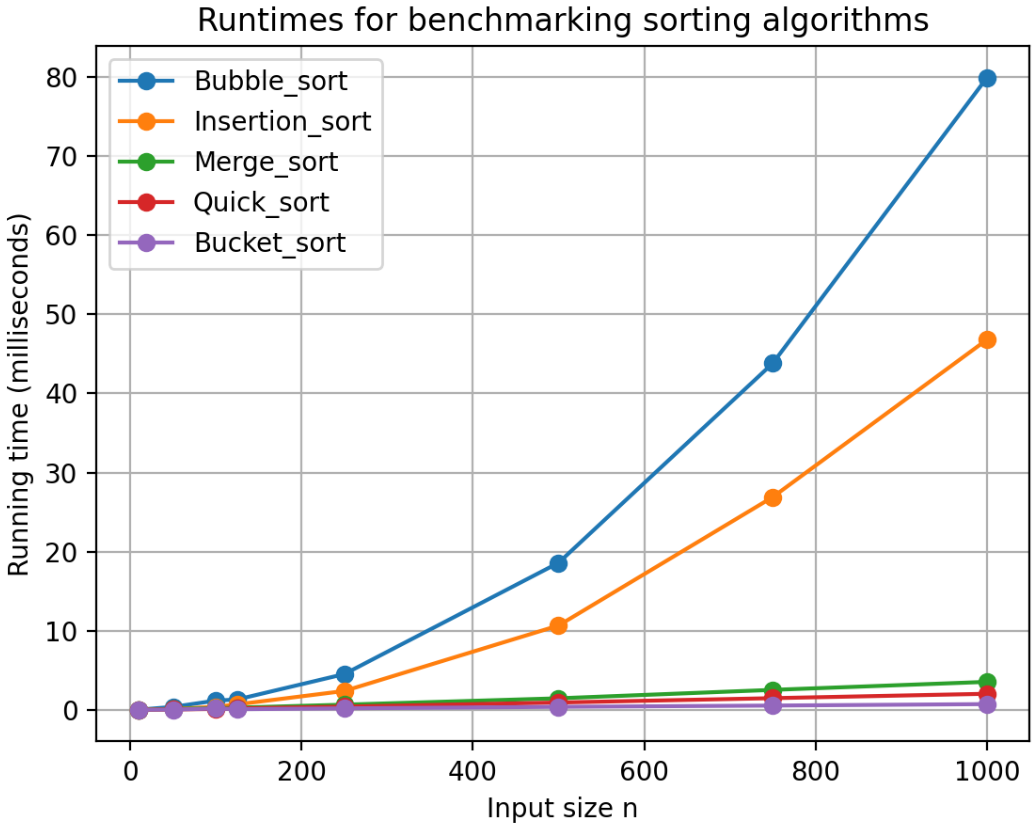
From the main menu, the user also has the option to view tables and plots of results from the previous time the program was run. For the plots they can also choose to view several of the algorithms in isolation. This is useful as the plot from the high input test is difficult to appreciate as the disparity in performance is quite huge.
There is also an option to view the full results including times, for every single run the last time the program was run.

Results

The following table and plot are the output from running option 1.

Array size	10	50	100	125	250	500	750	1000
Bubble sort	0.028	0.432	1.23	1.352	4.429	18.628	44.45	82.062
Insertion sort	0.009	0.103	0.35	0.541	2.44	10.415	24.377	45.485
Merge sort	0.019	0.101	0.204	0.251	0.566	1.214	2.114	2.768
Quicksort	0.008	0.046	0.11	0.14	0.328	0.812	1.659	1.941
Bucket sort	0.02	0.035	0.06	0.077	0.147	0.307	0.477	0.647

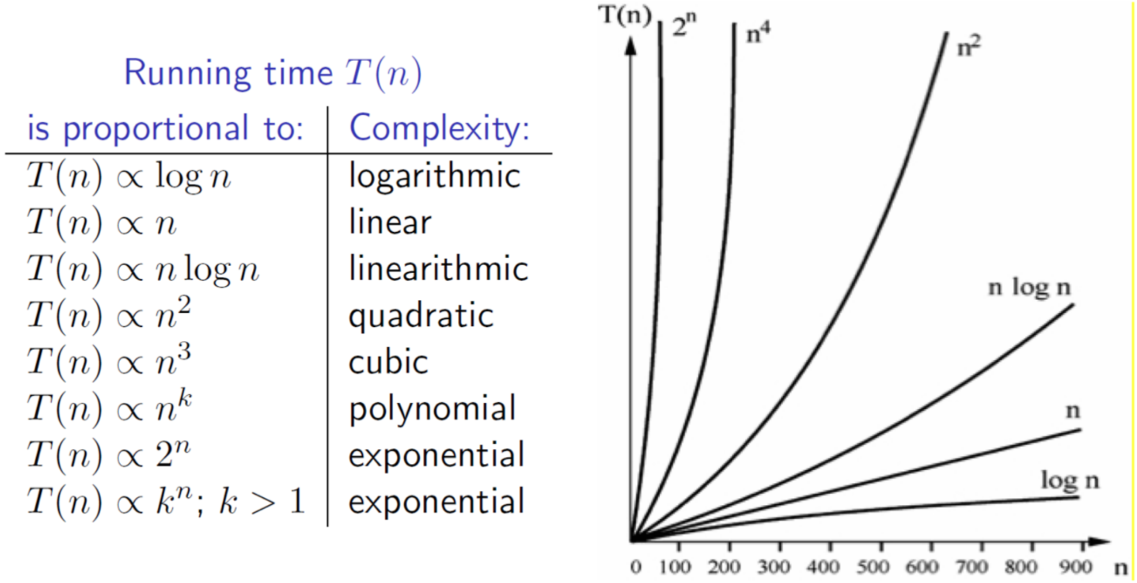
Average runtimes for random integer arrays of between 10 and 1,000



Plot of average runtimes for random integer arrays of between 10 and 1,000

We can see that the figures are broadly what we expected. The figures may differ from other published versions but the relative relationships between the results are consistent. The algorithms performed in the order they were expected with bubble being least efficient and bucket sort the most. We can see that there isn't a huge amount between them for the lower size arrays but the gap widens significantly the higher the input.

If we look at the plot and compare it to the following growth chart, we can see that bubble and insertion sort are, as anticipated, roughly reflecting quadratic growth (n^2) while the other three algorithms are closer to linearithmic growth (*n log n*) or better.



Running time $T(n)$

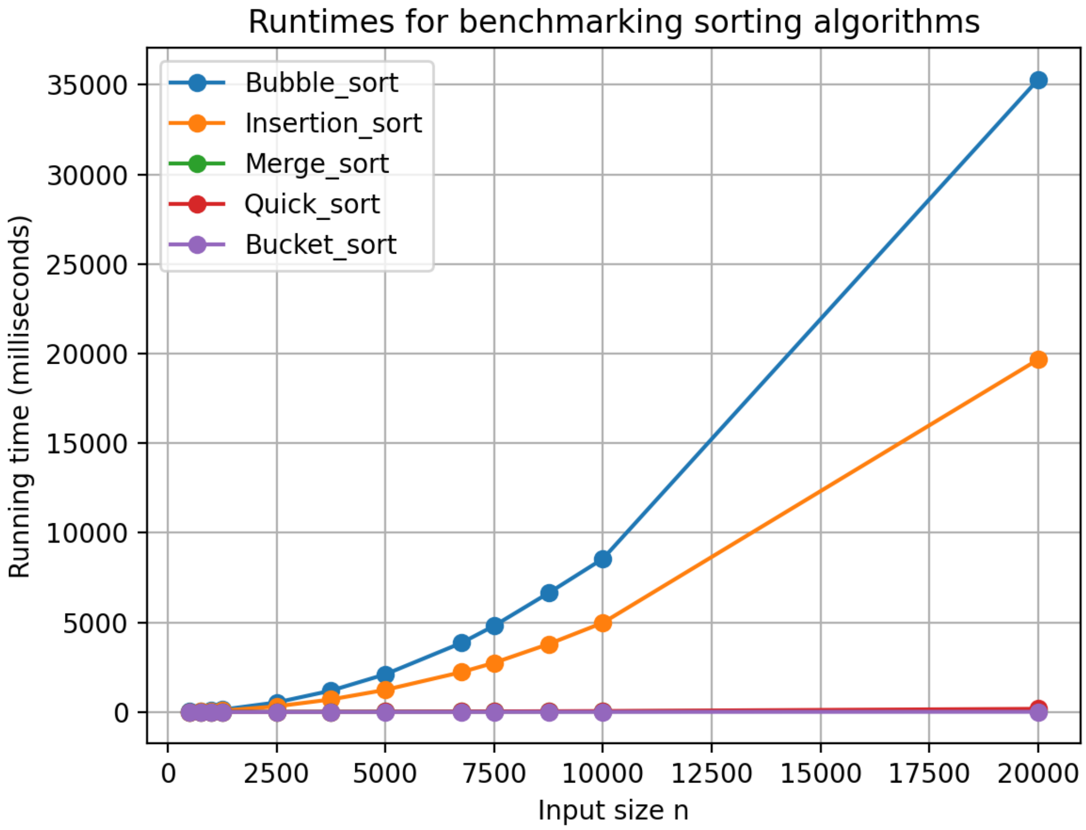
is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential

Taken from 'Sorting Algorithms Part 1' – lecture slides - Patrick Mannion GMIT

When we increase the input sizes we can see this pattern accentuated further.

Array size	1000	1250	2500	3750	5000	6750	7500	8750	10000	20000
Bubble sort	77.953	128.82	535.495	1185.534	2085.762	3911.44	5078.667	6860.179	8683.876	34510.863
Insertion sort	46.054	72.9	300.377	673.276	1210.67	2165.289	2708.771	3884.993	4958.266	19571.281
Merge sort	2.862	3.774	8.699	13.59	18.167	25.063	29.796	33.953	38.656	82.927
Quicksort	1.888	2.55	6.373	11.224	17.215	27.561	32.385	43.196	52.884	177.516
Bucket sort	0.657	0.82	1.64	2.508	3.304	4.668	5.219	5.791	6.69	13.632

Average runtimes for random integer arrays of between 1,000 and 20,000



Plot of average runtimes for random integer arrays of between 1,000 and 20,000

One notable feature though is that once the inputs go over a certain size, merge sort begins to outperform quicksort.

When we use **ordered** arrays, things change dramatically. The following results are from a benchmarking test on ordered arrays ascending.

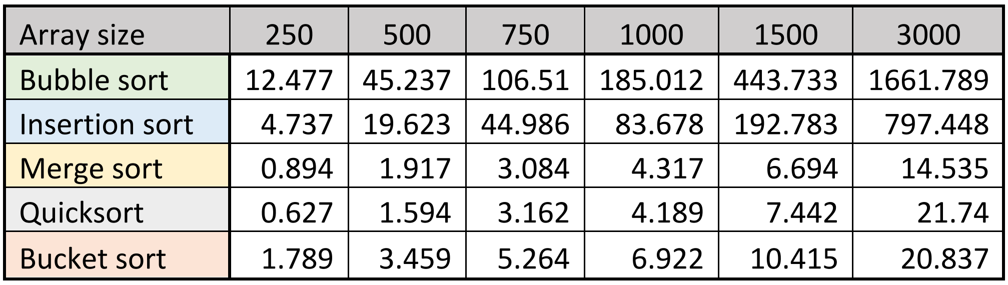
Array size	250	500	750	1000	1500	3000
Bubble sort	4.032	11.493	27.473	49.13	113.669	483.296
Insertion sort	0.052	0.078	0.114	0.157	0.232	0.468
Merge sort	0.598	1.177	1.947	2.666	4.098	9.606
Quicksort	2.632	13.864	25.321	43.64	101.524	386.709
Bucket sort	0.311	0.34	0.471	0.628	0.929	1.925

Average runtimes for ordered integer arrays of between 250 and 3,000

We can see that Quicksort has serious difficulties. This is because my version of the algorithm is using the first index as the pivot. In an ordered array this is the lowest value, which means the split point is always going to be to the extreme left leaving the algorithm to work to its maximum to complete the sort. To counter this I could have used an alternative method for choosing the pivot as suggested above.

The other notable finding here is that insertion sort is the best performing algorithm, even outperforming bucket sort.

For the random arrays above I used the *randint* function from python’s *random* module. This returns arrays which are based on a uniform distribution. As an experiment I tried creating arrays with *numpy.random*, based on a normal distribution, to see how that affected things. The following are the results:



Array size	250	500	750	1000	1500	3000
Bubble sort	12.477	45.237	106.51	185.012	443.733	1661.789
Insertion sort	4.737	19.623	44.986	83.678	192.783	797.448
Merge sort	0.894	1.917	3.084	4.317	6.694	14.535
Quicksort	0.627	1.594	3.162	4.189	7.442	21.74
Bucket sort	1.789	3.459	5.264	6.922	10.415	20.837

Average runtimes for random integer arrays (normally distributed) of between 250 and 3,000

We can see for the first time that bucket sort is now struggling somewhat. This is because with the normal distribution most of the elements are going to be concentrated in one bucket, which, as has been mentioned would be the worst case scenario for that particular algorithm.

Conclusion

We can see clearly that no single algorithm can suit every scenario, so it is important to properly assess the inputs in terms of size and type, before choosing the most suitable algorithm. Small lists or lists that are pre-sorted to a degree may suit insertion sort, while for larger lists that are uniformly distributed, bucket sort might be the most appropriate. It should be noted as well that the machine these tests are carried out on does affect the results, although the relative behaviour should remain the same.

References

Adamsmith.haus. 2022. *Kite*. [online] Available at: <https://www.adamsmith.haus/python/answers/how-to-print-an-entire-pandas-dataframe-in-python>.

Angela1c.com. 2022. *CTA Benchmarking Project Report final*. [online] Available at: https://www.angela1c.com/projects/cta_benchmarking/ctabenchmarkingproject#Python-Benchmarking-application-code.

Codewolfy.com. 2022. *Bucket Sort Program in Python*. [online] Available at: https://www.codewolfy.com/python/example/bucket-sort-in-python.

"Computational Thinking with Algorithms" - lecture notes by Patrick Mannion and Dominic Carr at GMIT

Cp.eng.chula.ac.th. 2022. [online] Available at: <https://www.cp.eng.chula.ac.th/~vishnu/datastructure/QuickSort.pdf>.

DataScience Made Simple. 2022. *Reshape using Stack() and unstack() function in Pandas python - DataScience Made Simple*. [online] Available at: <https://www.datasciencemadesimple.com/reshape-using-stack-unstack-function-pandas-python/>.

Docs.python.org. 2022. *random — Generate pseudo-random numbers — Python 3.10.4 documentation*. [online] Available at: <https://docs.python.org/3/library/random.html>.

GeeksforGeeks. 2022. *Bucket Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/bucket-sort-2/>.

GeeksforGeeks. 2022. *Insertion Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/insertion-sort/>.

numpy, h., 2022. *numpy, how to generate a normally distributed set of integers*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/33160367/numpy-how-to-generate-a-normally-distributed-set-of-integers>.

Programiz.com. 2022. *Bucket Sort (With Code in Python, C++, Java and C)*. [online] Available at: <https://www.programiz.com/dsa/bucket-sort>.

Runestone.academy. 2022. *6.6. Sorting — Problem Solving with Algorithms and Data Structures*. [online] Available at: <https://runestone.academy/ns/books/published/pythonds/SortSearch/sorting.html>.

En.wikipedia.org. 2022. *Sorting algorithm - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Sorting_algorithm>.

What is the maximum recursion depth in Python, a., Wouters, T. and Young, D., 2022. *What is the maximum recursion depth in Python, and how to increase it?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/3323001/what-is-the-maximum-recursion-depth-in-python-and-how-to-increase-it>.