Introduction:

Sorting is any method that can take a collection of items that is unorder and reorder them in a certain order. An algorithm is a set of steps that achieve a predefined goal. A sorting algorithm therefore is just a set of steps that that sorts a collection of items. This report investigates the use of five different computational algorithms. It will investigate time and space complexity of each. Where time complexity is the length of time it takes to execute, and space complexity is the space needed for it (for different input sizes). These two factors will indicate the overall performance of an algorithm.

When categorizing computational sorting algorithms, they are either in-place or out of place, are stable or instable and whether it is comparison or non-comparison based. When using an in-place algorithm there is no extra space required. Stable sorting means that same value entries stay in the same order. “Comparison based algorithms use comparator functions which take in two values and compare the two directly. Non-Comparison based functions are the rest. These can include counting sort which sorts using key value, radix sort which examines individual bits of keys, and bucket Sort which also examines bits of keys.” [14]

Sorting Algorithms:

Processes:

1. Insertion Sort:

The insertion sort algorithm takes in a list of integers. It then iterates through the list starting with the second element. It compares it with the element before it and if it is greater than it swaps it. It then compares it with the rest of the list down along. Once the whole list is compared it moves on to the third element and so on.

2	53	20	89	35	50	68	28	30	100	4	82	47	1	73	22	27
	←															
2	53	20	89	35	50	68	28	30	100	4	82	47	1	73	22	27
		←														
2	20	53	89	35	50	68	28	30	100	4	82	47	1	73	22	27
	←															
2	20	53	89	35	50	68	28	30	100	4	82	47	1	73	22	27
			←													
2	20	53	89	35	50	68	28	30	100	4	82	47	1	73	22	27
				←												
2	20	53	35	89	50	68	28	30	100	4	82	47	1	73	22	27
			←													
2	20	35	53	89	50	68	28	30	100	4	82	47	1	73	22	27
		←														
2	20	35	53	89	50	68	28	30	100	4	82	47	1	73	22	27
					←											

Table 1: Showing Insertion sort in action.

Here we can see a list of number. Using insertion sort, we take the second element (53) and compare it to the element to its left (or at the previous index). In this case the second element is larger than the first (2) so no swap occurs. Moving on to the third element (20) we compare it to the

second element. In this case the comparison shows us that the third is smaller than the first, so a swap occurs. It then goes on to compare itself with the first and is greater, so no swap occurs. When we reach the fifth element (35) we can see that it swaps twice, once to become the fourth and again to become the third. This is the comparison used in insertion sort.[1]

2. Quick Sort:

The quick sort algorithm takes in a list of integers. It uses a pivot point that can be chosen at random or depending on the data set. Here we have chosen the first element as the pivot point. What quick sort then does is move every other element to the left or right depending whether it's less or greater than pivot value. It will then do the same for either side of the pivot point upon completion. Meaning for all the elements less than the original pivot point it will choose the first element among them as a new pivot point. The same happens for the elements greater the original pivot point.

68	53	20	89	35	50	2	28	30	100	4	82	47	1	73	22	27
68	53	20	89	35	50	2	28	30	100	4	82	47	1	73	22	27
68	53	20	89	35	50	2	28	30	100	4	82	47	1	73	22	27
68	53	20	27	35	50	2	28	30	100	4	82	47	1	73	22	89
68	53	20	27	35	50	2	28	30	100	4	82	47	1	73	22	89
.	.	.	.	.	.	.	.									
68	53	20	27	35	50	2	28	30	100	4	82	47	1	73	22	89
68	53	20	27	35	50	2	28	30	22	4	82	47	1	73	100	89
.	.	.	.	.	.	.	.									
68	53	20	27	35	50	2	28	30	22	4	82	47	1	73	100	89
68	53	20	27	35	50	2	28	30	100	4	1	47	82	73	100	89
47	53	20	27	35	50	2	28	30	100	4	1	68	82	73	100	89

Table 2: Quick Sort in action

This is the first pass with 68 (the first element) being the first pivot point. Next insertion sort will take everything to the left and sort them this way and the same with the right. Again, and again until the array is sorted. [2]

3. Bucket Sort:

Bucket sort takes in the unsorted array and you set the size of buckets you'll need. The buckets start as empty arrays.

To determine the number of buckets needed (or the number of indexes needed inside the buckets array) you find the difference between the minimum and maximum values and divide it by the set bucket size. Round to lowest integer and add one (possibly could just round to the highest integer

haven't had time to try). The same process is used to find the index of each number by taking difference of each element to the maximum value and again dividing by the bucket size. Basically, by getting the difference between max and min you can set a base value. That base value is then separated into equal parts (each bucket) by the bucket size. By then putting each element through the same procedure you can guarantee that elements of similar range are kept together.

This means that there are a number of buckets created and each element is place into their respective bucket. A selection sort is then used on the smaller arrays. These smaller sorted arrays are then added together.

This method is called divide and conquer. You divide the list into smaller ones and sort them and add them back together. [3]

4. Radix Sort:

Radix Sort is a non-comparison-based sorting algorithm. It organises digit by digit. A counting sort is used to organise these digits. Since we use a base ten number system the range is from zero to nine (ten in total) for each digit. Starting at the power of one and building up by multiplying itself by 10 each time. So, one would be position one (0 to 9), ten would be position two (0 to 9 on the second digit of 10 to 99) and so on until all the digits are in order.

2	4	6	,	3	4	8	,	0	3	4	,	8	7	1	,	0	0	9	,	0	5	8	,	9	9	9
		↑				↑				↑				↑				↑				↑				↑
8	7	1	,	0	3	4	,	2	4	6	,	3	4	8	,	0	5	8	,	0	0	9	,	9	9	9
	↑				↑				↑				↑				↑				↑				↑	
0	0	9	,	0	3	4	,	2	4	6	,	3	4	8	,	0	5	8	,	8	7	1	,	9	9	9
↑				↑				↑				↑				↑				↑				↑		
0	0	9	,	0	3	4	,	0	5	8	,	2	4	6	,	3	4	8	,	8	7	1	,	9	9	9

Table 3: Radix Sort in action

Shown above it becomes clearer. The first digits of each element are used to sort the elements, then the second and so on until you're left with a sorted list. This method although not comparison based is stable. [4]

5. Timsort:

Timsort is a hybrid algorithm employing a combination of Merge and Insertion sort. First (a bit like bucket sort) it combines elements into arrays of ascending order. These separate arrays of ascending order are then merged together to form the sorted array. [5, 6, 7 & 8]

68	53	20	89	35	50	2	28	30	100	4	82	47	1	73	22	27

Given the above list, the timsort algorithm would execute the following steps:

Step 1: It will separate either into smaller ascending lists or lists on its own as follows.

Runs = [[68], [53], [20, 89], [35, 50], [2, 28, 30, 100], [4, 82], [47], [1, 73], [22, 27]]

Step 2: Merge these smaller parts together.

→ [68] + [53] → [53, 68] → [53, 68] + [20, 89] → [20, 89] (temp) + [53, 68]

→ [89] (temp) + [20, 53, 68] → [20, 53, 68, 89] → [20, 53, 68, 89] + [2, 28, 30, 100]

→ [2, 28, 30, 100] (temp) + [20, 53, 68, 89] → [28, 30, 100] (temp) + [2, 20, 53, 68, 89]

→ [30, 100] (temp) + [2, 20, 28, 53, 68, 89] → [100] (temp) + [2, 20, 28, 30, 53, 68, 89]

→ [2, 20, 28, 30, 53, 68, 89] . . . etc.

Time and space complexity:

Below shows a table with information on the five sorting algorithms chosen for this report. For each it shows the best, worst and average case for the time complexity, it's space complexity and whether or not the algorithm is stable or not. Where n is the size of the input and k is the number of inversions needed.

Algorithm	*Best case*	*Worst case*	*Average case*	*Space Complexity*	*Stable*
Insertion Sort	n	n^2	n^2	1	Yes
Quick Sort	n log(n)	n^2	n log(n)	O(n)	No
Bucket Sort	n + k	n^2	n + k	n X k	Yes
Radix Sort	n X k	n X k	n X k	n + k	Yes
Timsort	n	n log(n)	n log(n)	n	Yes

Table 4: Time and space complexities for each algorithm [11]

1. Insertion Sort:

Time complexity is a very useful way of predicting the efficiency of an algorithm. It is done by analysing the algorithm itself. Take insertion sort for example. It's best case scenario when it comes to time complexity is n. Meaning that it is equal to about the size of the input. What must be noted here is that this only occurs when the list is already completely sorted. As this case is highly unlikely in the real world the average case seems much more likely. Where it will compare every element in the list with every other one hence n^2. Which means that it is exponential to the size of the input. [11]

The space complexity however is 1 which means no extra space is needed. Looking at Table 1 its clear to see why. The same space is used and only two elements are being swapped at a time.

Finally, this algorithm is stable as when the current element under investigation is compared with an element of the same value they are not swapped.

2. Quick Sort:

With quick sort the worst case is the same but the average and best case are both equal to n log(n). Meaning for example that for an input size of 1250 the average time complexity case of insertion sort is around 1,562,500 and of quick sort is around 3,871. This is a considerable difference when you consider that by increasing the input size this difference will continue to increase.

Quick sort has a worst case of n^2 is caused when the split points may be off to one side. This means that the split is very uneven and larger splits will have to be done.

This algorithm is not stable as there is no telling where a split happened and how equal values are placed. [11]

3. Bucket Sort:

The bucket and radix sort algorithms are affected by the number of inversions (k) when it comes to time complexity. As you can see for the average case the radix is considerably higher then the bucket. This is due to radix multiplying the input size by number of inversions (n X k) versus adding for the bucket (n + k). Bucket sort depends on how many buckets are empty at the end of filling them.

The space complexity for bucket however is expensive. Due to creating new arrays to hold smaller lists the storage required increases greatly at a space complexity of n X k. [10]

4. Radix Sort:

Due to the method radix sort uses its time is the same for worst, best and average cases. Even if a list is sorted radix will go through its steps regardless.

It is a stable algorithm because it works through each element one by one. [9, 11]

Implementation & Benchmarking:

This section will include notes on the application and how to improve the accuracy of the results. The python application for each algorithm is heavily commented for explanation. It is designed with convince in mind in that you simply have to run the script. There is no input required.

Note: The following module imports will be required.

- from random import randint
- from tabulate import tabulate [12]
- import math
- import time

The first thing requiring more investigation is for bucket sort. Specifically, the bucketSize variable. With some testing an effective bucketSize could be obtained for the range of input size. Either that or as the input size increases the bucketSize could increase by some function. [13]

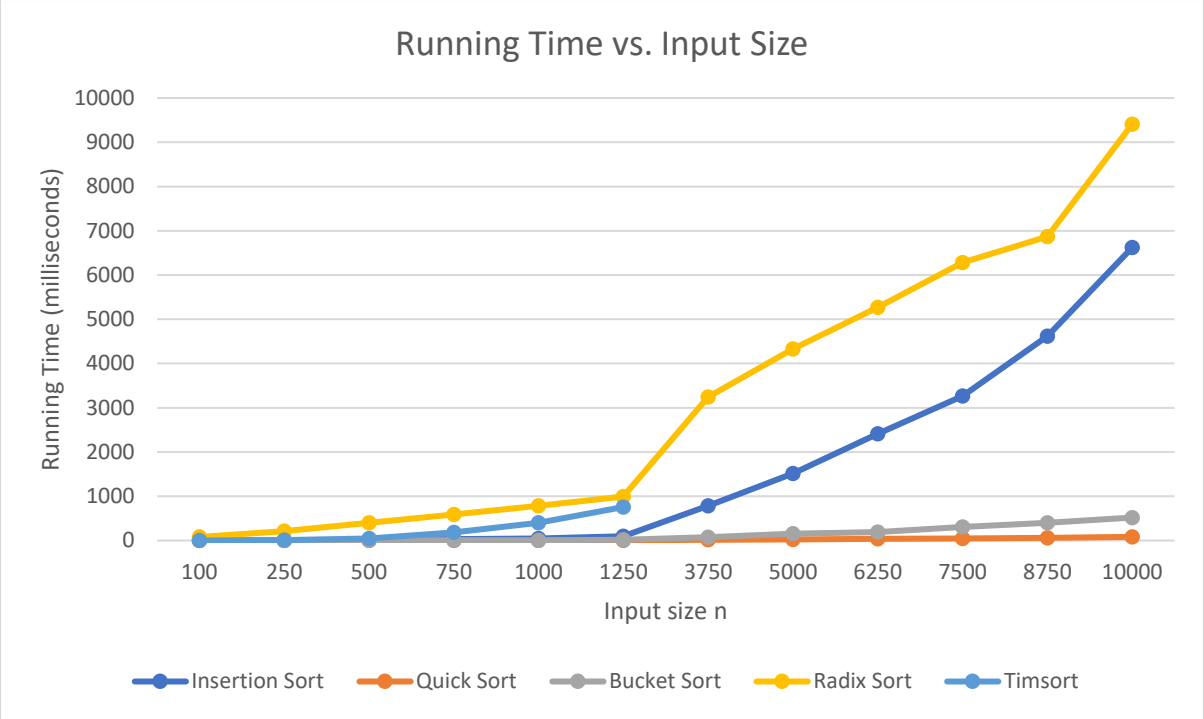
I would start by testing for a single input size and various bucket sizes starting with 100. I would choose 10, 50, 100, 500 and 1,000 as bucket sizes. Then I would test input sizes 5,000, and 10,000 with the same bucket sizes and investigate for a correlation from bucket size to performance.

For both bucket and Timsort I would try and implement not only collecting increasing arrays but decreasing arrays and then reverse them before merging.

Finally, I would use a more capable machine. For the timsort algorithm the computer I was using reached its recursion limit at input size 1250.

Results:

Going on time complexities for the average case (because best is highly unlikely to ever occur from random generated lists) by order of consistency you would expect bucket (n + k), quick (n logn) and timsort (n logn) to be similar being very close to linear. You would expect Insertion (n^2) to be the worst performing due to being exponential larger with input size. Finally, you would expect Radix (nk) to fall under insertion but about the others because it reliant on the number of inversions (k).



From these results we can clearly see that radix sort is the poorest performer out of any of the algorithms. This is since radix sort time complexity increases proportional to the increase in input size and the number of inversions.

Second to that is insertion sort. Insertion sorts best case is rarely going to happen as it's when the list is already sorted or just n. Instead in the real-world application it is exponential to the input size and so the running time will increase as we increase our input size.

Now might be the time to point out the sort sample size for timsort. Recursion would happen on anything with an input size over 1250. I'm not sure if this is due to poor design on my part or if this is normal. It is performing faster than radix sort up to this point but after an input size of over around 500 the others start to out perform it. Obviously more investigations must be done in order to determine how it performs on larger inputs.

Both quick sort and bucket sort perform educationally well up to an input size of about 1250. After this point the quick sort out performs bucket but in comparison to the other sorts not by much. This is due to the n log(n) time complexity.

The figures below show a significant difference from radix to any other algorithm even at the lowest input size. Quick sort clearly lives up to its name and would be interesting to see the results from testing out different starting pivot points.

Size	100	250	500	750	1000	1250	3750	5000	6250	7500	8750	10000
Insertion Sort:	0.447	2.766	16.4	30.478	49.645	97.302	784.508	1513.48	2414.18	3264.51	4615.72	6619.15
Quick Sort:	0.222	0.665	1.552	2.549	3.435	4.544	17.869	26.922	39.092	48.423	64.17	83.554
Bucket Sort:	0.332	0.665	1.995	3.546	6.095	15.625	78.797	152.28	188.126	309.049	397.621	519.009
Radix Sort:	79.796	210.149	402.364	591.441	782.132	991.126	3241.76	4332.59	5267.36	6280.15	6871.65	9406.65
Timsort:	0.886	7.65	50.001	180.53	399.148	757.105						

Table 5: Running time (in milliseconds) each algorithm at different input sizes (n).

References:

1. interactivepython. 2019. The Insertion Sort. [ONLINE] Available at: http://interactivepython.org/courselib/static/pythonds/SortSearch/TheInsertionSort.html. [Accessed 1 May 2019].

2. interactivepython. 2019. The Quick Sort. [ONLINE] Available at: http://interactivepython.org/courselib/static/pythonds/SortSearch/TheQuickSort.html. [Accessed 3 May 2019].

3. geeksforgeeks. 2019. Bucket Sort. [ONLINE] Available at: https://www.geeksforgeeks.org/bucket-sort-2/. [Accessed 6 May 2019].

4. geeksforgeeks. 2019. Radix Sort. [ONLINE] Available at: https://www.geeksforgeeks.org/radix-sort/. [Accessed 8 May 2019].

5. hackernoon. 2019. Timsort—the fastest sorting algorithm you've never heard of. [ONLINE] Available at: https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399. [Accessed 10 May 2019].

6. medium. 2019. This is the fastest sorting algorithm ever. [ONLINE] Available at: https://medium.com/@george.seif94/this-is-the-fastest-sorting-algorithm-ever-b5cee86b559c. [Accessed 11 May 2019].

7. medium. 2019. This is the fastest sorting algorithm ever. [ONLINE] Available at: https://medium.com/@george.seif94/this-is-the-fastest-sorting-algorithm-ever-b5cee86b559c. [Accessed 11 May 2019].

8. dev. 2019. Timsort: The Fastest sorting algorithm for real-world problems. [ONLINE] Available at: https://dev.to/s_awdesh/timsort-fastest-sorting-algorithm-for-real-world-problems--2jhd. [Accessed 11 May 2019].

9. codingpointer. 2019. Python Program for Radix Sort. [ONLINE] Available at: https://www.codingpointer.com/python-tutorial/radix-sort. [Accessed 12 May 2019].

10. growingwiththeweb. 2019. Bucket sort. [ONLINE] Available at: https://www.growingwiththeweb.com/2015/06/bucket-sort.html. [Accessed 15 May 2019].

11. P.Mannion (2019). Sorting Algorithms Lecture 3, 4 &5. Analysing Algorithms. Galway- Mayo Institute of Technology.

12. pypi.org. 2019. tabulate 0.8.3. [ONLINE] Available at: https://pypi.org/project/tabulate/. [Accessed 17 May 2019].

13. P.Mannion (2019). Sorting Algorithms Lecture 11. Benchmarking Algorithms. Galway- Mayo Institute of Technology.

14. javarevisited. 2019. Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) based Sorting Algorithms? Read more: https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html#ixzz5oPMiNSsN. [ONLINE] Available at: https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html. [Accessed 18 May 2019].