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

REVIEW ON SORTING ALGORITHMS

Computational Thinking with Algorithms

Lecturer: Dominic Carr

MAY 3, 2021

HIGHER DIPLOMA IN SCIENCE – COMPUTING (DATA ANALYTICS)
Olga Rozhdestvina
G00387844

Table of Contents

I. Introduction

Sorting is a process of organising a list of elements in a particular order. The most used orders are lexicographical (alphabetic) and numeric (ascending or descending). Sorting increases efficiency of subsequent operations performed on the elements since it is easier to handle sorted elements than randomize data (S. Paira et al., 2014). While sorting is a simple concept, it is frequently used as an intermediate step by complex computer programs such as data compression, path finding, data search, media recovery etc. This makes sorting a fundamental operation in computer science.

Sorting algorithm is an algorithm that takes an array as input and outputs a permutation of that array that is sorted (Sorting Algorithms, n.d.). Enhancing the existing sorting algorithms or producing new algorithms can greatly optimize other algorithms. Thus, a large number of algorithms has been developed to improve sorting, each approaching the reordering of elements differently in order to increase the performance and efficiency of the practical applications (K. S. Al-Kharabsheh et al., 2013).

There are two categories of sorting algorithms: *comparison* and *non-comparison based* sorts. Comparison based sorting algorithms determine the sorted order by comparing input elements, whereas non-comparison based - by performing operations other than comparisons. In other words, comparison sorts require a *comparator function* to define ordering, and thus can be used for sorting any object. On the other hand, non-comparison based sorting algorithms do not rely on a *comparator function* so they can only be used to sort integers (T. Cormen et al., 2003). Examples of comparison based sorts are simple comparison-based sorts (Bubble sort, Selection sort and Insertion sort) and efficient comparison-based sort (Merge sort, Quicksort and Heap sort). Counting sort, Bucket sort or Radix sort are examples non-comparison sorts.

When comparing various sorting algorithms, there are several factors that must be taken in consideration.

1. Time complexity.

Time complexity of an algorithm signifies the amount of time that required by an algorithm to run till its completion. The time complexity of an algorithm is generally written in form *big O(n) notation*, where the *O* represents the complexity of the algorithm and a value *n* represent the number of elementary operations performed by the algorithm (C. Scheideler, 2005). For example, a sorting algorithm has *O(1)*, or constant time complexity if it needs to operate on one element of input list (regardless of the size). Time complexity of *O*(*n*) means an algorithm operates on each of the *n* elements of input list only once, etc. There are three types of time complexity: *best, average*, and *worst case* complexity (K. S. Al-Kharabsheh et al., 2013).

2. Space complexity

Space complexity describes the amount of memory (space) necessary to perform a task that an algorithm is expected to solve. For example, space complexity of *O(1)* means that the algorithm doesn't require extra memory allocation to sort the input list. The sorting in this case is done *in-place*. A sorting algorithm has a *O(n)* space complexity when it needs the allocation of new space in memory like creating a new. This is an *outplace* sorting technique (Space Complexity, n.d.). Sorting algorithms that use recursive techniques require more copies of sorting data which increases their space complexity. It is debated whether to call them in-place or outplace. In this report, I will support Cormen's view that a sorting algorithm sorts in-place if in addition to the original input array it does not require any more than a constant amount of storage (T. Cormen et al., 2003).

3. Stability

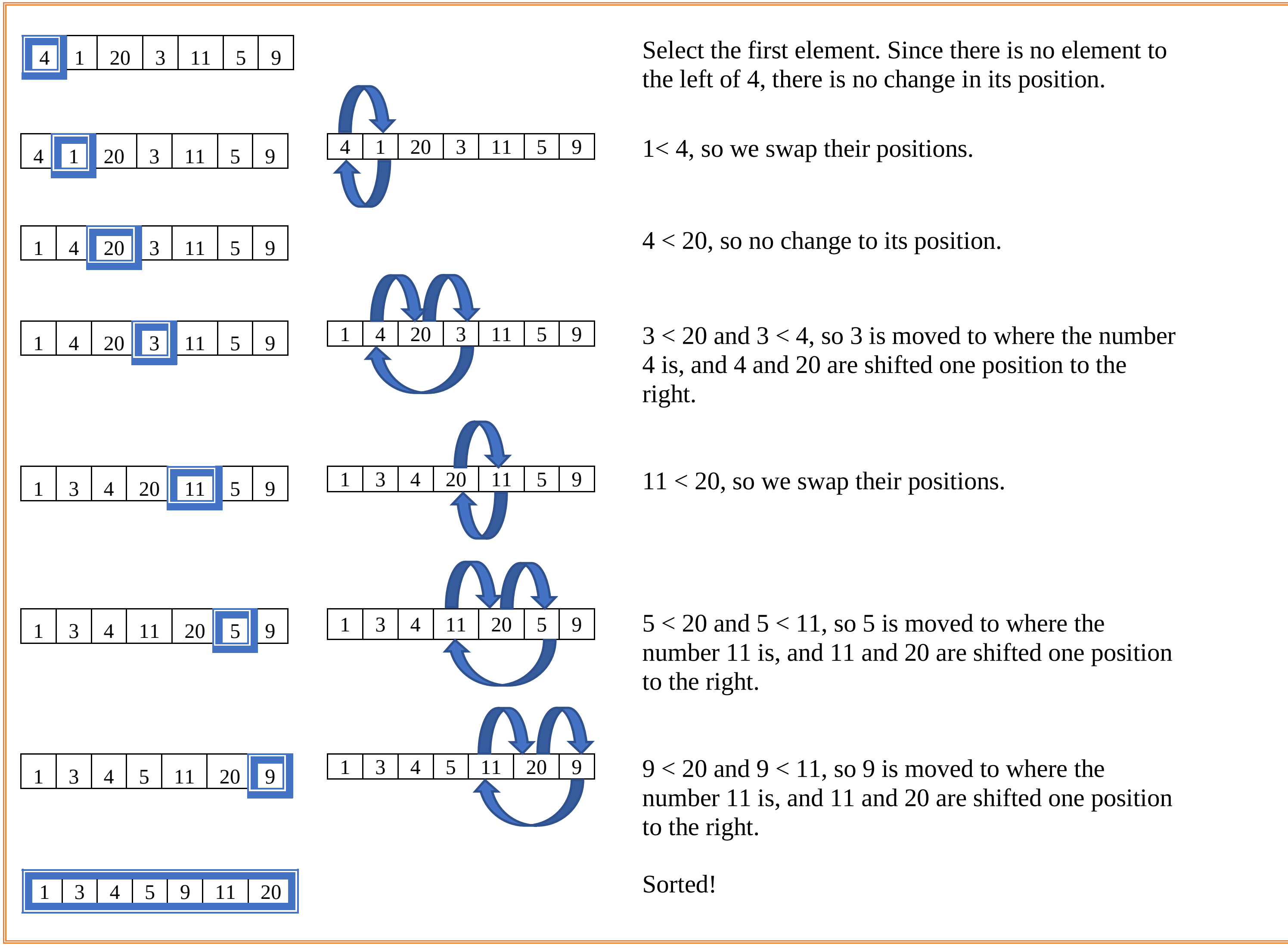
The stability of a sorting algorithm concerns preserving the same relative order of elements with equal values in the output as they were in the input. Some sorting algorithms are stable by its nature (Bubble sort, Insertion sort, Merge sort) while some sorting algorithms are not (Selection sort, Heap sort, Quicksort). However, any given unstable sort can be modified to be stable. (K. S. Al-Kharabsheh et al., 2013)

II. Five Sorting Algorithms

In this report I examined five sorting algorithms: Insertion sort, Quicksort, Heap sort, Bucket sort and Introsort.

1. Insertion Sort

Insertion sort is a simple comparison-based sort that builds a final list one element at a time. Consider the example below (Figure 1) where an array [4, 1, 20, 3, 11, 5, 9] needs to be sorted.



Select the first element. Since there is no element to the left of 4, there is no change in its position.

1< 4, so we swap their positions.

4 < 20, so no change to its position.

3 < 20 and 3 < 4, so 3 is moved to where the number 4 is, and 4 and 20 are shifted one position to the right.

11 < 20, so we swap their positions.

5 < 20 and 5 < 11, so 5 is moved to where the number 11 is, and 11 and 20 are shifted one position to the right.

9 < 20 and 9 < 11, so 9 is moved to where the number 11 is, and 11 and 20 are shifted one position to the right.

Sorted!

Figure 1 Insertion Sort Diagram

Insertion sort is a stable in-place sort with $O(1)$ space complexity. In the best case its time complexity is $O(n)$, in the worst and average cases - $O(n^2)$. The best case occurs when the data is already sorted, so Insertion sort only compares $O(n)$ elements without performing any swaps. It runs worst when the elements in the list are sorted in decreasing order when for the last element insertion at most $n-1$ comparisons and $n-1$ swaps are needed, for the second to last element insertion – at most $n-2$ comparisons and $n-2$ swaps, etc. Hence the number of steps required come to $2 \times (1+2 + \cdots + n-2 + n-1)$ (Insertion Sort, n.d.).

Among the advantages of Insertion sort are a tight code and efficiency when sorting small and nearly sorted data. However, it is much less efficient for sorting large and more disordered lists.

2. Quicksort

Quicksort is an efficient comparison-based sort that utilizes a divide-and-conquer approach to sorting lists. A divide-and-conquer technique recursively breaks an algorithm into two or more sub-problems until they are simple enough to be solved, and then combines the results back together to solve the original problem (T. Cormen et al., 2003). In the case of Quicksort, a pivot is selected in order to divide the algorithm into subarrays. The example of quicksort is in the Figure 2.

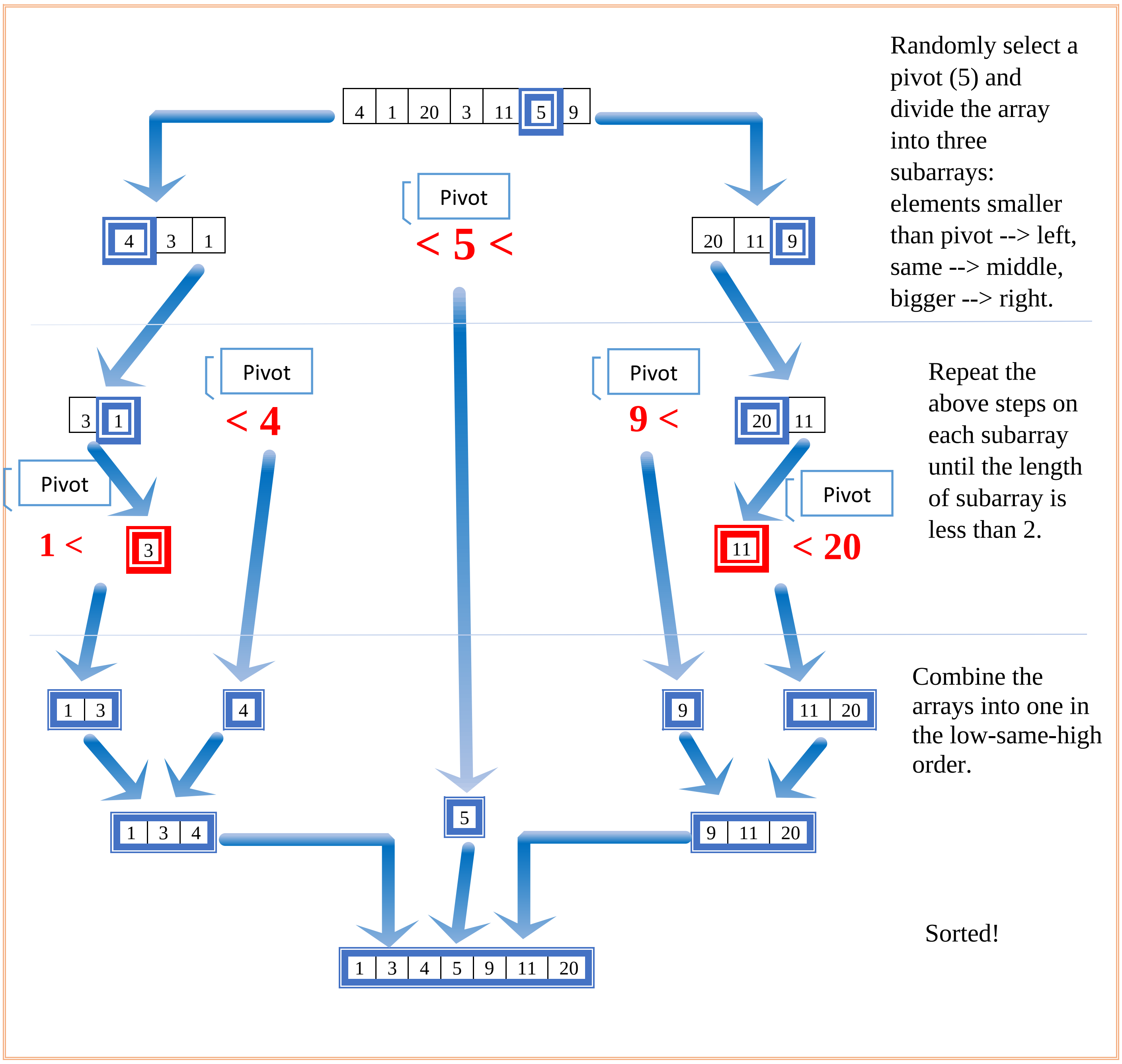

4 1 20 3 11 5 9
Randomly select a pivot (5) and divide the array into three subarrays: elements smaller than pivot --> left, same --> middle, bigger --> right.
Pivot
< 5 <
4 3 1
20 11 9
Pivot
< 4
Pivot
9 <
3 1
20 11
Repeat the above steps on each subarray until the length of subarray is less than 2.
Pivot
1 <
3
Pivot
11
< 20
1 3
4
9
11 20
Combine the arrays into one in the low-same-high order.
1 3 4
5
9 11 20
1 3 4 5 9 11 20
Sorted!

Figure 2 Quicksort Diagram

Choosing a good pivot is a crucial step in assuring fast implementation of Quicksort. The best pivot would split the input array in two even subarrays, which would halve the problem size (the best case time complexity). The example above uses a randomly selected pivot. However, there are other ways of picking it:

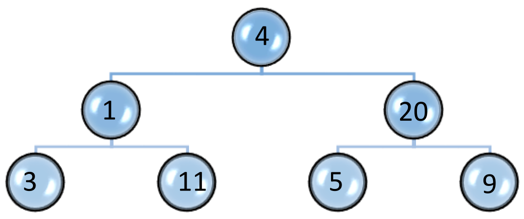
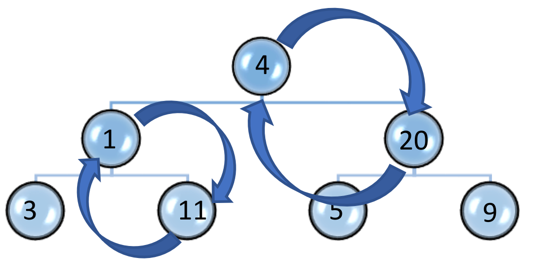
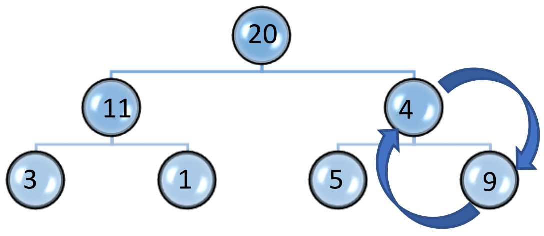
1. Select the leftmost or rightmost element as the pivot.
2. Median-of-three method: take the first, middle, and last value of the array, and choose the median of those three numbers as the pivot.
3. Use a median-finding algorithm such as the median-of-medians algorithm.

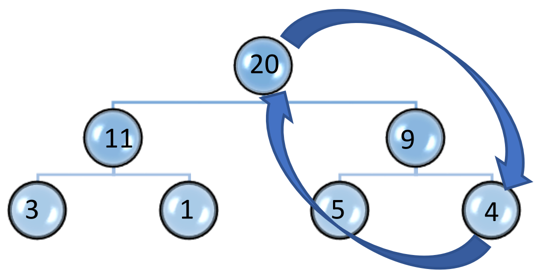
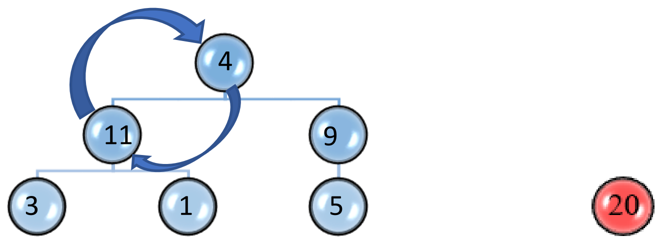
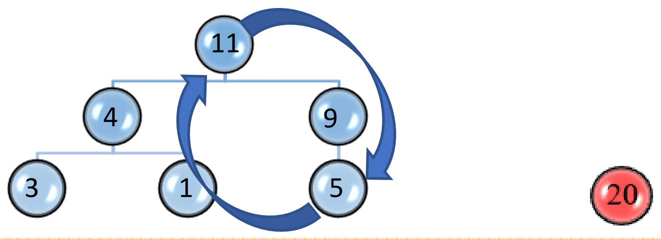
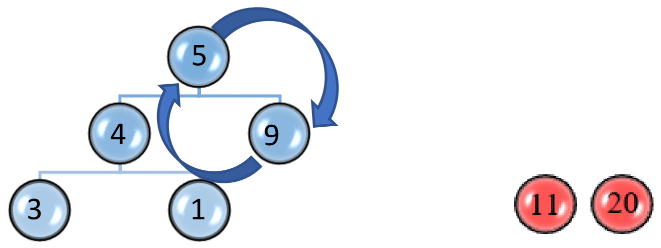
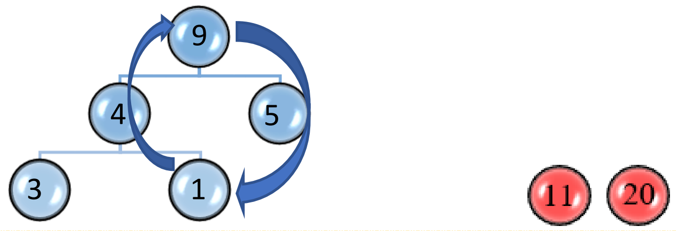
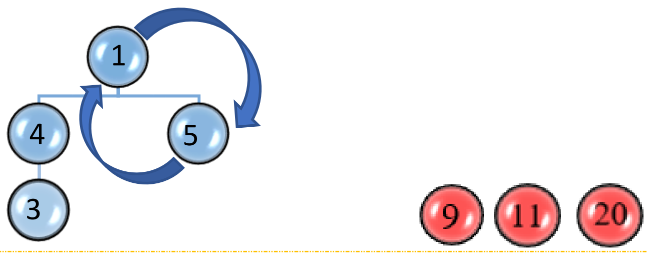
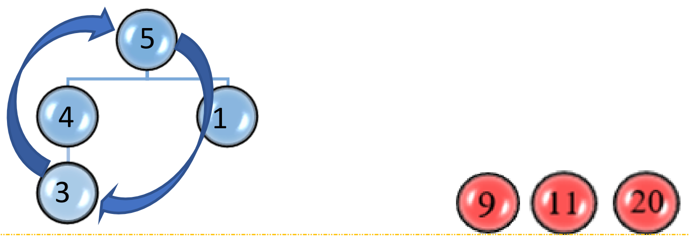
Quicksort is usually implemented as an unstable sort with a best case space complexity of $O(log\ n)$ and an average case space complexity of $O(n)$. In the best and average cases, the algorithm runs $O(n\ log\ n)$ time, in the worst case – $O(n^2)$. The worst case would be when the pivot is either the minimum or maximum element in the array (Quick Sort, n.d.).

Quicksort usually outperforms other comparison based sorts since it has better constant factors, thus considered to be the best practical choice for sorting large data. Like Insertion sort, it has simple implementation, and it sorts in-place, but it also can run as slow as Insertion sort in the worst case (T. Cormen et al., 2003).

3. Heap Sort

Heap sort is another efficient comparison-based sorting algorithm that uses a binary heap data structure. The algorithm has two parts: building a max-heap and then sorting it. The max-heap suggests that a node cannot be of a greater value than its parent. When building a max-heap, the largest element is moved to the root, and the minimum elements - to the leaves. The sorting is done by repeated removal of the largest element from the heap and its insertion into the array (Heap Sort, n.d.). Consider the Figure 3 below.

	Initial array [4, 1, 20, 3, 11, 5, 9] as a binary heap.
	Initially build a max-heap of elements in the array. Array now [20, 11, 9, 3, 1, 5, 4]
	

	20 is swapped with 4. Array now [4, 11, 9, 3, 1, 5, 20]	
	20 is removed from heap as it is in correct position now. Max-heap is created. Array now [11, 4, 9, 3, 1, 5, 20]	
	11 is swapped with 5. Array now [5, 4, 9, 3, 1, 5, 11, 20]	
	11 is removed from heap as it is in its correct position now. Max-heap is created. Array now [9, 4, 5, 3, 1, 11, 20]	
	9 is swapped with 1. Array now [1, 4, 5, 3, 9, 11, 20]	
	9 is removed from heap as it is in its correct position now. Max-heap is created. Array now [5, 4, 1, 3, 9, 11, 20]	
	5 is swapped with 3. Array now [3, 4, 1, 5, 9, 11, 20]	

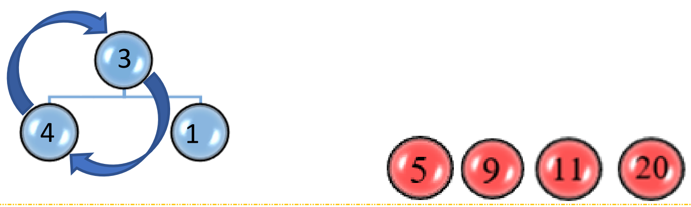
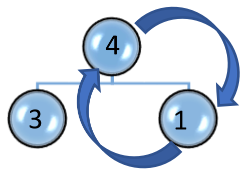
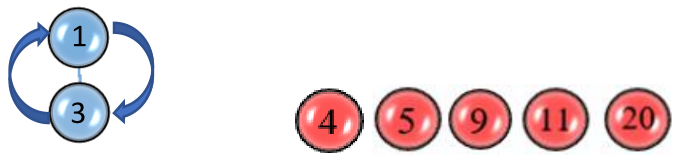
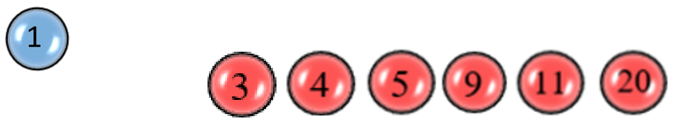
	5 is removed from heap as it is in its correct position now. Max-heap is created. Array now [4, 3, 1, 5, 9, 11, 20]
	4 is swapped with 1. Array now [1, 3, 4, 5, 9, 11, 20]
	4 is removed from heap as it is in its correct position now. Max-heap is created and 3 is swapped with 1. Array now [1, 3, 4, 5, 9, 11, 20]
	3 is removed from heap as it is in its correct position now. 1 is removed from heap as in its own position now.
	Sorted!

Figure 3 Heap Sort Diagram

Heap sort is an unstable sorting algorithm that has a space complexity of *O(1)* and performs in-place sorting. It has a worst, average and best case running time of *O(n log n)* which means that it is consistent in its performance. As seen in Figure 3, the algorithm uses the heap properties of a binary heap data structure, such as the efficient removal from and inserting to the heap, to its advantage. However, it is still slower than Quicksort in its best and average cases (T. Cormen et al., 2003).

Heap sort is the best choice when it is very important to have a consistently fast running time (especially in the worst case) and efficient memory usage.

4. Bucket Sort

Bucket sort is a non-comparison sort that uses a method of assigning each element of the input array in its corresponding *bucket*, and then recombines all buckets to yield an ordered array. The size and content of the input elements define the number and range of the buckets.

There are several special conditions that are required for bucket sort:

1. The input array to be sorted consists of evenly distributed elements in a range [0, max].
2. It is possible to tell which part of the range each element is in.
3. The range can be divided into equal parts of same size (J. Paton, 2012; T. Cormen et al., 2003).

In case that input elements do not follow these conditions, Bucket sort require an assistance of any comparison based sorting algorithm (usually Insertion sort). For this first the size of each bucket is found by dividing maximum input element by the length of the array. Then each element is divided by the size to find the bucket index it needs to be assigned to. Next, each bucket is sorted with a comparison based sort. Finally, the buckets are recombined.

In Figure 4 to demonstrate the bucket sort working without assistance of any comparison sort, I chose an equally distributed array in a range [0, 6] including.

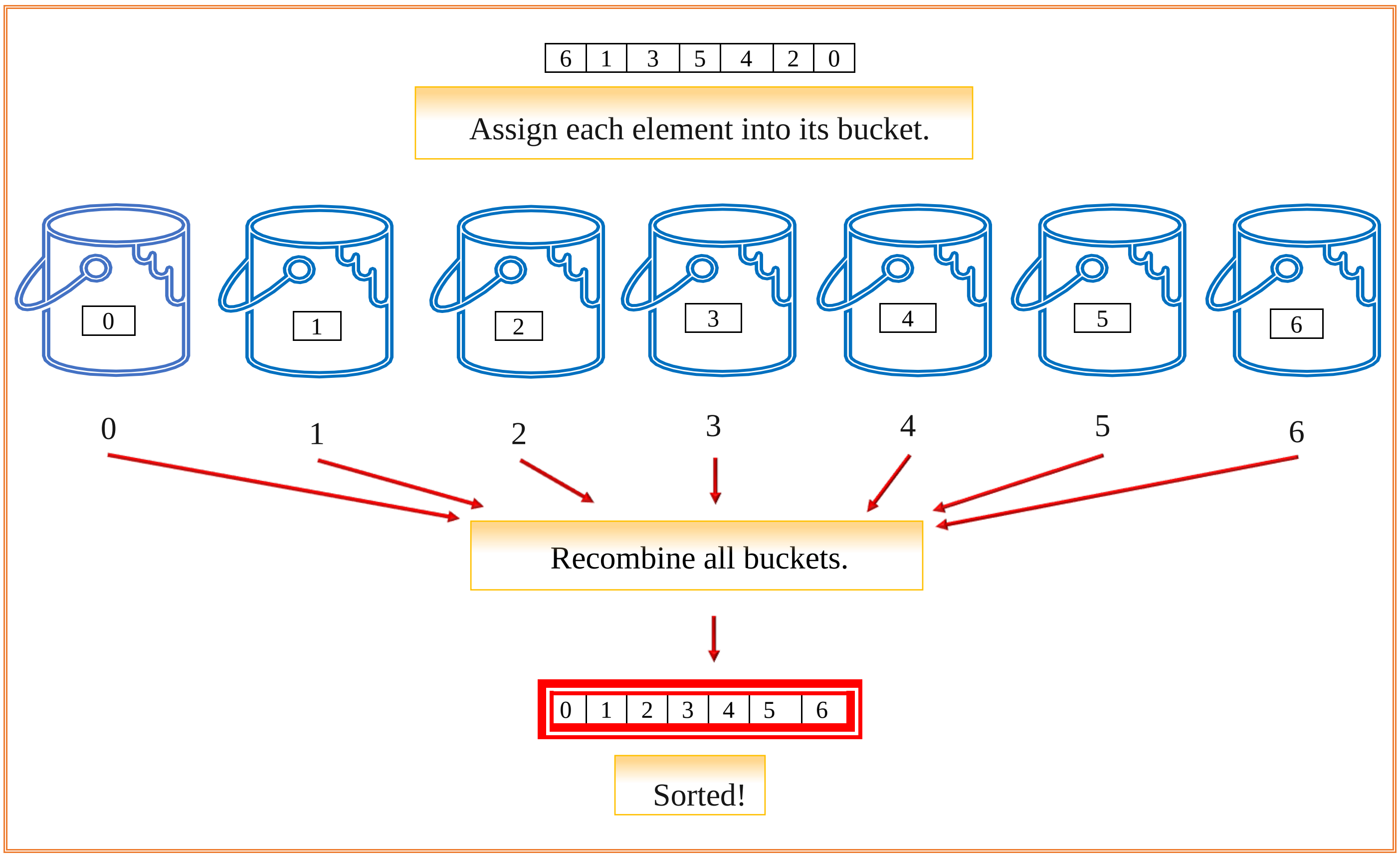


Figure 4 Bucket Sort Diagram

Bucket sort is a stable sorting algorithm since we add items to the buckets in order, and concatenating them preserves that order (D. Eppstein, 1996). However, if Bucket sort requires assistance of another sorting algorithm, then its stability depends on the stability of the underlying algorithm. It has a space complexity of *O(nk)* and it performs the sorting outplace.

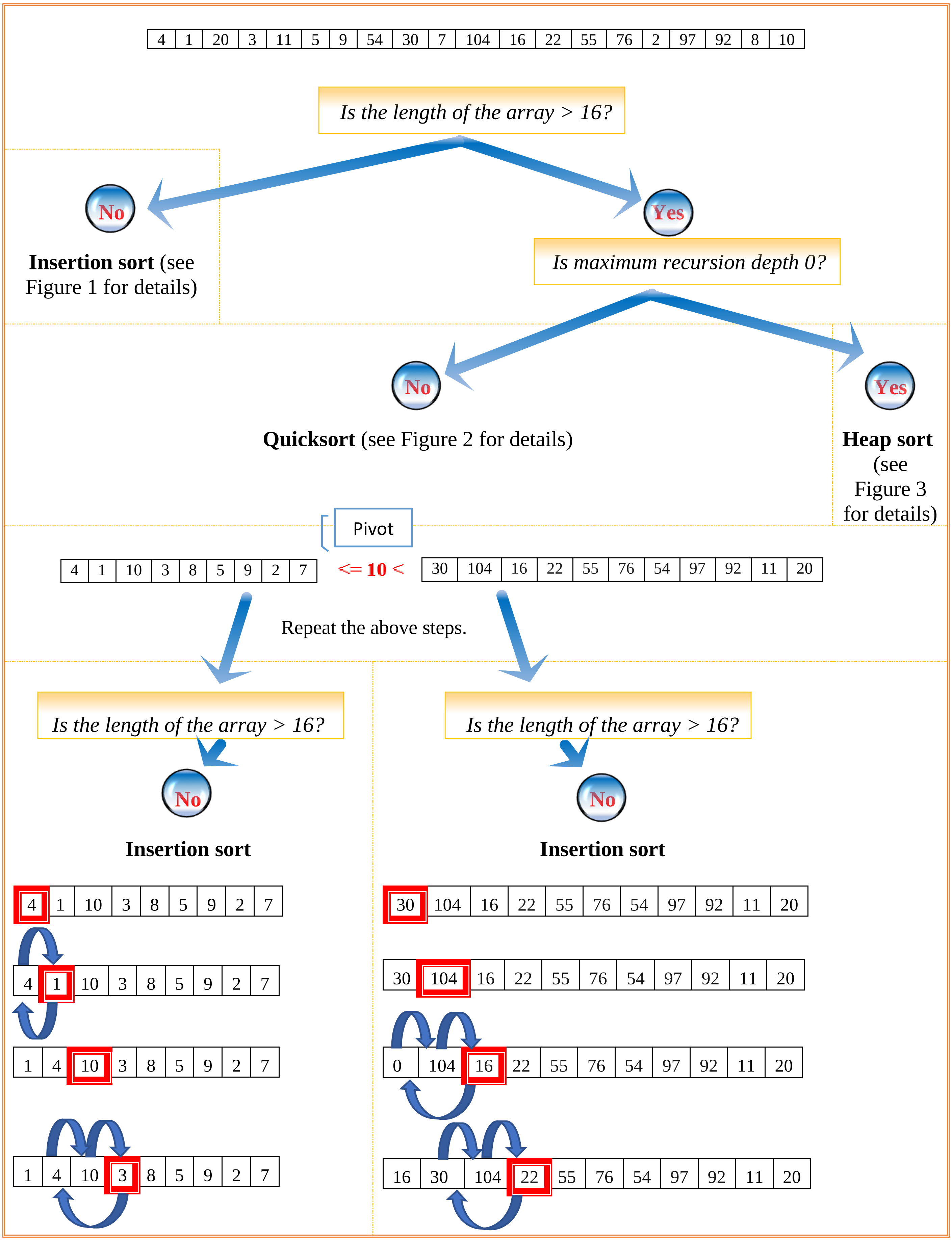
The best and average case time complexity of Bucket sort is $O(n+k)$ where $O(n)$ time is for making a bucket list and $O(k)$ for sorting the elements in each bucket. The best case occurs when each bucket in the bucket list has only one element. Bucket sort runs $O(n^2)$ in the worst case and happens when the input elements are in close range which increases their likelihood to be placed in the same bucket. This makes the time complexity depend on the underlying comparative sorting algorithm. For example, if the input array is in reverse order and the underlying sort is Insertion sort, the time complexity will become $O(n^2)$ (J. Paton, 2012).

Bucket sort is the best option when the data is uniformly distributed. It also works well if there is a large degree of parallelism available. Another advantage of Bucket sort is that it can be used as an external sorting algorithm (in case of a large amount of data that cannot fit into memory) (Aho et al., 2003).

5. Introsort

Introsort (Introspective sort) is a hybrid sorting algorithm that consists of three sorting comparison based sorts: Insertion sort, Quicksort, and Heap sort. It behaves almost exactly like Quicksort with a Median-of-three chosen pivot for most input data with ability to detect partitioning tendency toward quadratic behaviour.

Introsort begins by checking the length of the input array. If it is smaller than a certain threshold, it sorts the array using Insertion sort. If the length is bigger, then it starts with Quicksort and if the recursion depth goes above a particular limit it switches to Heap sort to avoid Quicksort's worse case $O(n^2)$ time complexity (Musser, 1997). The work of Introsort in shown in Figure 5.



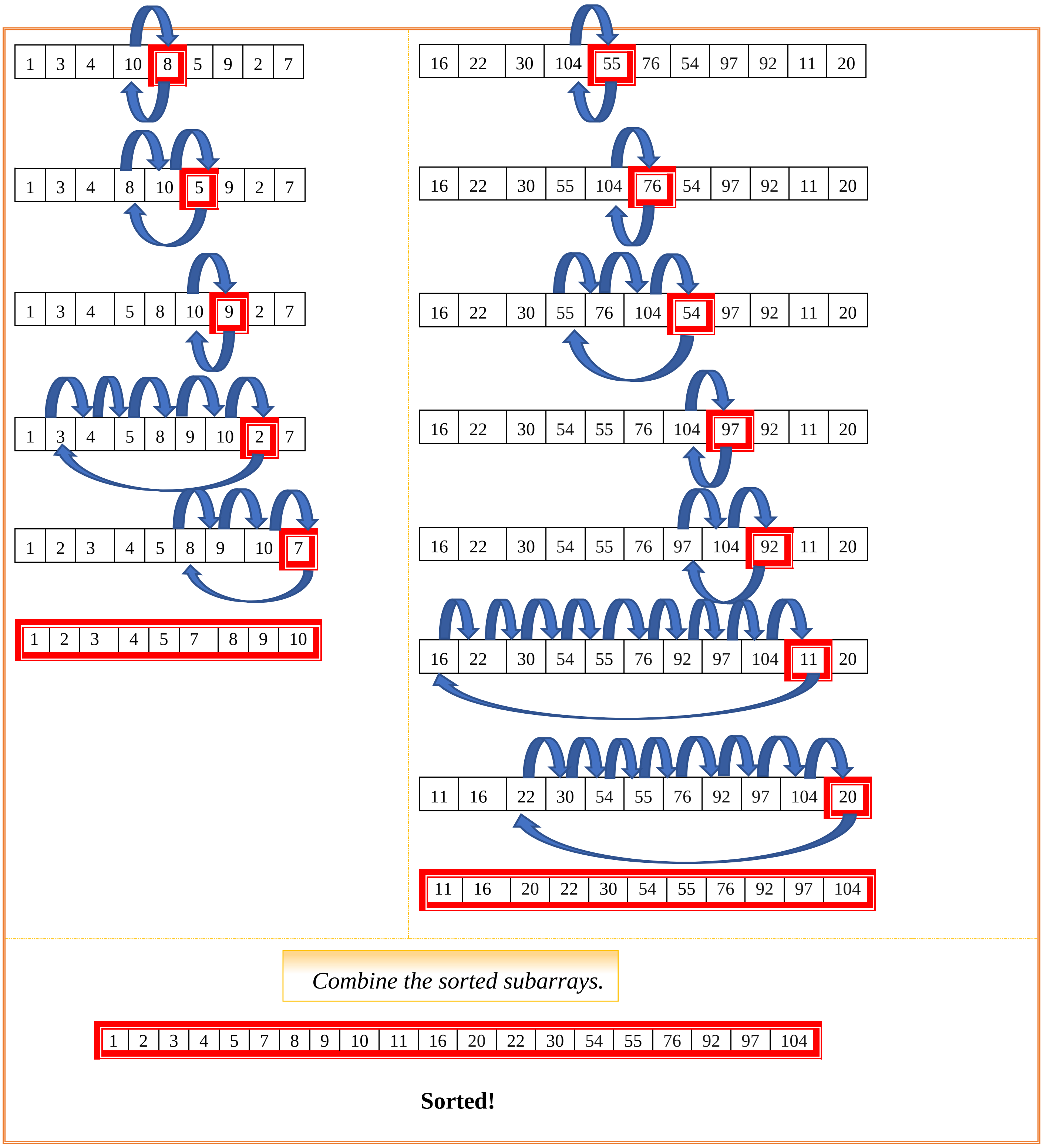


Figure 5 Introsort Diagram

Introsort is an unstable sorting algorithm (since Quicksort and Heap sort are unstable) with space complexity of Quicksort (the best case *O*(*log n*) and average - *O*(*n*)). It runs in its worst, average and best cases at *O(n log n)* time, but is almost always faster than just using Heap sort. In other words, it has best and average running time of Quicksort and worst - of Heap sort.

III. Implementation & Benchmarking

All five sorting algorithms (Insertion sort, Quicksort, Heap sort, Bucket sort and Introsort) were implemented in Python and tested for the random sequence input of length from 100 to 15000. All five were executed on machine Operating System having Intel(R) Core (TM) i7-7700HQ CPU @ 2.80 GHz and installed memory (RAM) 8.00 GB. The built-in function *time* was used to get the execution time of the algorithms (in millisecond).

My original expectations of running sorting algorithms:

1. Bucket sort should outperform all other algorithms and show its best running time since all input arrays are uniformly distributed in range [0, max], and thus my implementation of Bucket sort does not require an underlying comparison based sort.
2. Insertion sort performance is likely to be the worst since the input arrays are large and randomised, therefore the sort should quickly escalate to its worst case $O(n^2)$.
3. Introsort is expected to slightly outperform Quicksort since it always chooses the best pivot (using Median-of-three method), whereas my Quicksort implementation uses a randomly selected pivot.
4. Heap sort should demonstrate its *O(n log n)* running time which is usually slower than Quicksort.

These assumptions are summed up in Table 1.

Performance rating	***Sorting Algorithm***	***Time Complexity***
1	Bucket sort	O(n+k)
2	Introsort	O(n log n)
3	Quicksort	O(n log n)
4	Heap sort	O(n log n)
5	Insertion sort	O(n) $\longrightarrow$ $O(n^2)$

Table 1 Expected performance of sorting algorithms

The result of running the benchmark application for each algorithm is given in Table 2 and the curves are shown in Figure 6 (all five algorithms) and Figure 7 (zoom-in on Quicksort, Heap sort, Bucker sort and Introsort).

Sizes	**100**	**250**	**500**	**750**	**1000**	**1250**	**2500**	**5000**	**6250**	**7500**	**8750**	**10000**	**15000**
Insertion Sort	3.041	21.59	90.587	241.919	459.373	804.925	2106.572	8517.051	12356.96	18223.15	28473.93	33703.88	73387.91
Quicksort	1.222	2.74	3.974	7.828	10.384	12.641	37.459	73.064	84.658	127.222	140.433	169.978	241.43
Heap Sort	1.543	4.479	10.483	16.908	23.068	31.06	69.398	206.932	202.412	290.046	316.534	389.566	584.6
Bucket Sort	0.209	0.412	0.518	0.539	0.873	6.104	3.231	8.733	12.706	14.062	19.19	21.531	27.872
Introsort	0.4	2.927	6.755	10.262	12.015	11.775	22.814	51.243	64.962	94.549	120.797	109.447	233.377

Table 2 Running time benchmark (the average of 10 repeated runs)

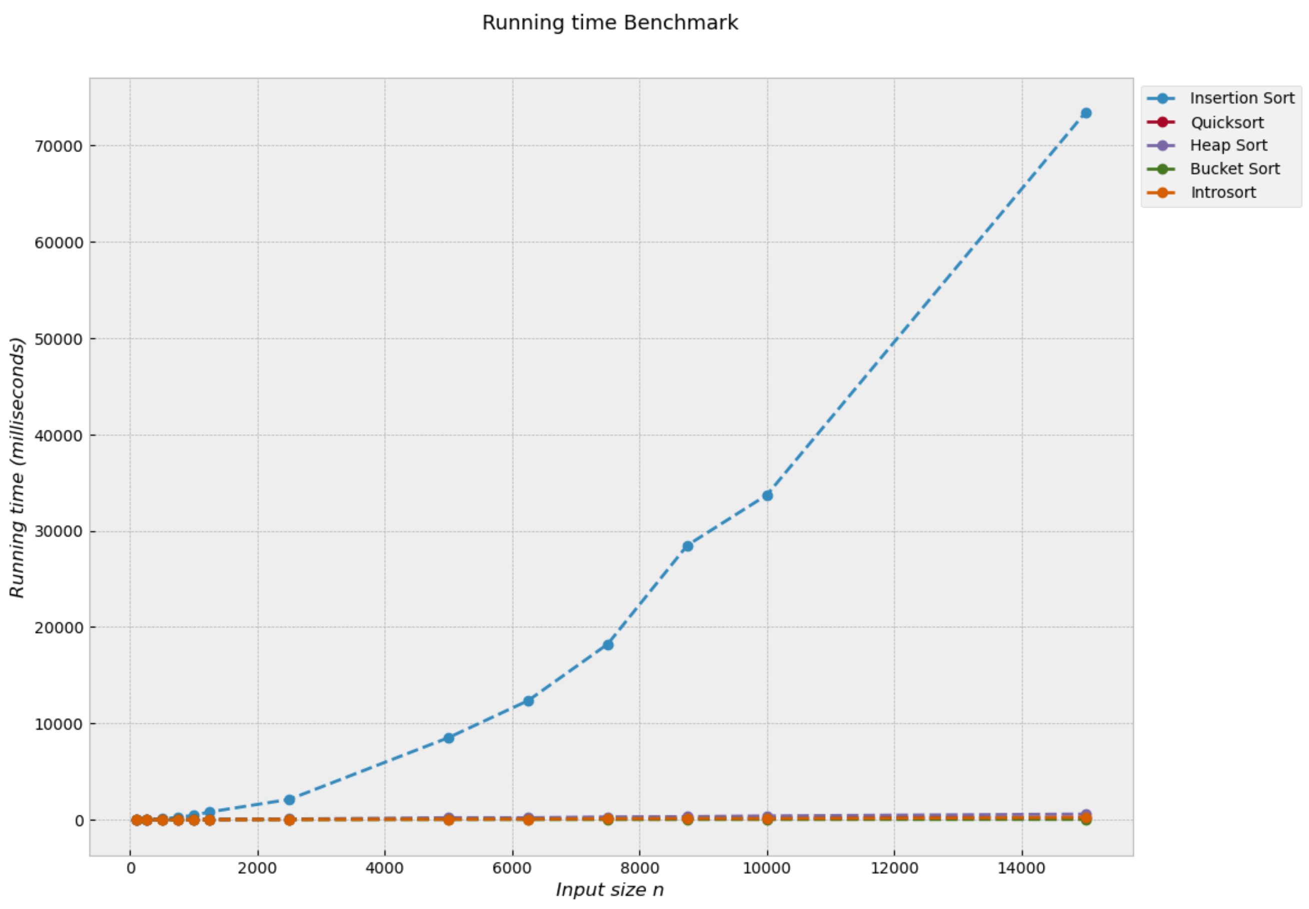


Figure 6 Benchmark plot for all five sorting algorithms

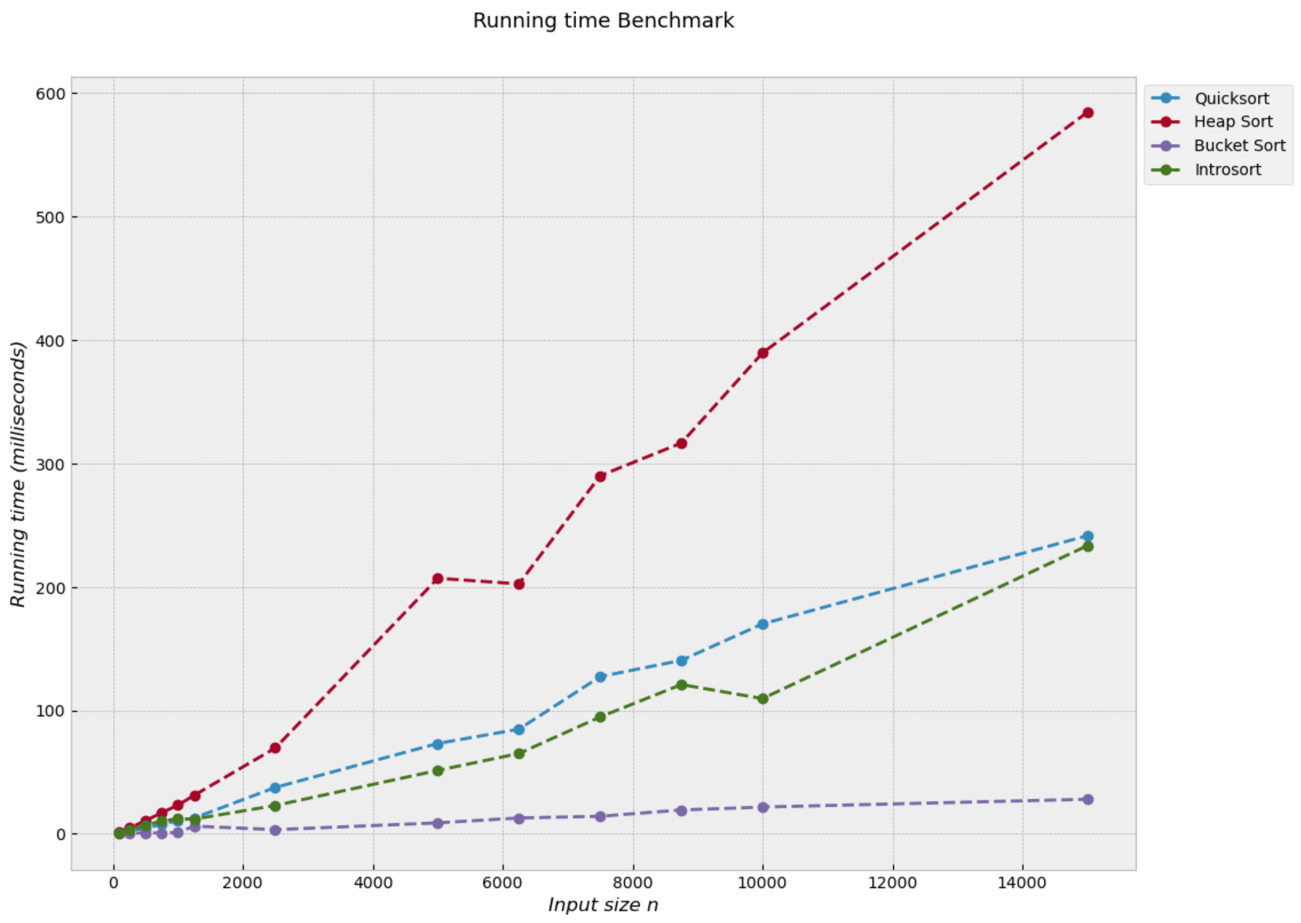


Figure 7 Benchmark plot (excluding Insertion sort)

For small input, the performance for all five algorithms is almost nearest, whereas for the large input, Insertion sort shows the slowest running time and Bucket sort - the fastest. The result confirms the original assumptions on the performance rating.

To confirm the time complexity of each sorting algorithm the above curves were plotted alongside their anticipated running time. The results are depicted in Figure 8 (note that Bucket sort is compared to Logarithmic time complexity).

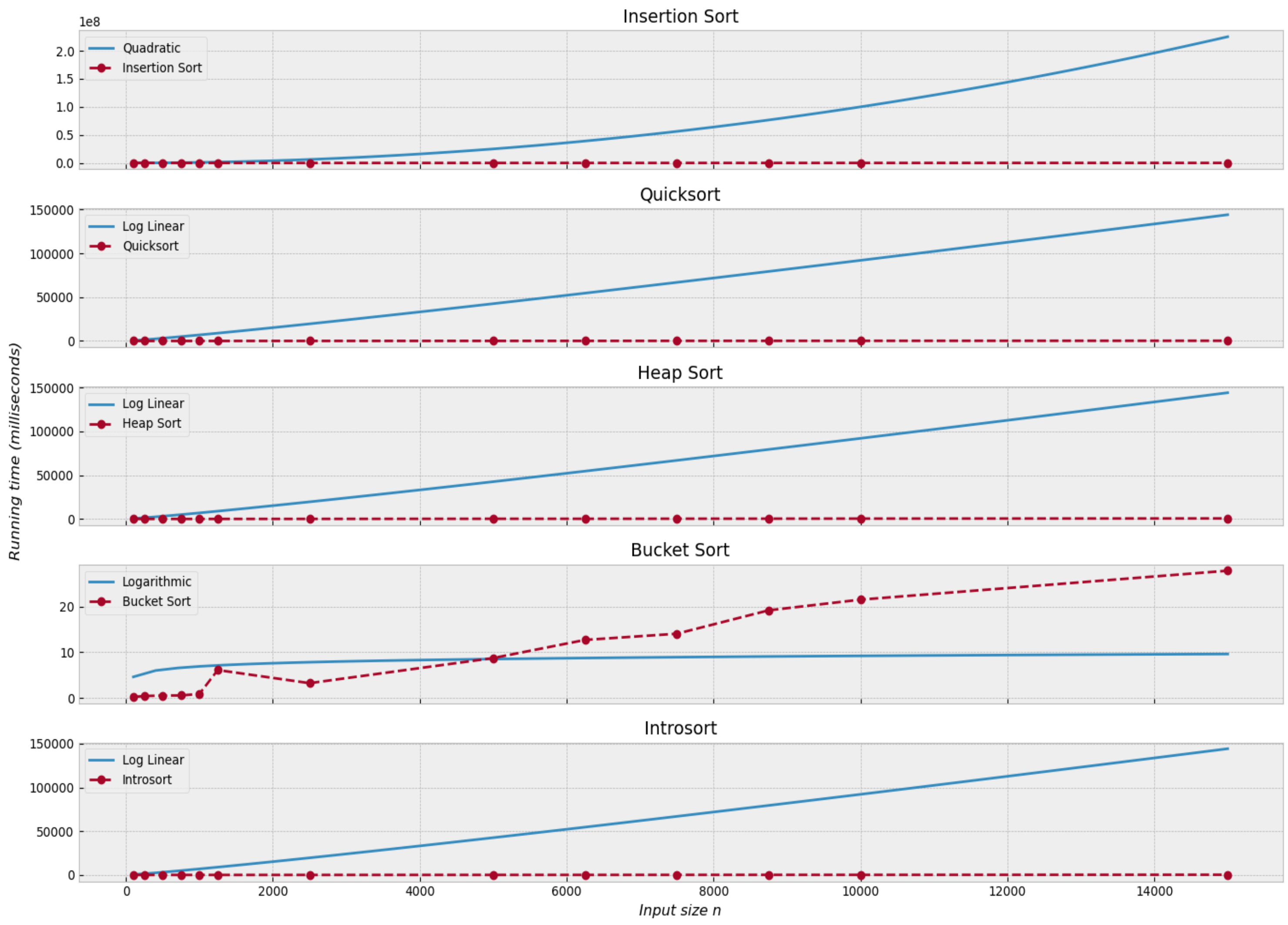


Figure 8 Expected Big O Time Complexity plot

It is clear that Quicksort, Introsort and Heap sort performed better than their best and average *O(n log n)* time complexity – their performance appears to be closer if not better than *O(n)*. Performance of Insertion sort seems to be closer or better than *O(n log n)*. Bucket sort performed near to what was originally expected (slightly worse than *O(log n)*. The adjusted result is shown in Figure 9.

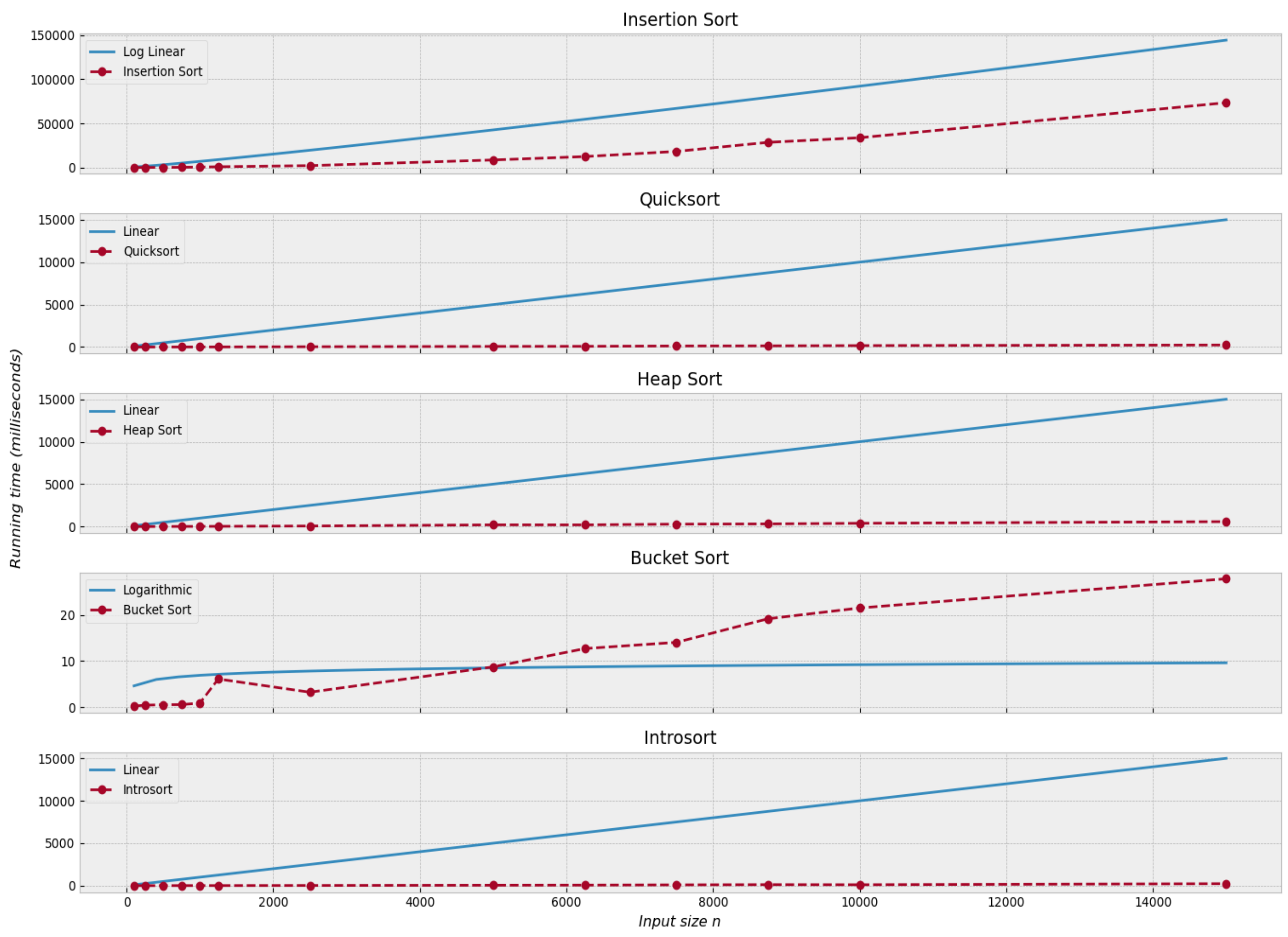


Figure 9 Adjusted Big O Time Complexity plot

IV. Conclusion

This report introduces and compares five sorting algorithms: Insertion sort, Quicksort, Heap sort, Bucket sort and Introsort. It also discusses the basic concepts of time and space complexity and the stability of these sorting techniques and their importance for choosing one algorithm over the other. This brief includes performance analysis of the algorithms for the same number of input elements in range from 100 to 15000. Table 1 shows that for small inputs the performance of all five algorithms is quite similar, but for the large input, Bucket sort is the most efficient and the Insertion sort the least. In Figure 8 we can see that Insertion sort, Quicksort, Heap sort and Introsort perform better than originally anticipated given the time complexity.

V. References

1. S. Paira, S. Chnadra, A. Sk Safikul, D.S. Partha (2014) 'A Review Report on Divide and Conquer Sorting Algorithm'. Available at: https://www.researchgate.net/publication/276847633_A_Review_Report_on_Divide_and_Conquer_Sorting_Algorithm (Accessed: May 3, 2021)

2. ‘Sorting Algorithms’. *Brilliant.org*. Available at: https://brilliant.org/wiki/sorting-algorithms/ (Accessed: May 3, 2021)
3. K. S. Al-Kharabsheh, I. M. AlTurani, A. M. I. AlTurani, N. I. Zanoon (2013) ‘Review on Sorting Algorithms A Comparative Study’. Available at: https://www.researchgate.net/publication/259911982_Review_on_Sorting_Algorithms_A_Comparative_Study (Accessed: May 3, 2021)
4. C. Scheideler (2005) ‘Introduction to Complexity Theory’. Available at: http://www.cs.jhu.edu/~scheideler/courses/600.471_S05/lecture_1.pdf (Accessed: May 4, 2021)
5. ‘Space Complexity’. *Brilliant.org*. Available at: https://brilliant.org/wiki/space-complexity/ (Accessed: May 4, 2021)
6. ‘Insertion Sort’. *Brilliant.org*. Available at: https://brilliant.org/wiki/insertion/ (Accessed: May 4, 2021)
7. T. Cormen, C. Leiserson, R. Rivest, C. Stein (2003) ‘Introduction to Algorithms (Second Edition)’. Available at: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf (Accessed: May 4, 2021)
8. ‘Quick Sort’. *Brilliant.org*. Available at: https://brilliant.org/wiki/quick-sort/ (Accessed: May 4, 2021)
9. ‘Heap Sort’. *Brilliant.org*. Available at: https://brilliant.org/wiki/heap-sort/ (Accessed: May 5, 2021)
10. M. J. Khalid. ‘Bucket Sort in Python’. Available at: https://stackabuse.com/bucket-sort-in-python/ (Accessed: May 5, 2021)
11. J. Paton (2012) ‘Introduction to Data Structures’. Available at: http://pages.cs.wisc.edu/~paton/readings/Old/fall08/LINEAR-SORTS.html (Accessed: May 7, 2021)
12. D. Eppstein (1996) ‘Design and Analysis of Algorithms’. Available at: https://www.ics.uci.edu/~eppstein/161/960123.html (Accessed: May 7, 2021)
13. A. V. Aho, M. Hill, J. E. Hopcroft, J. D. Ullman (2001) ‘Data Structures and Algorithms’. Available at: https://doc.lagout.org/Alfred%20V.%20Aho%20-%20Data%20Structures%20and%20Algorithms.pdf (Accessed: May 7, 2021)
14. D. R. Musser (1997). ‘Introspective Sorting and Selection Algorithms’ Available at: http://oucsace.cs.ohio.edu/~razvan/courses/cs4040/introsort.pdf (Accessed: May 8, 2021)