Report Computational Thinking

Introduction

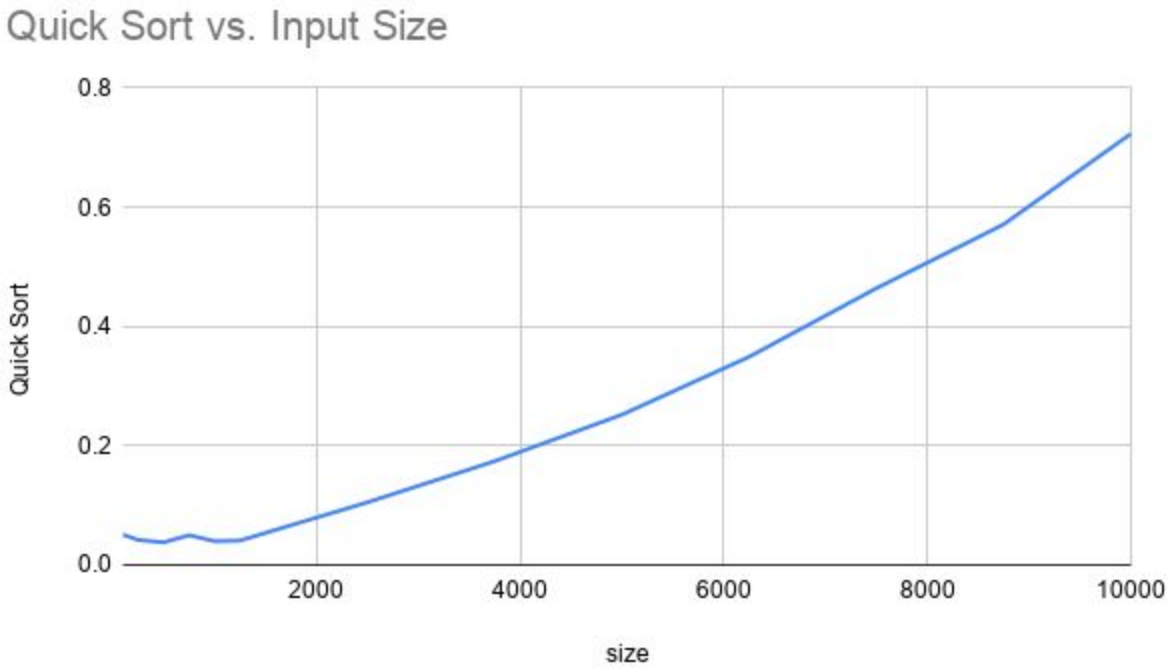
In this report I will outline the concept of sorting and sorting algorithms. I will discuss the relevance of time and space complexity, performance, in place sorting, stable sorting, comparator functions, comparison based and non comparison based sorts.

Sorting is the operation of arranging sets of data in a specific order. This technique is carried out to rearrange the data of a table or a list in accordance with criteria of one form or another. There are two categories of sorting techniques, internal sorting, where the sorted data can be adjusted at a time in the main memory and external sorting, where the information can not be held in the memory at a time and needs to be kept in auxiliary memory i.e. hard disks.
Sorting algorithms are employed to reorder array or list elements in accordance to a comparison operator used on the elements, which makes a decision of what the new order of the elements will become in the structure of the data. When analyzing an algorithm, space and time complexity is considered.
Space complexity quantifies the amount of space or memory that is being used by an algorithm to run as a function of the length of the input. Similar to this, the time complexity of an algorithm quantifies the amount of time that is taken by an algorithm to run as a function of the length of the input. The performance of an algorithm depends on the input size of the array or list. The performance of the algorithm can be examined by the best, worst and average cases which show how the algorithm uses resources at least, most and on average. In-place sorting refers to sorting without any space requirement. An algorithm that uses in place sorting transforms input using no auxiliary data structure. Stable sorting refers to algorithms where if the elements are considered equal they will follow the previous order, meaning, the order of equal elements is guaranteed to be preserved.
Comparator functions are functions that take arguments and contain logic in order to decide the relative order of sorted output. Algorithms that have this function are called comparison based sorts. Examples of such sorting algorithms are Quicksort, Merge sort, Heap sort and Tim Sort. Sorting algorithms that are comparison sorts i.e. comparing pairs of values, can never achieve a worst case time better than O(N log N). Sorting algorithms that are not comparison based can have better worse- case times. Such algorithms are Bucket sort, Counting sort and Radix sort.

Sorting Algorithms

Quick Sort

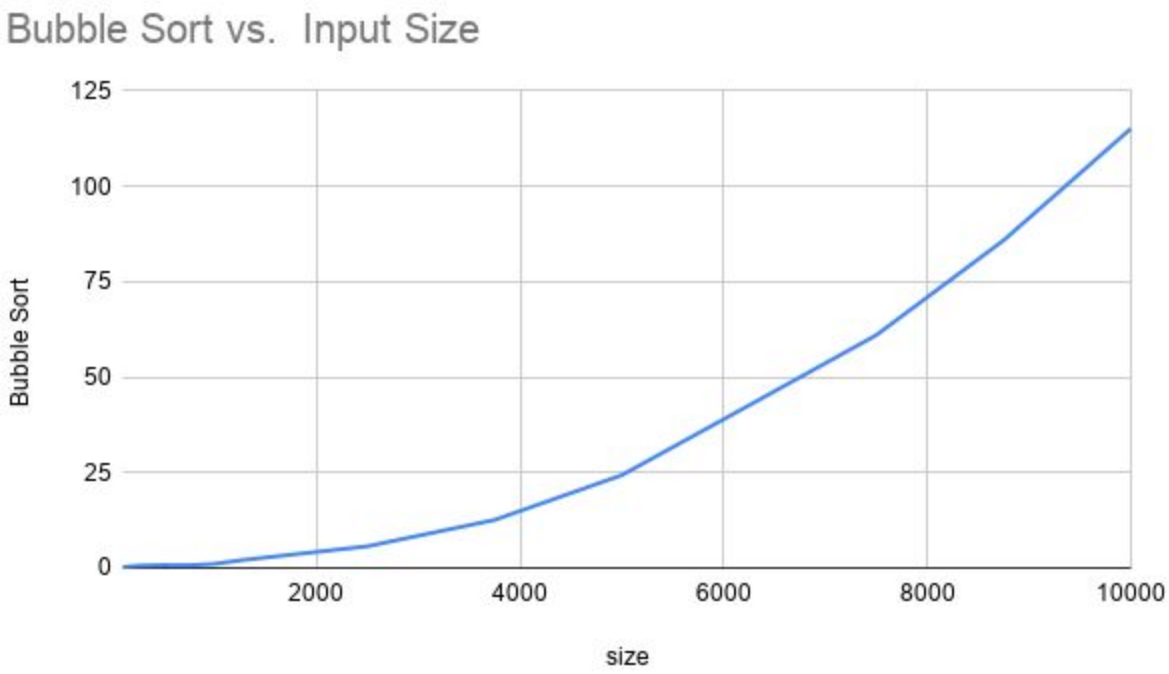
Quick Sort is an efficient divide and conquer algorithm. It's worst case complexity is n^2. The average is n*log(n) and the best is n*log(n). It operates by using a method of partitioning where It uses a pivot element, which is selected from the array and partitions the other elements into two sub arrays depending on if they are less than or greater than the value of the pivot. The sub arrays are sorted using recursion. It is an unstable comparison sort.



To measure this sorting algorithm, I used the input sizes :
100 250 500 750 1000 1250 2500 3750 5000 6250 7500 8750 10000.
As can be seen in the above graph, quicksort behaves as expected with linearithmic growth from the offset.

Bubble sort

Bubble sort is a simple comparison sort algorithm. It has a worst complexity of n^2 and an average complexity of n^2. The best complexity is n. It uses a partitioning method where elements are compared and sorted and appear to bubble to the top.

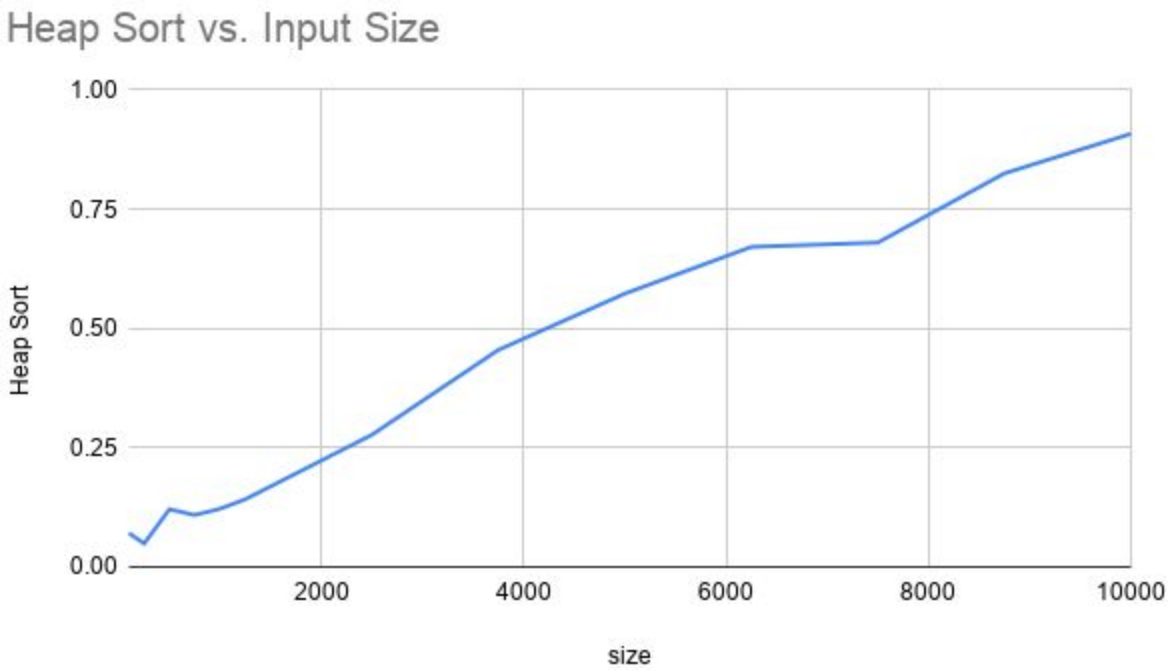


To measure this sorting algorithm, I used the input sizes :
100 250 500 750 1000 1250 2500 3750 5000 6250 7500 8750 10000.
As can be seen in the graph, it initially begins as n which is the best case for this algorithm, but eventually begins to behave quadratically as the input sizes increase in keeping with the best and worst cases i.e. n^2.

Heap Sort

Heap sort is an unstable comparison based sorting algorithm. the worst complexity is n*log(n) and the average is n*log(n). The best complexity like both of the previous is n*log(n). It has a space complexity of 1. The method it uses is selection which entails two steps. Firstly a heap is built out of the data. The heap is placed in an array with a binary tree layout which maps the

binary tree structure into the indices, with each array index representing a node. Secondly, the largest element is removed from the heap and inserted into an array. When everything is removed from the heap the end result is a sorted array.

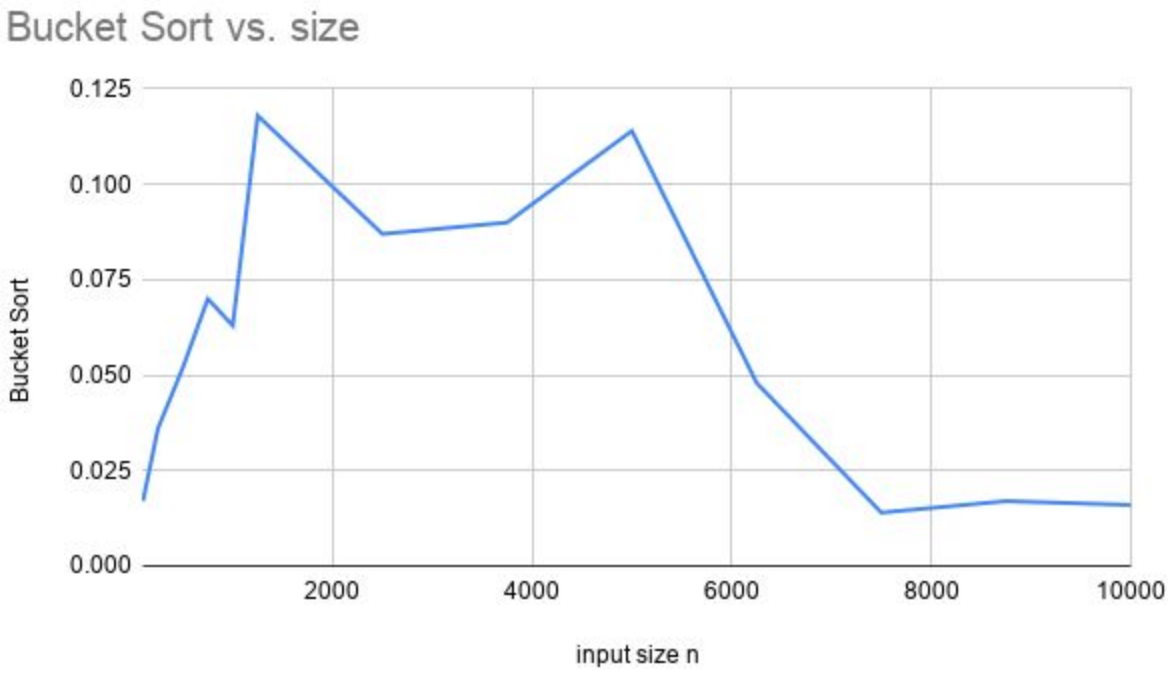


To measure this sorting algorithm, I used the input sizes :
100 250 500 750 1000 1250 2500 3750 5000 6250 7500 8750 10000.

The unstable nature of this algorithm can be seen in the above graph. Best , worst and average cases are all n*log(n). It does appear to grow in a linearithmic fashion albeit in an unstable way as the input sizes become larger.

Bucket Sort

Bucket sort is a comparison sort algorithm. It has a worst case performance of O(n2) and an average case of n+k. It also has a best case of n+k. This sort operates by sorting the elements of an array into buckets and then sorting each bucket individually.

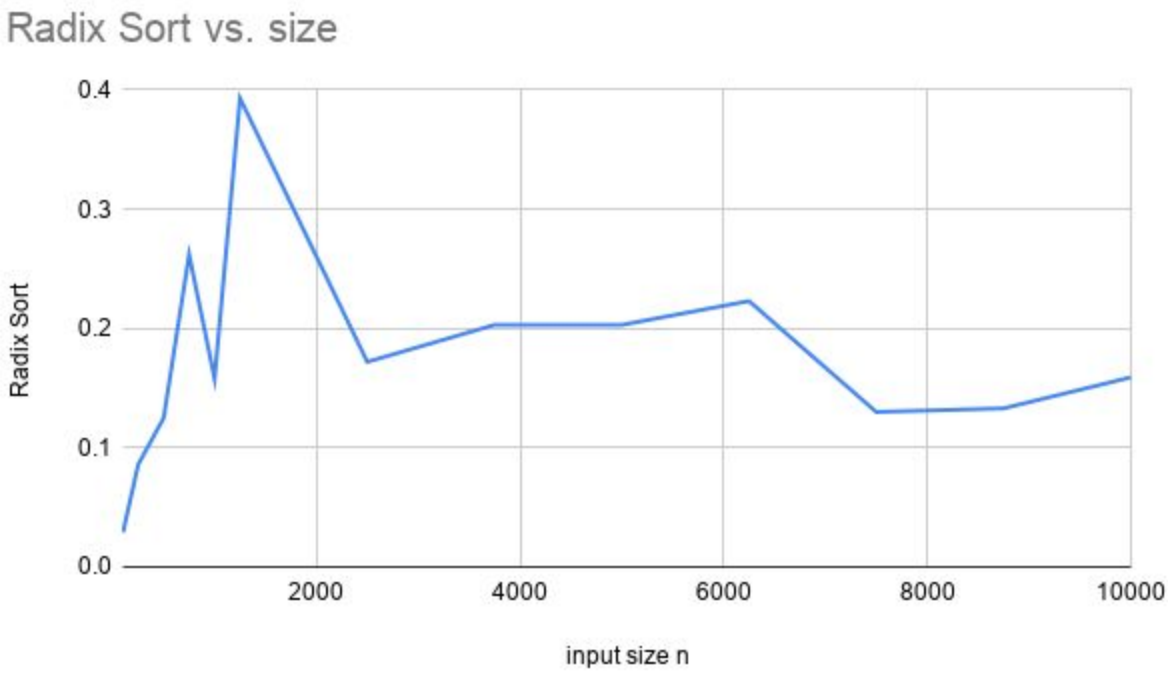


To measure this sorting algorithm, I used the input sizes :
100 250 500 750 1000 1250 2500 3750 5000 6250 7500 8750 10000.

The graph above illustrates how initially the running time complexity is exponential when handling the smaller input sizes but gradually becomes logarithmic when using the larger input sizes. This could be due to the distribution of the input sizes , it does not behave in a predic

Radix Sort

Radix sort is a non comparative sort. The complexity for the worst and average is n*k/d. The best case is O(k * n). Radix sort creates and distributes elements into buckets according to their radix. Each element that has more than one significant digit, the process repeats for each of those digits while maintaining the order of the previous step. This continues until all digits have been sorted.

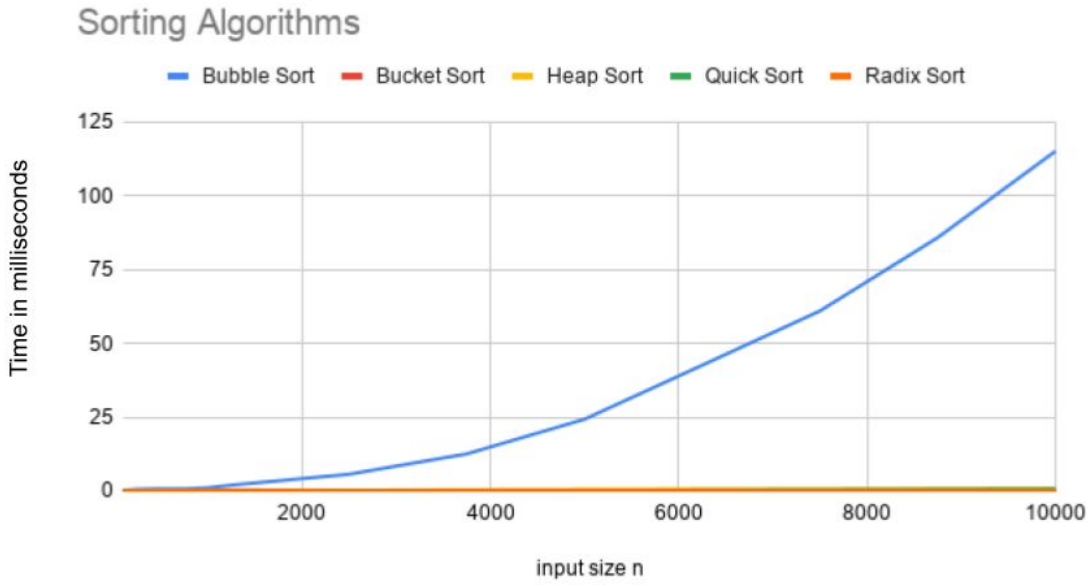


To measure this sorting algorithm, I used the input sizes :
100 250 500 750 1000 1250 2500 3750 5000 6250 7500 8750 10000.

The graph shows exponential growth from the beginning when working with the smaller input sizes. It then behaves in a logarithmic fashion when using the larger input sizes.

Implementation and Benchmarking

Throughout the benchmarking process, I used the same input sizes for each algorithm in order to effectively compare and contrast their performance. The algorithms were run ten times and the average of those ten runs was taken.



size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	0.239	0.601	0.770	0.709	1.078	2.003	5.683	12.585	24.324	42.621	61.009	85.837	115.100
Bucket Sort	0.017	0.036	0.052	0.070	0.063	0.118	0.087	0.090	0.114	0.048	0.014	0.017	0.016
Heap Sort	0.071	0.049	0.121	0.109	0.122	0.142	0.277	0.455	0.573	0.671	0.680	0.825	0.908
Quick Sort	0.051	0.042	0.038	0.050	0.040	0.041	0.105	0.174	0.252	0.349	0.464	0.571	0.723
Radix Sort	0.029	0.086	0.125	0.262	0.158	0.393	0.172	0.203	0.203	0.223	0.130	0.133	0.159

The above graph displays the average running time in milliseconds of each of each sorting algorithm.

Bubble Sort

Bubble sort has a worst and average complexity of n^2 and a best complexity of n. It is a comparison sort that works by comparing and swapping elements. The average run time of bubble sort is the slowest of all the algorithms that were tested. From the offset, it takes 0.239 milliseconds for a very small input size of 100. This can be accredited to how the algorithm functions. Two passes of the array are needed to sort the elements however one final pass is needed in order to know if the array has been sorted.

Bucket Sort

Bucket sort uses a scatter and gather approach, with each of the elements of an array being sorted into empty buckets and then gathered by taking the element in each sorted bucket and put back into the original array. Again it has a worst case of O(n^2). It begins with the fastest time with the smallest input size of 100 and a speed in milis 0.017. However, It remains as the fastest algorithm with the largest input size that was tested i.e. 10000. Bucket sort uses less comparisons than quicksort so it’s speed is expected. It is worth mentioning that the sample data has maximum three digits per input (random input ranges from 0 to 100).

Heap Sort

Heapsort's best, worst and average complexities are n*log(n). On average it performed as the second slowest sorting algorithm when it was tested. On average it took 0.071 milliseconds for the smallest input size of 100. It also performs as the second slowest algorithm tested when looking at the largest input size of 10000, where it needed 0.908 milliseconds. Compared to quicksort which is another comparison based sort, heap sort is only marginally slower.

Quick Sort

Quick sort has a best and average case of n*log(n) and a worst case of (n^2). In this benchmarking exercise it is the third quickest algorithm. It is the quickest comparison based algorithm that was tested. With the lowest input size of 100 it took 0.051 milliseconds. It remains in third place with the largest input size of 10000 using 0.723 milliseconds. It comes in third position with the two algorithms before it being non comparison sorts i.e. radix and bucket sort.

Radix Sort

Radix sort is the second fastest sort that was tested. It took on average 0.029 milliseconds to run an input of 100 and 0.159 to run an input size of 10000. It is a non comparative stable sort that distributes elements into buckets according to their radix. The worst case for radix sort is n*k/d. It comes second only to bucket sort which is also a non comparison stable sort. From the offset, there is only a marginal difference between the average running times when handling an input size of 100. However, radix sort pulls away with an average running time of 0.159 when the input size is 10000, and bucket sort achieves an average time of 0.016 with the same input.