REVIEW ON SORTING ALGORITHMS

Computational Thinking with Algorithms

Lecturer: Dominic Carr

MAY 3, 2021

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

I. Introduction

Sorting is a process of organising a list of elements in a particular order. The most used orders are lexicographical (alphabetic) or 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 having 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 on-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 ().

2. Space complexity

Space complexity describes the amount of memory (or space) necessary to perform the task that the 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 array (or list). This is an *outplace* sorting technique. Sorting algorithms that use recursive techniques require more copies of sorting data which increases their space complexity (Space Complexity, n.d.).

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.

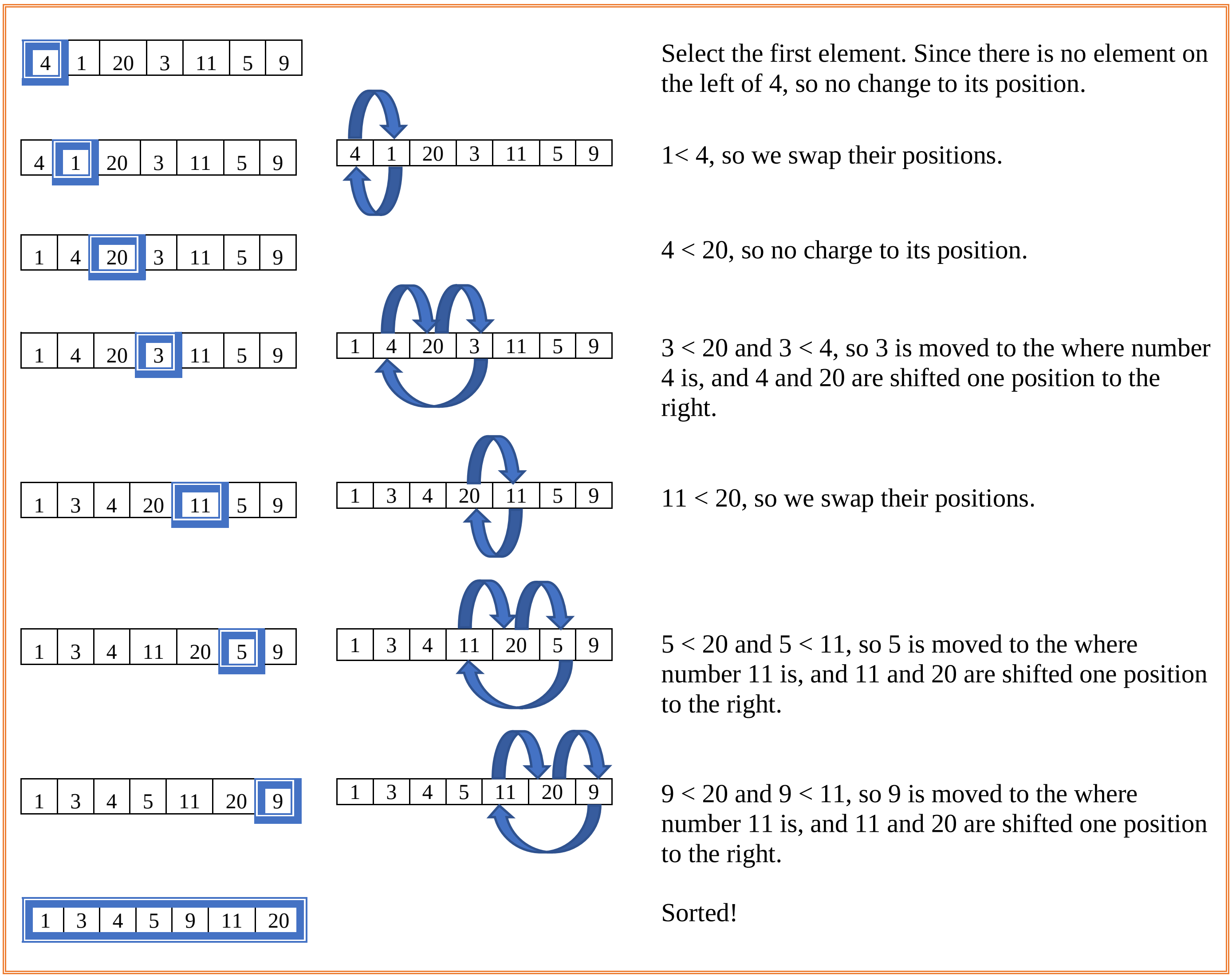


Figure 1 Insertion Sort Diagram

Insertion sort is a stable in-place sort with *O(1)* space complexity. Regarding the time complexity, in the best case it 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 in its 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 unordered lists.

2. Quicksort

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

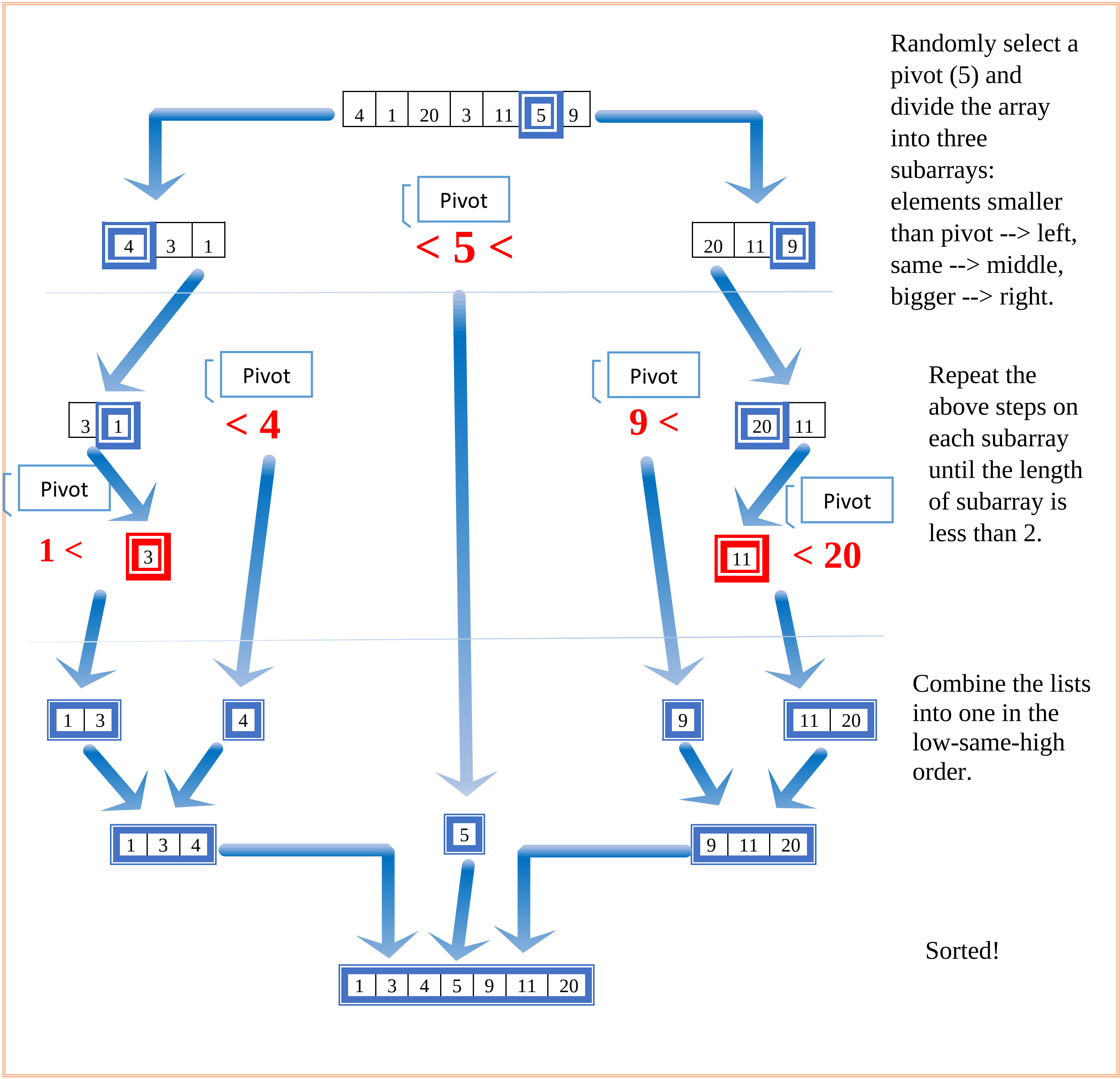


Figure 2 Quicksort Diagram

Choosing a good pivot is the crucial part to a 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.

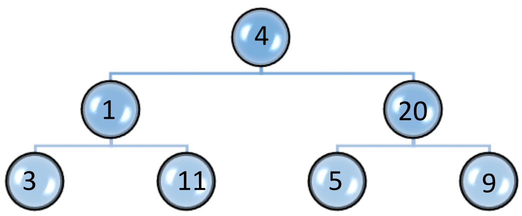
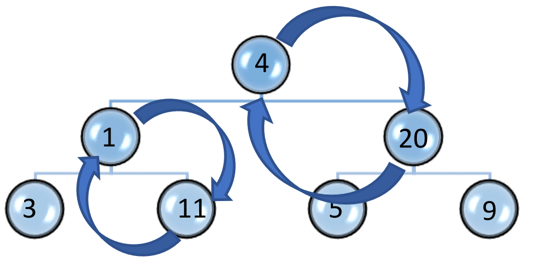
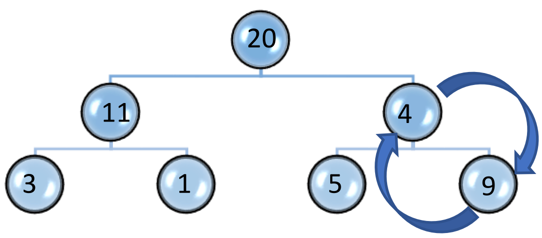
Quicksort is usually implemented as an unstable sort with a best-case space complexity of *O*(*log n*) and an average-case space complexity of *O*(*n*). In the best and average cases, the algorithm runs *O(n log n)* time, in the worst case – $O(n^2)$. The worst case would be when the pivot is either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

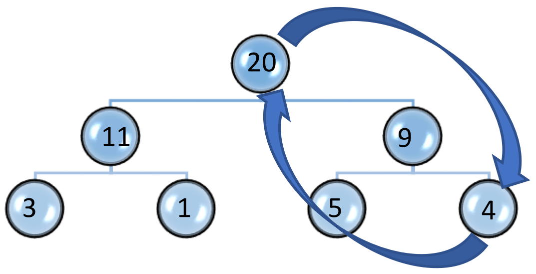
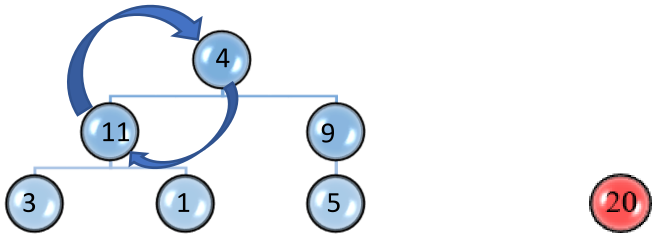
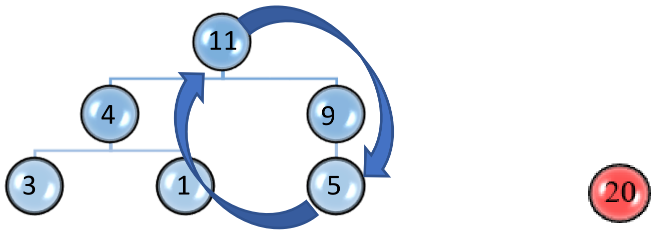
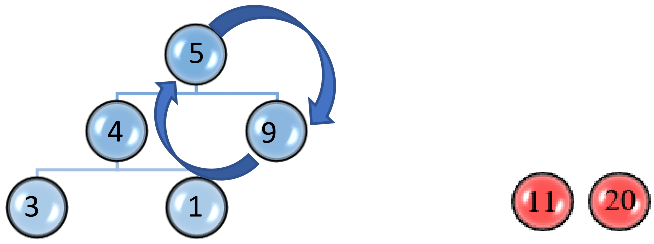
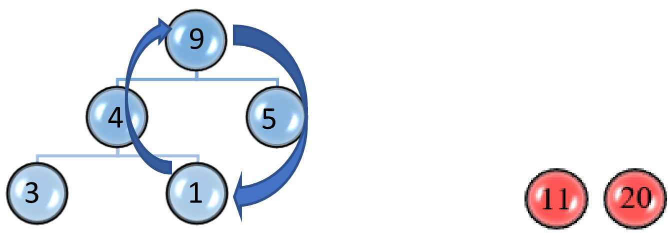
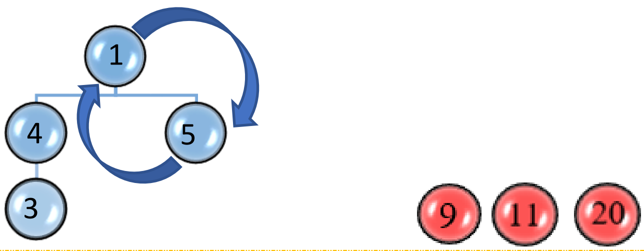
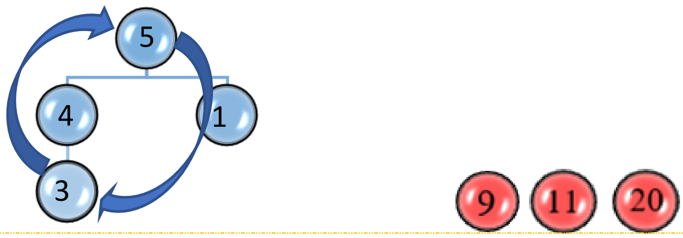
Quicksort usually outperforms other comparison based sorts since it has better constant factors, thus considered to be the best practical choice for sorting large data. Like Insertion sort, it has simple implementation, and it sorts in-place, but it also can run as slow as Insertion sort in the worst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

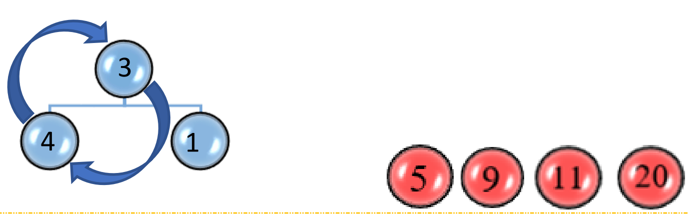
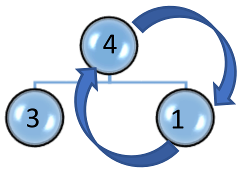
	5 is removed from heap as it is in its correct position now. Max-heap is created. Array now [4, 3, 1, 5, 9, 11, 20]
	4 is swapped with 1. Array now [1, 3, 4, 5, 9, 11, 20]
	4 is removed from heap as it is in its correct position now. Max-heap is created and 3 is swapped with 1. Array now [1, 3, 4, 5, 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)* (since it 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 the Figure 3, the algorithm uses the heap properties of a binary heap data structure, such as the efficient deleting 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 consistently fast running time (especially in the worst case) and efficient memory usage.

4. Bucket Sort

https://stackabuse.com/bucket-sort-in-python/

Optimal size of each bucket: Maximum value / Array length = 20/7 = 2.9

https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf 8.4 Bucket sort

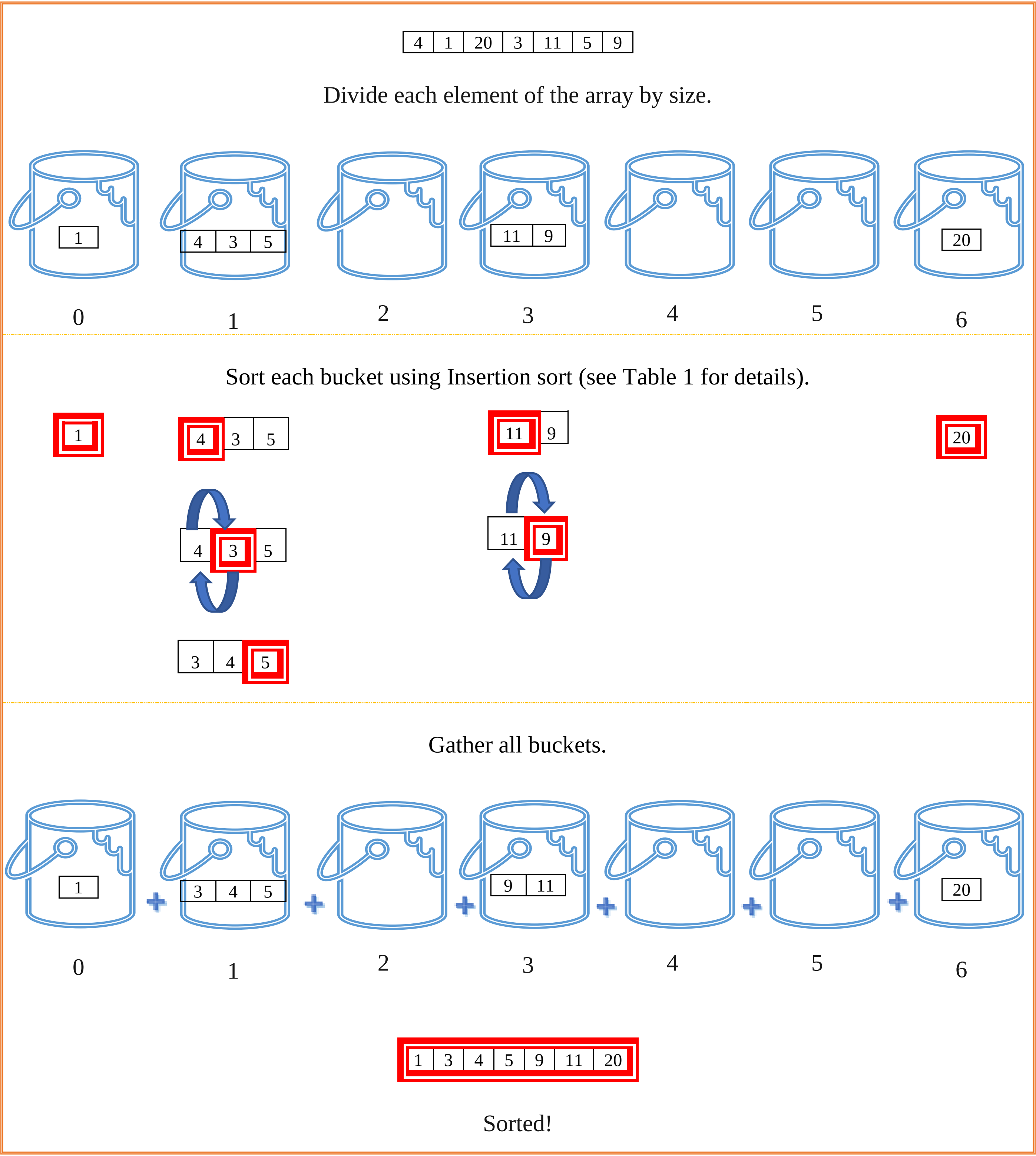
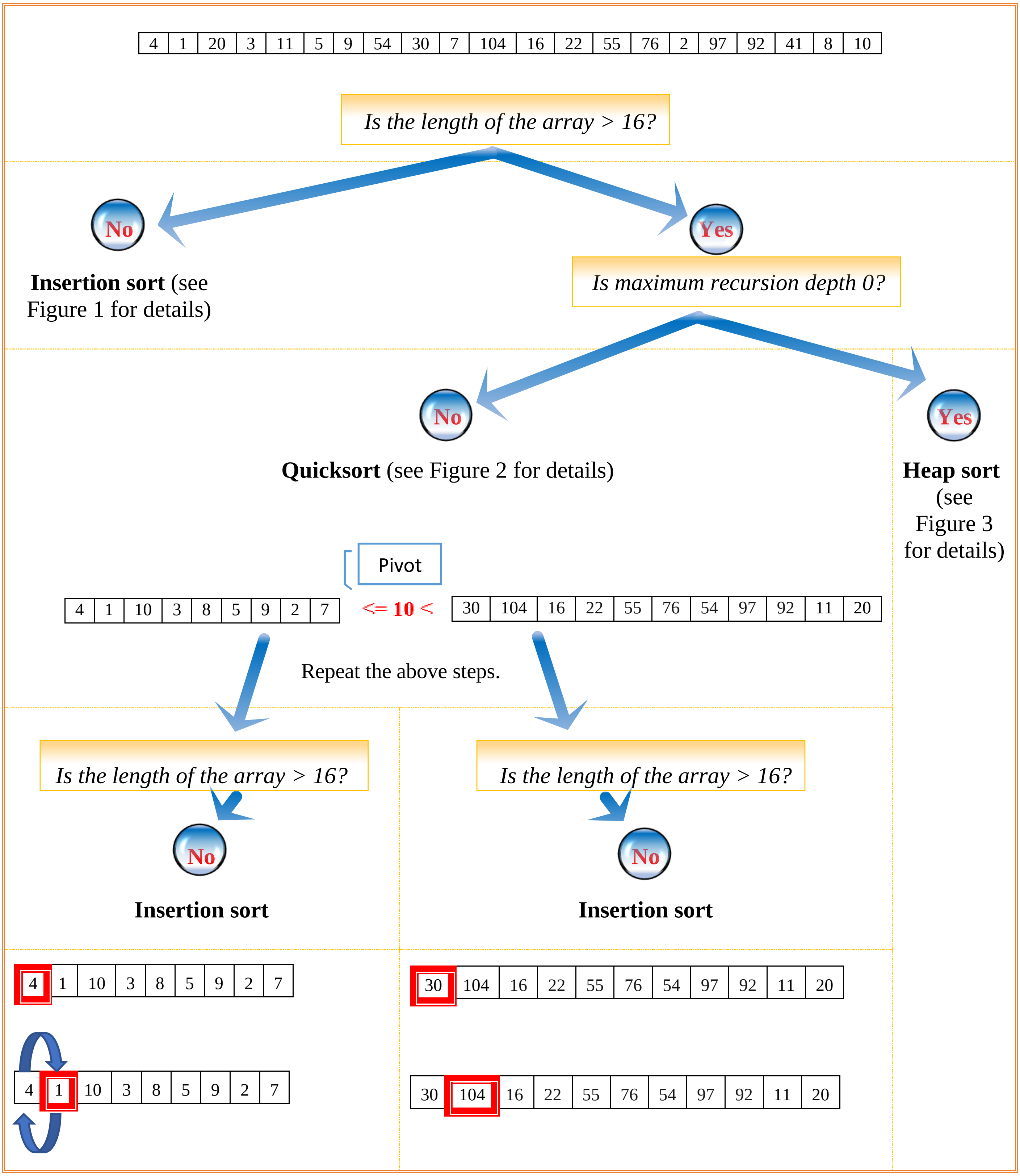
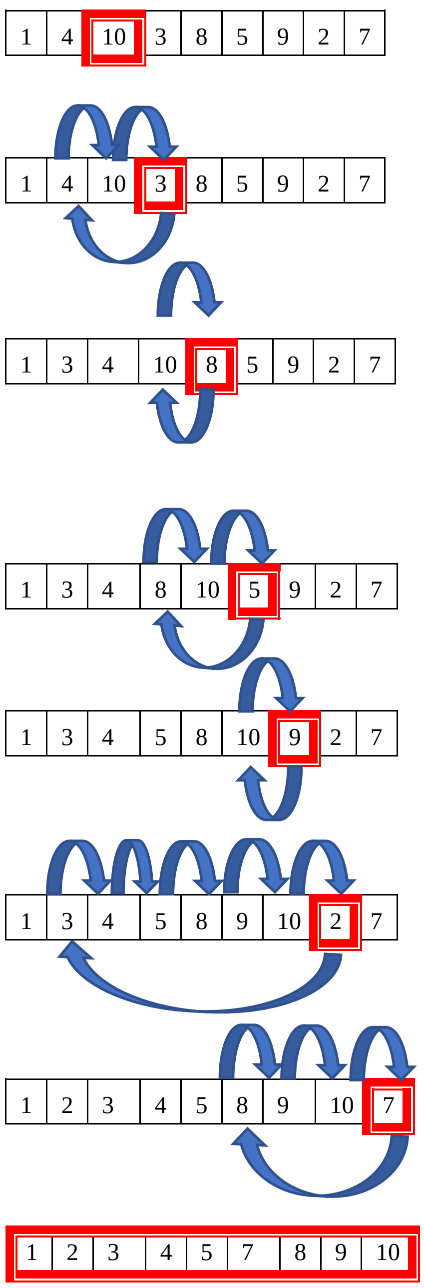


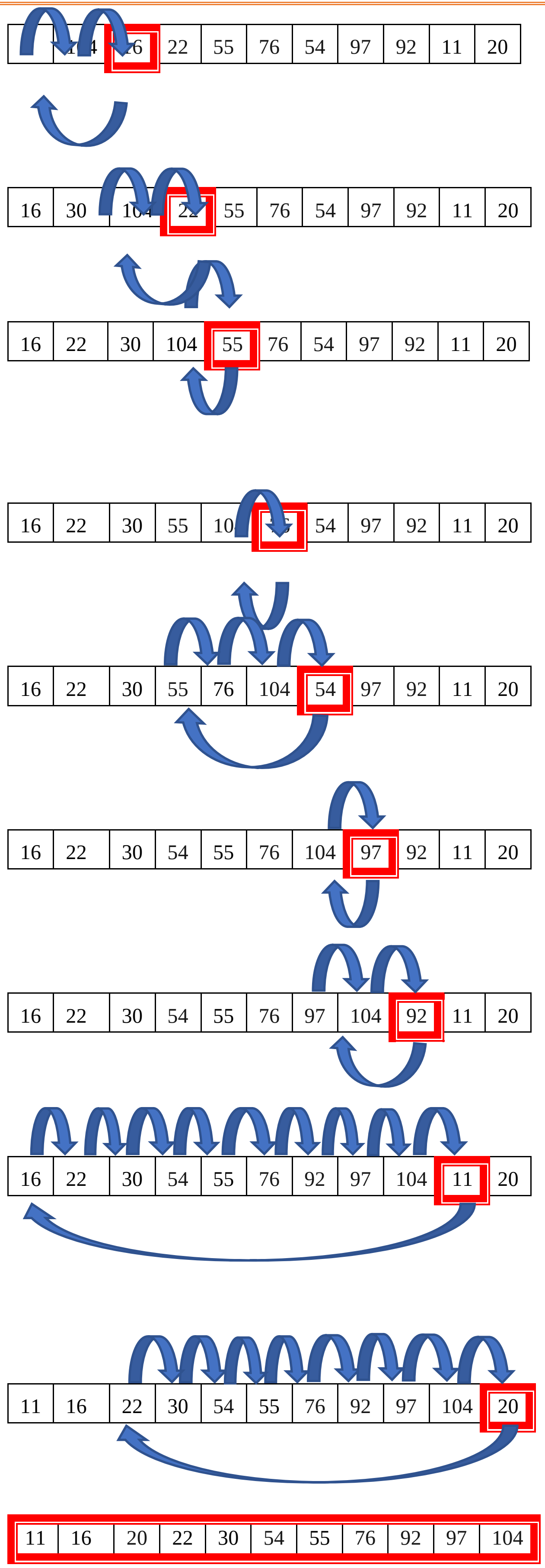
Figure 4 Bucket Sort Diagram

5. Introsort



4 1 20 3 11 5 9 54 30 7 104 16 22 55 76 2 97 92 41 8 10
Is the length of the array > 16?
No
Yes
Insertion sort (see Figure 1 for details)
Is maximum recursion depth 0?
No
Yes
Quicksort (see Figure 2 for details)
Heap sort (see Figure 3 for details)
Pivot
4 1 10 3 8 5 9 2 7
<= 10 <
30 104 16 22 55 76 54 97 92 11 20
Repeat the above steps.
Is the length of the array > 16?
No
Insertion sort
Is the length of the array > 16?
No
Insertion sort
4 1 10 3 8 5 9 2 7
30 104 16 22 55 76 54 97 92 11 20
4 1 10 3 8 5 9 2 7
30 104 16 22 55 76 54 97 92 11 20


1 4 10 3 8 5 9 2 7
1 4 10 3 8 5 9 2 7
1 3 4 10 8 5 9 2 7
1 3 4 8 10 5 9 2 7
1 3 4 5 8 10 9 2 7
1 3 4 5 8 9 10 2 7
1 2 3 4 5 8 9 10 7
1 2 3 4 5 7 8 9 10


22 55 76 54 97 92 11 20
16 30 104 22 55 76 54 97 92 11 20
16 22 30 104 55 76 54 97 92 11 20
16 22 30 55 10 54 97 92 11 20
16 22 30 55 76 104 54 97 92 11 20
16 22 30 54 55 76 104 97 92 11 20
16 22 30 54 55 76 97 104 92 11 20
16 22 30 54 55 76 92 97 104 11 20
11 16 22 30 54 55 76 92 97 104 20
11 16 20 22 30 54 55 76 92 97 104

Here we need to take a bigger array (for instance, with length of 21) since the array used in the previous examples will only be sorted using Insertion sort part of Introsort. Pick a pivot using a Median-of-three method. Here we reduce depth limit so if recursion reaches 9 with Quicksort, it will change to Heap sort (however, the example needs to be large to do so).

III. Implementation & Benchmarking

This section will describe the process followed when implementing the application above, and will present the results of your benchmarking. Discuss how the measured performance of the algorithms differed – were the results similar to what you would expect, given the time complexity of each chosen algorithm?

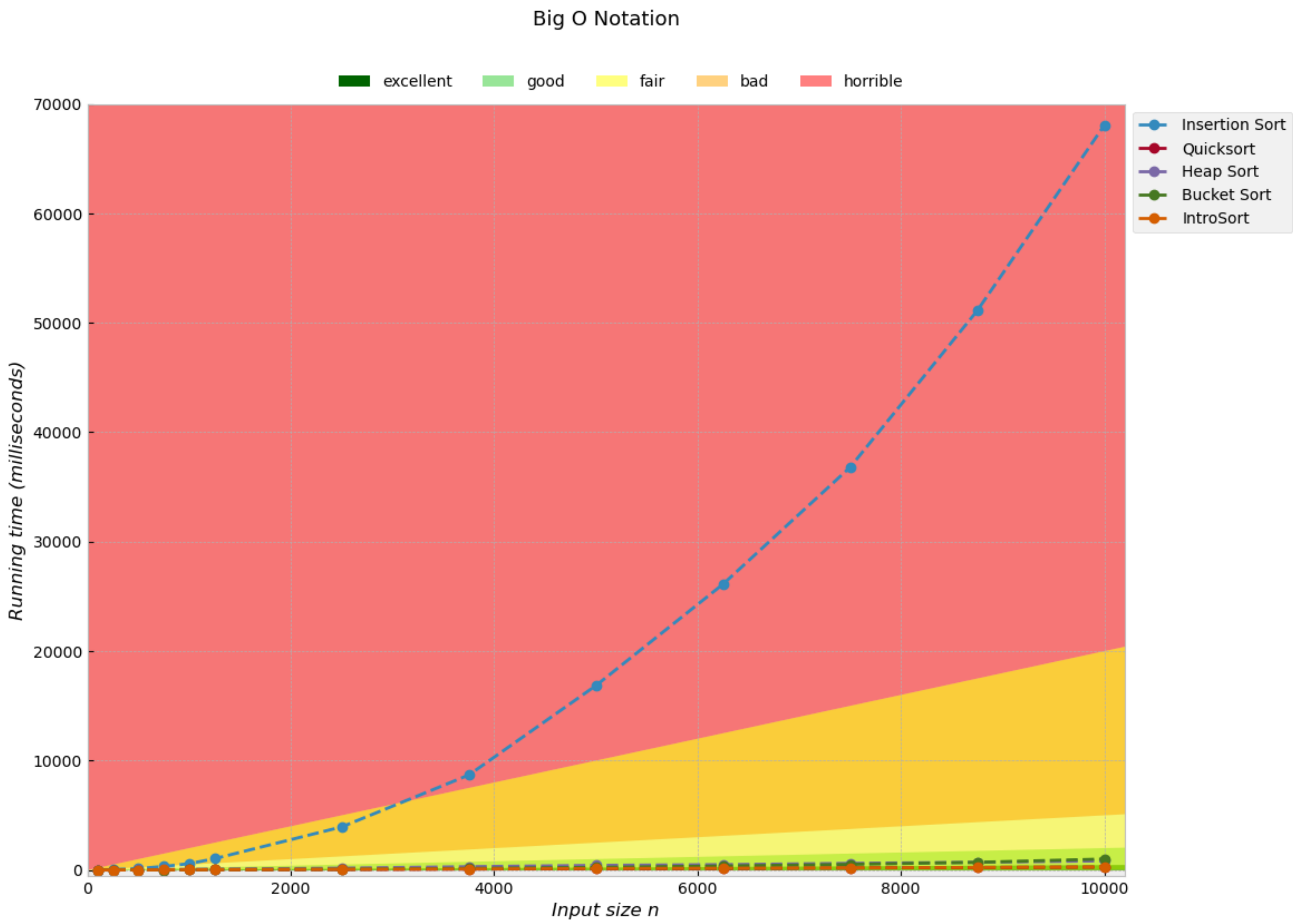
All five sorting algorithms (Insertion Sort, Quicksort, Heap Sort, Bucket Sort and Introsort) were implemented in Python and tested for the random sequence input of length from 100 to 10000. All five sorting algorithms were executed on machine Operating System having Intel(R) Core(TM) i7-7700HQ CPU @ 2.80 GHz and installed memory (RAM) 8.00 GB. The Plot of length of input and CPU time taken (in milliseconds) is shown in Figure 1. The result shows that for small input the performance for the five algorithms is almost nearest, but for the large input Quicksort and Introsort are the fastest and the Insertion sort the slowest.

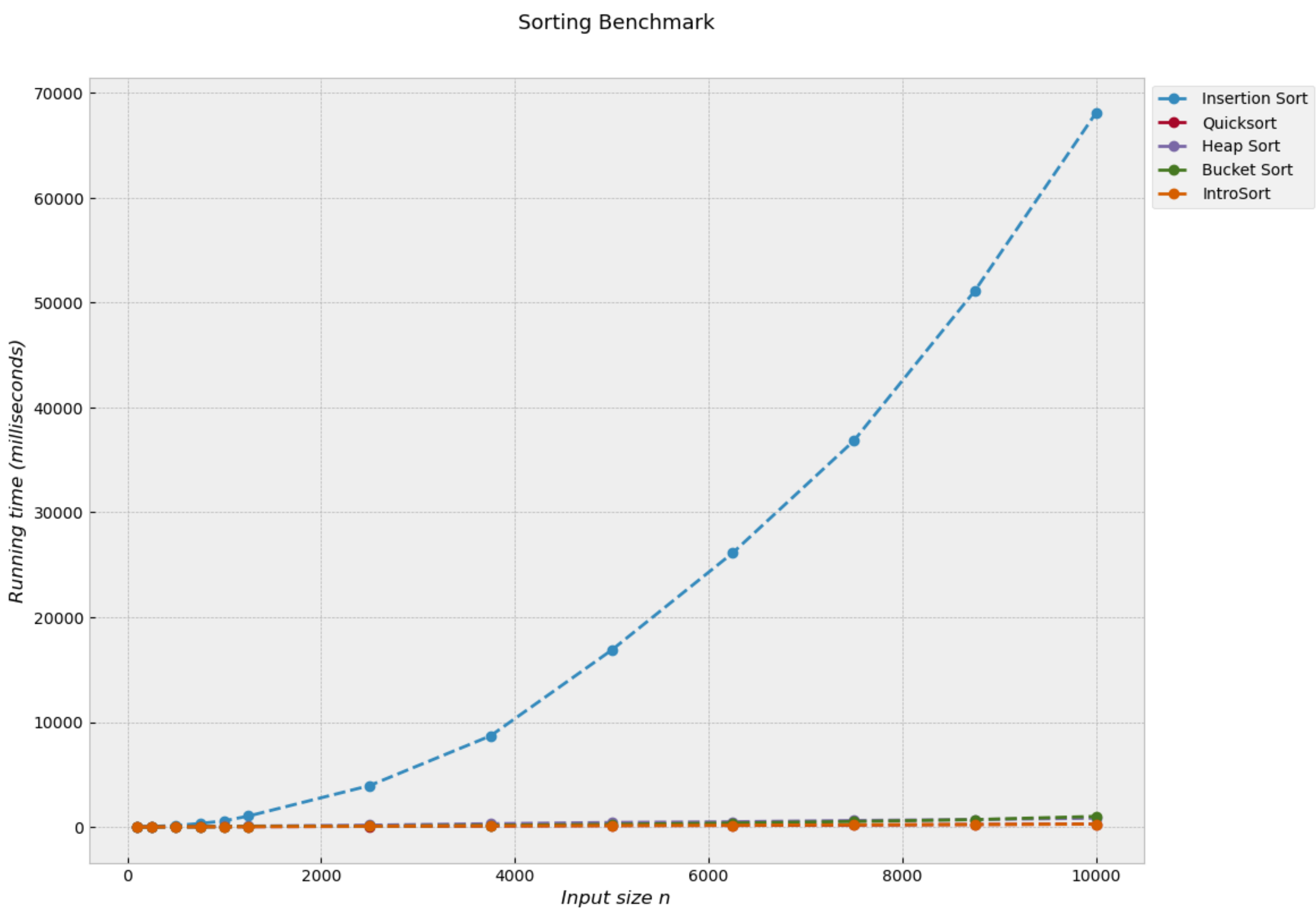
and measure the execution time of all programs with the same input data using the same computer. The built-in function (clock ()) in C++ is used to get the elapsed time of the implementing algorithms, execution time of a program is measured in milliseconds [6].The performances of GCS algorithm and a set of conventional sort algorithms are comparatively tested under average cases by using random test data from size 10000 to 30000. The result obtained is given in Table 1 to Table 6 for each Algorithm and the curves are shown in figure 1.

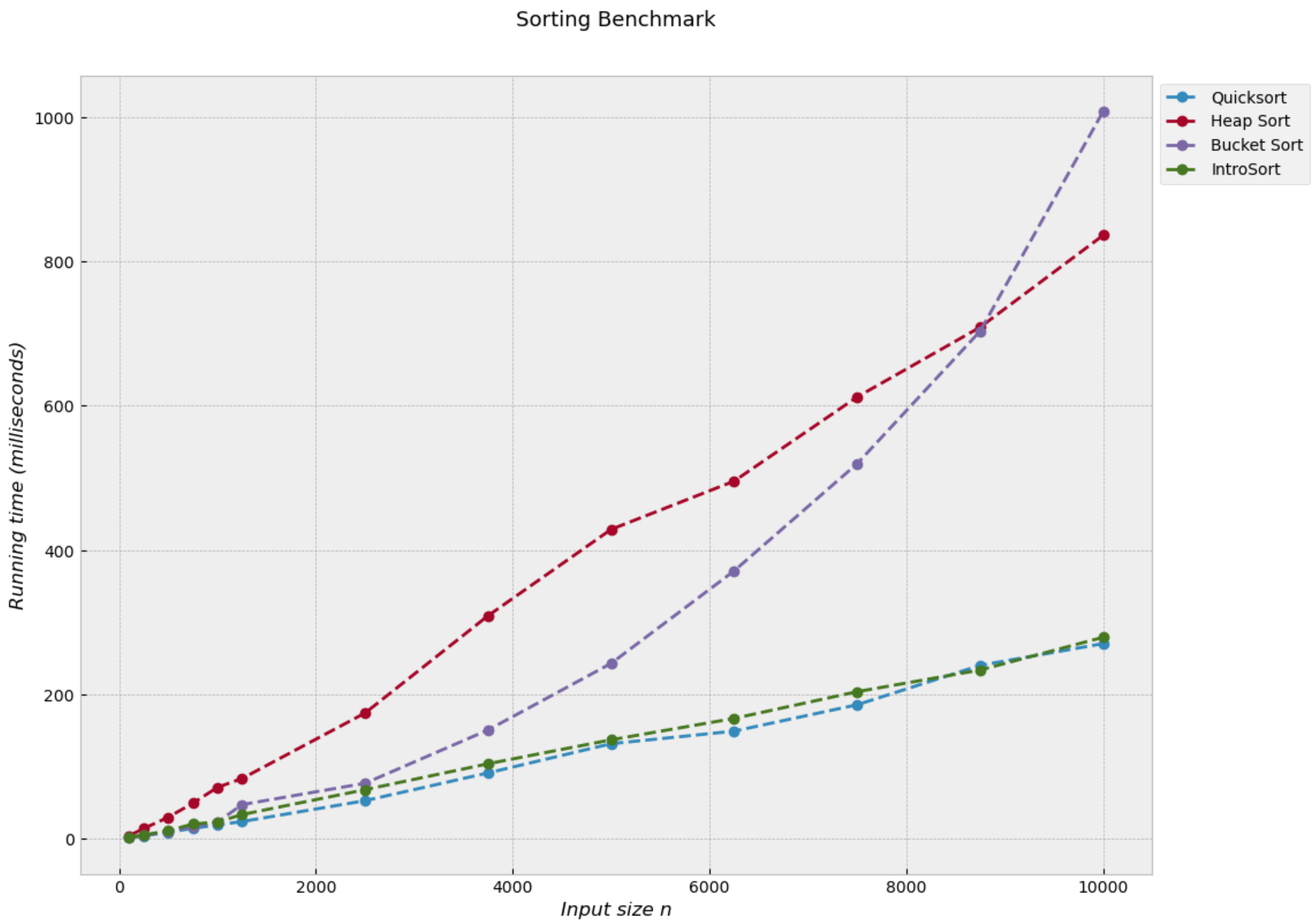
Sizes	***100***	***250***	***500***	***750***	***1000***	***1250***	***2500***	***3750***	***5000***	***6250***	***7500***	***8750***	***10000***
Insertion Sort	8.615	44.187	145.094	330.76	584.569	1038.718	3940.958	8683.307	16853.674	26118.586	36798.18	51102.482	68049.178
Quicksort	2.132	3.994	8.788	14.288	18.977	23.661	52.62	91.082	131.531	148.987	185.526	240.117	270.273
Heap Sort	3.938	14.024	29.678	49.135	71.095	83.327	174.13	309.035	428.624	495.514	612.139	708.832	836.248
Bucket Sort	1.598	4.179	10.328	16.611	22.959	47.141	76.723	150.642	242.875	371.101	519.634	703.357	1007.795

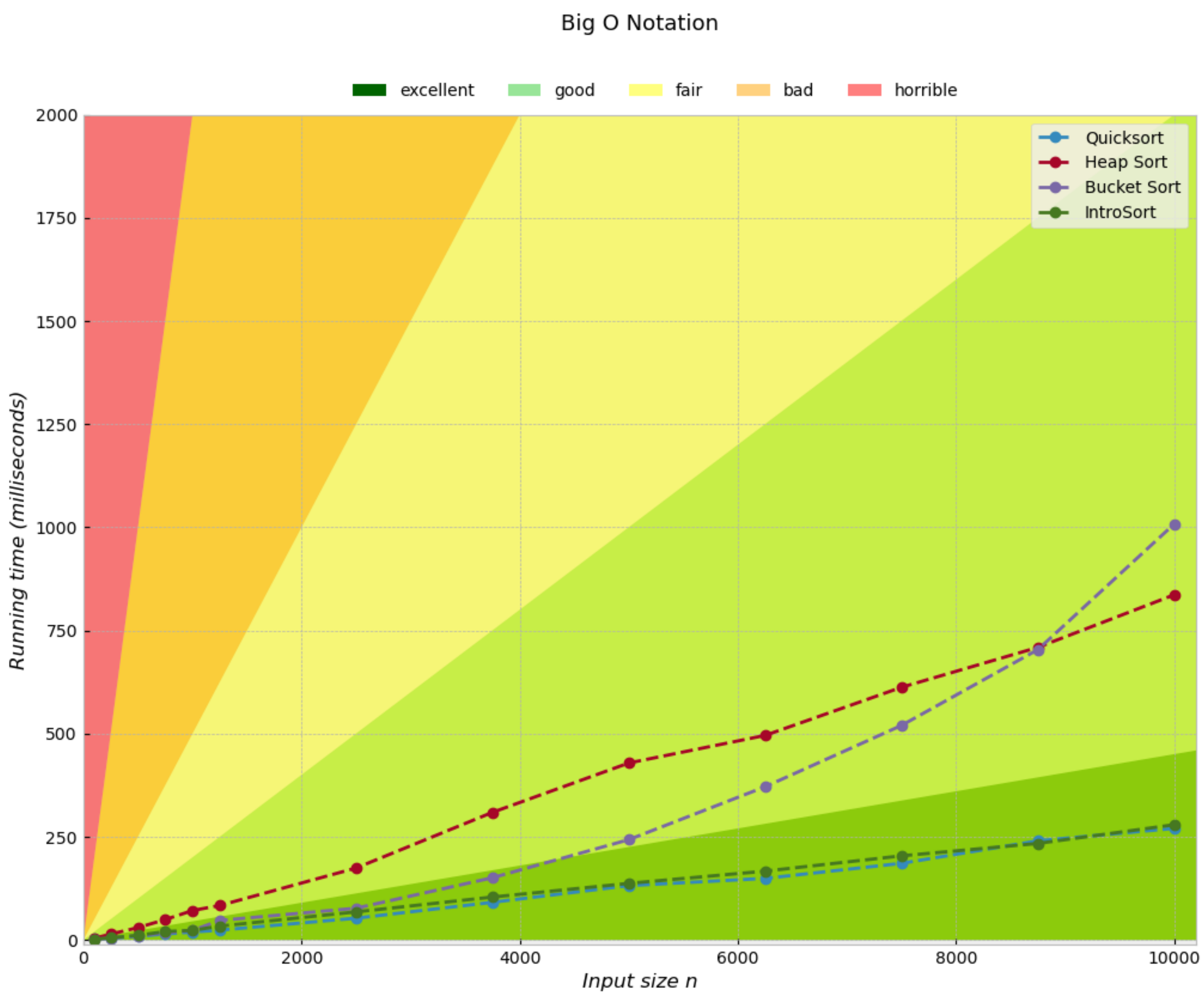
Introsort	1.674	5.043	11.098	20.134	23.068	33.364	67.835	103.779	136.952	166.592	203.662	233.689	279.067

Figure 1 Running time benchmark (the average of 10 repeated runs)


Big O Notation
excellent
good
fair
bad
horrible
Insertion Sort
Quicksort
Heap Sort
Bucket Sort
IntroSort
Running time (milliseconds)
Input size n
0
10000
20000
30000
40000
50000
60000
70000
2000
4000
6000
8000
10000


Sorting Benchmark
Insertion Sort
Quicksort
Heap Sort
Bucket Sort
IntroSort
Running time (milliseconds)
Input size n
0
10000
20000
30000
40000
50000
60000
70000
2000
4000
6000
8000
10000


Sorting Benchmark
Quicksort
Heap Sort
Bucket Sort
IntroSort
Running time (milliseconds)
Input size n
0
200
400
600
800
1000
2000
4000
6000
8000
10000


Big O Notation
excellent
good
fair
bad
horrible
Quicksort
Heap Sort
Bucket Sort
IntroSort
Running time (milliseconds)
Input size n
0
250
500
750
1000
1250
1500
1750
2000
4000
6000
8000
10000

IV. 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)