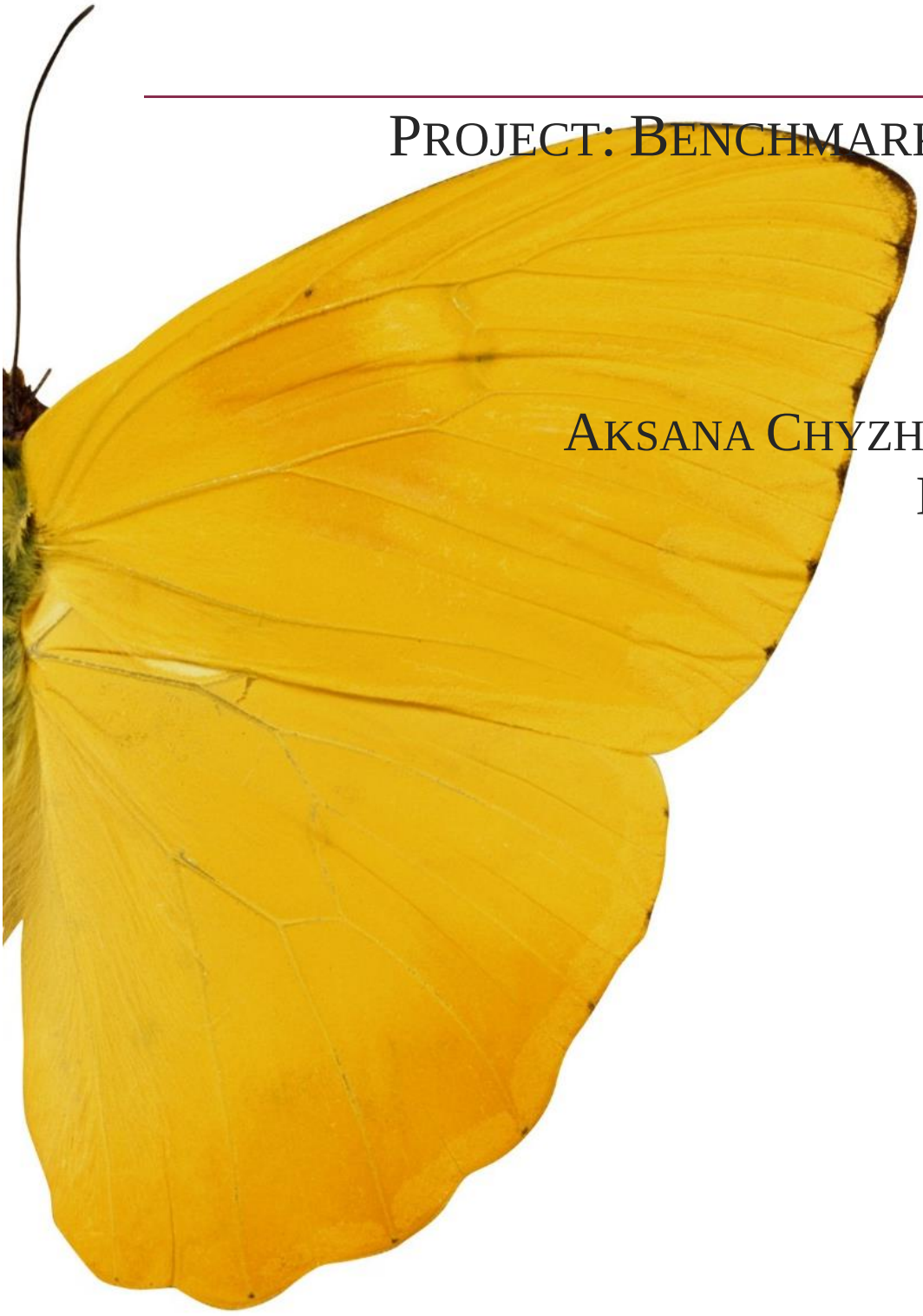


Project: Benchmarking Sorting Algorithms

By
Aksana Chyzheuskaya, PhD
ID: G00376371

GMIT
2020

TABLE OF CONTENTS

Acknowledgements 2
1. Introduction and Project Summary 3
1.2. Introduction to Sorting Algorithms 3
2. Sorting Algorithms Description 5
2.1. Bubble Sort 5
2.2. Quicksort 7
2.3. Bucket Sort 8
2.4. Heapsort 10
2.5. Introsort 12
3. Implementation and Results 13
3.1. Bubble Sort 13
3.2. Quicksort 14
3.3. Bucket sort 14
3.4. Heapsort 15
3.5. Introsort 15
3.6. Results and Discussion 16
4. References 20

TABLE OF FIGURES

Figure 1. Bubble Sort. 6
Figure 2. Example of the Quick sort algorithm process. 8
Figure 3. Bucket sort. 9
Figure 4. Build a heap. 11
Figure 5. Heapsort preocedure. 11
Figure 6 Introsort Procedure. 13
Figure 7. Time running algorithms – BEST CASE. 16
Figure 8. Best case: Bubble_sort vs quick_sort. 17
Figure 9. Average case – sorting a randomly shuffles array (left – with bubble_sort, right – without bubble_sort) 18
Figure 10. worst case –sorting through presorted array in descending order (left – with bubble_sort, right – without bubble_sort). 19

ACKNOWLEDGEMENTS

I would like to acknowledge the wide Internet community of programming learners, students, teachers, professionals and enthusiasts for the vast rich amount of materials that available free of charge on the Internet. This materials contributed greatly in my acquisition of knowledge directly or indirectly related to this subject.

I would like to thank my online-classmates who participate in discussion groups raising important questions and responding to queries, or just by being supportive and sharing their frustration to show we are all in the same boat.

I would like to thank lecturers who teach this semester – Dr. Dominic Carr who is a lecturer on this module but also Dr. Gerard Harrison and Dr. Michael Duignan who helped to improve coding skills through their comprehensive lecture materials.

I would like to thank my family for their support and energy that kept me going.

1. Introduction and Project Summary

1.2. Introduction to Sorting Algorithms

In this section I would like to talk about the importance of sorting algorithms in terms of learning and in term of computing and data analysis. I would also like to outline the project content and I envisioned it to be, what my main findings were and what I leant. There is a Results and Discussion section at the end of the project but this section will hopefully will give a ready a taste of something to look forward to.

I read an article by Hudgens (2016) on the importance of sorting algorithms and one of the first things he points out is that coding even a simple algorithm like Quicksort helped him to understand the programming language (Rubi in his case) like he has not before. So here it is a rationale to learn sorting algorithms and try to implement it in practice and code

To understand importance of algorithms from the practical point of view, it helps to start with a definition of sorting. In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order (Wikipedia.org, 2020d). Sorting algorithms help to place data in certain order. The most frequently used orders are numerical order and lexicographical order. Placing data in an order that makes it easy to perform CRUD (create, read, update and delete) operations is important because the less code is needed to make data access work, the better (Mueller and Massaron, 2017). Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalising data and for producing human-readable output (Wikipedia.org, 2020d). The efficiency of sorting algorithms is discussed in more detail in the next section.

There is a variety of sorting algorithm that have been developed. There are a number of ways to classify algorithms. For example, in lecture notes/videos (Mannion, 2017c), they are classifies into: comparison, non-comparison and hybrid. But there are other ways to classify algorithms based on their characteristics. Looking at classification is informative to gain a quick incites what characteristics of sorting algorithms may differ by.

So sorting algorithms can be classified by (Wikipedia.org, 2020d):

1. *Computational complexity* (worst, average and best behavior) in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n), with parallel sort in O(log2 n), and bad behavior is O(n2). (See Big O notation.) Ideal behavior for a serial sort is O(n), but this is not possible in the average case. Optimal parallel sorting is O(log n). Comparison-based sorting algorithms need at least Ω(n log n) comparisons for most inputs.
2. *Computational complexity of swaps* (for "in-place" algorithms).
3. *Memory usage* (and use of other computer resources). In particular, some sorting algorithms are "in-place". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(n)) additional memory is considered "in-place".

4. *Recursion*. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
5. *Stability*: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
6. *Whether or not they are a comparison sort*. A comparison sort examines the data only by comparing two elements with a comparison operator.
7. *General method*: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
8. *Serial vs parallel*. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
9. *Adaptability*: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

These algorithms suit different purposes, however, there is way to analyse their performance in general terms with desirable characteristics including:

- Stability
- Good run time efficiency
- In-place sorting
- Suitability (Mannion, 2017a)

STABILITY

Stable sort algorithms sort repeated elements in the same order that they appear in the input. When sorting some kinds of data, only part of the data is examined when determining the sort order. This allows the possibility of multiple different correctly sorted versions of the original list. Stable sorting algorithms choose one of these, according to the following rule: if two items compare as equal, then their relative order will be preserved, so that if one came before the other in the input, it will also come before the other in the output (Wikipedia.org, 2020d).

Stability is important for the following reason: say that student records consisting of name and class section are sorted dynamically on a web page, first by name, then by class section in a second operation. If a stable sorting algorithm is used in both cases, the sort-by-class-section operation will not change the name order; with an unstable sort, it could be that sorting by section shuffles the name order. Using a stable sort, users can choose to sort by section and then by name, by first sorting using name and then sort again using section, resulting in the name order being preserved (Wikipedia.org, 2020d).

More formally, if a comparator function determines that two elements ai and aj in the original unordered collection are equal, it may be important to maintain their relative ordering in the sorted set. For example, if i < j, then the final location for A[i] must be to the left of the final location for [j].

Sorting algorithms that guarantee this property are stable. Unstable sorting algorithms do not preserve this property (Mannion, 2017a).

The data being sorted can be represented as a record or tuple of values, and the part of the data that is used for sorting is called the key. In the card example, cards are represented as a record (rank, suit), and the key is the rank. A sorting algorithm is stable if whenever there are two records R and S with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list (Wikipedia.org, 2020d).

When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different (Wikipedia.org, 2020d).

There is no ideal algorithm that fits all purposes. Different algorithms are better at serving different purposes. At the moment no perfect search strategy exists, but the exploration for such method continues (Mueller and Massaron, 2017).

1.2. Summary of the Project

As instructed in Assignment Brief, five algorithms were chosen for analysis in this project:

- Bubble Sort
- Quicksort
- Bucket sort
- Heapsort
- Introsort

The sorting algorithms were chosen at random. While the first and the second algorithms belong to comparison sorting algorithms group, the third and fourth – non-comparison. The fifth one is a hybrid algorithm. These algorithms are described in detail in the following section.

The code snippets for these algorithms are sourced online and adapted, it was referenced appropriately. The benchmarking part is written by me.

2. Sorting Algorithms Description

The general description of sorting algorithms and their purpose were outlined in the previous section. In this section I'll talk about the types of sorting algorithms, their difference and introduce 5 sorting algorithms that I chose for implementation and benchmarking in this project in more detail.

2.1. Bubble Sort

Bubble Sort is one of the first and the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order (geeksforgeeks.org, 2020a). It was named "Bubble

Sort" for the way larger values in a list "bubble up" to the end as sorting takes place. It was first analised circa 1956. The time complexity for this algorithm is n in best case, and n^2 in worst and average cases). It is a comparison-based and in-place sorting algorithm. It uses a constant amount of additional working space in addition to the memory required for the input. It is very simple to understand and implement, however as will be shown in the Implementations and Results section of this project, it is slow and impractical for most problems even when compared to Insertion Sort. It can be practical in some cases on data which is nearly sorted (Mannion, 2017b).

Characteristics (geeksforgeeks.org, 2020a):

- Worst and Average Case Time Complexity: ***O(n²)***. Worst case occurs when array is reverse sorted.
- Best Case Time Complexity: ***O(n)***. Best case occurs when array is already sorted.
- Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.
- Sorting In Place: Yes
- Stable: Yes

Implementation on Bubble Sort is depicted in Figure 1.

FIGURE 1. BUBBLE SORT.[1]

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
		1	2						

[1] Source: GEEKSFORGEEKS.ORG. 2020a. *Bubble Sort* [Online]. Available: https://www.geeksforgeeks.org/bubble-sort/ [Accessed].

Procedure for Bubble Sort algorithm is well described by Mannion (2017b):

- Compare each element (except the last one) with its neighbour to the right
    - If they are out of order, swap them. This puts the largest element at the very end. The last element is now in the correct and final place
  - Compare each element (except the last two) with its neighbour to the right
    - If they are out of order, swap them. This puts the second largest element next to last. The last two elements are now in their correct and final places.
  - Compare each element (except the last three) with its neighbour to the right…
  - Continue as above until there are no unsorted elements on the left.

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm. In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm (geeksforgeeks.org, 2020a, Wikipedia.org, 2020d).

2.2. QUICKSORT

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of this algorithms that differ by the ways pivot is selected (geeksforgeeks.org, 2020e, Skiena, 2009):

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot (geeksforgeeks.org, 2020e).

Characteristics (geeksforgeeks.org, 2020a):

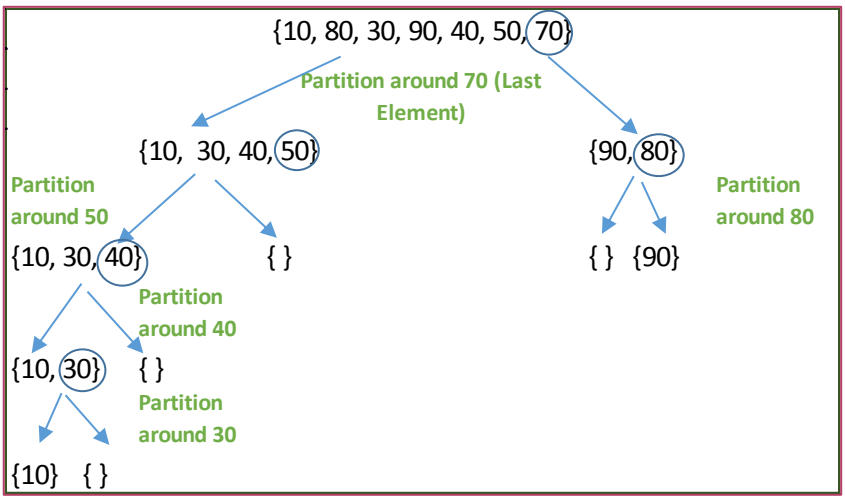
- Worst and Average Case Time Complexity: $O(n^2)$ and $n(logn)$ respectively. The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.
- Best Case Time Complexity: $n\ log(n)$. The best case occurs when the partition process always picks the middle element as pivot.
- Sorting In Place: Yes. Considering a broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input (geeksforgeeks.org, 2020e).
- Stable: No. The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

The time taken by Quick Sort depends upon the input array and partition strategy.

The process for the Quick Sort:

The key process in Quick Sort is partition. An example of Quick Sort process is depicted in Figure 2. Target of partitions is, given an array and an element x of array as pivot, to put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. This process is estimated to be done in linear time (geeksforgeeks.org, 2020e).

FIGURE 2. EXAMPLE OF THE QUICK SORT ALGORITHM PROCESS[2].



Although the worst case time complexity of Quick Sort is O(n2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. Quick Sort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data (geeksforgeeks.org, 2020e). Arguably, this algorithm, if implemented correctly, can be three times quicker than its main competitors, merge sort and heapsort (Skiena, 2009).

2.3. BUCKET SORT

Bucket Sort is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm. The computational complexity depends on the algorithm used to sort each bucket, the number of buckets to use, and whether the input is uniformly distributed (Wikipedia.org, 2020a).

[2] The chart is by Aksana Chyzheuskaya, adapted from :GEEKSFORGEEKS.ORG. 2020e. *QuickSort* [Online]. Available: https://www.geeksforgeeks.org/quick-sort/ [Accessed].

Characteristics:

- Worst Case: ***O(n²)***, where k is a number of buckets. The worst-case scenario occurs when all the elements are placed in a single bucket. The overall performance would then be dominated by the algorithm used to sort each bucket, which is typically ***O(n²)*** insertion sort, making bucket sort less optimal than ***O(n(log(n)))*** comparison sort algorithms like Quicksort (Wikipedia.org, 2020a).
- Average Case Time Complexity: ***O(n+n²/k+k)***.
- Best Case Time Complexity: ***O(n*k)***. Best case occurs when array is already sorted.
- Sorting In Place: Yes
- Stable: Yes

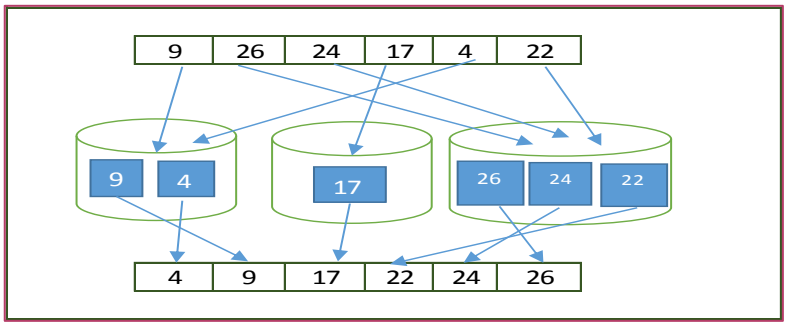
Use Bucket Sort works best when the input data is uniformly distributed for a given range (Heineman et al., 2016, geeksforgeeks.org, 2020b). Bucket Sort is not appropriate for sorting arbitrary strings. However, it could be used to sort a set of uniformly distributed floating-point numbers in the range [0, 1) (Heineman et al., 2016).

Bucket sort works as follows:

- Set up an array of initially empty "buckets".
- Scatter: Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather: Visit the buckets in order and put all elements back into the original array (Wikipedia.org, 2020a).

Normally insertion sort would be used, but other algorithms could be used as well, including Bucket Sort itself. In the implementation of this algorithm for this assignment Insertion Sort is used to sort each bucket. The example of the Bucket Sort is depicted in Figure 3.

FIGURE 3. BUCKET SORT[3].



[3] The chart is by Aksana Chyzheuskaya, adapted from: SEHGAL, K. 2018. An Introduction to Bucket Sort.

2.4. HEAPSORT

HeapSort is a comparison based sorting technique based on Binary Heap data structure (geeksforgeeks.org, 2020c). Heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step (Wikipedia.org, 2020b).

Characteristics:

- Worst Case: O(n log n)
- Average Case Time Complexity: : O(n log n)
- Best Case Time Complexity: : O(n log n)
- Sorting In Place: Yes
- Stable: No

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data (depicted in Figure 4).

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree (see Figure 5).

3. Repeat above steps while size of heap is greater than 1 (geeksforgeeks.org, 2020c).

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used (geeksforgeeks.org, 2020c, Miyaki, 2020).

In this project the HeapSort is used twice – on its own and as a part of IntraSort, the hybrid sorting algorithm described in the following section.

HeapSort can be described as having two procedure. The first one is building a heap. During this stage either Top-Down or Bottom Up approach can be used, In this project Top-Down approach was adopted (Figure 4). Building a heap is a recursive procedure.

FIGURE 4. BUILD A HEAP.

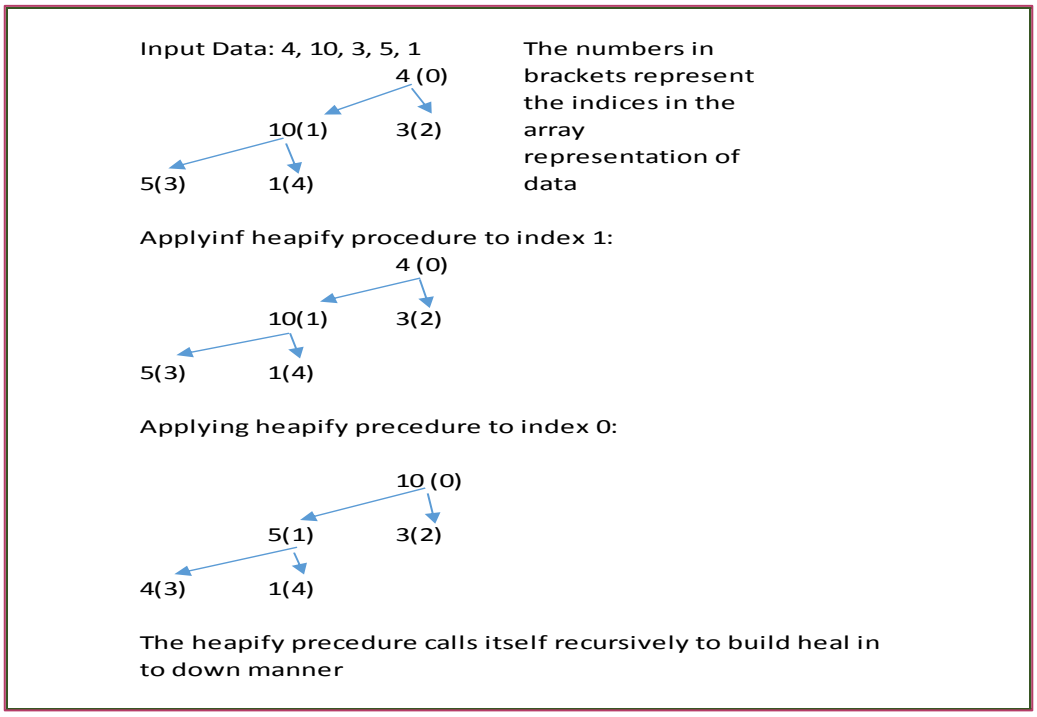
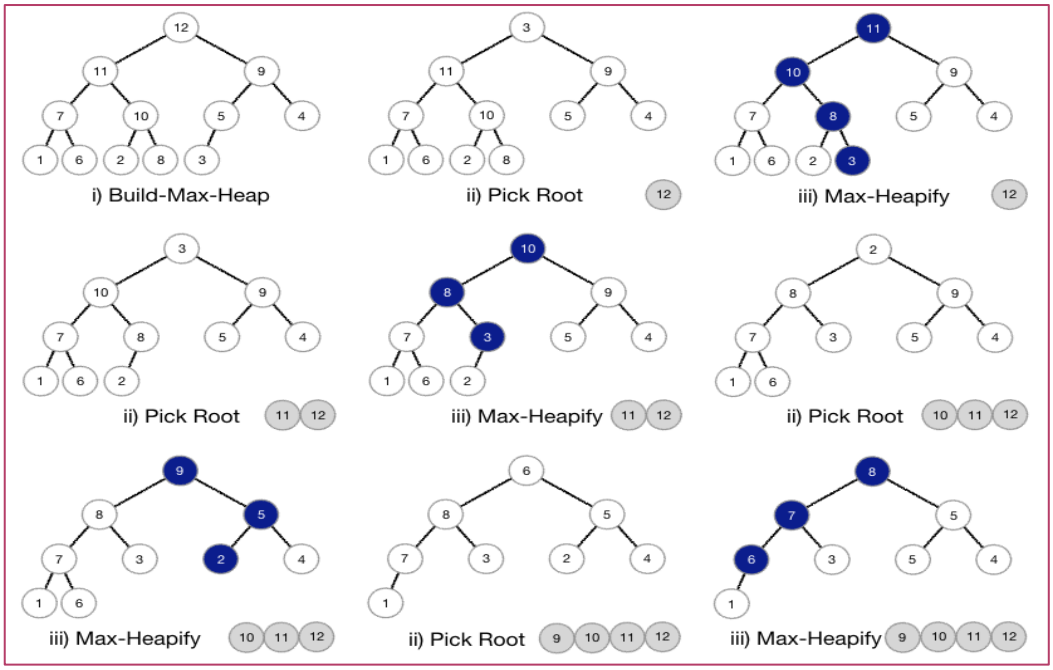


FIGURE 5. HEAPSORT PREOCEDURE.[4]



[4] Source: MIYAKI, K. 2020. Basic Algorithms — Heapsort. Sorting an array with "Heap" data structure.

2.5. INTROSORT

IntroSort (Introspective sort) is a hybrid comparison based sort that consists of three sorting phases, which include: QuickSort, HeapSort, and Insertion Sort. QuickSort was described in Section 2.2. of this project. The HeapSort was described in Section 2.4 of this project document. The following section shows how the Introsort algorithm is formulated (geeksforgeeks.org, 2020d).

Introsort provides both fast average performance and optimal worst-case performance. It begins with quicksort, it switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted and it switches to insertion sort when the number of elements is below some threshold (Figure 6). This combines the good parts of the three algorithms, with practical performance comparable to quicksort on typical data sets and worst-case O(n log n) runtime due to the heap sort. Since the three algorithms it uses are comparison sorts, it is also a comparison sort (Wikipedia.org, 2020c). There are a lot of hybrid algorithms that outperforms the general sorting algorithms. One such is the Introsort. Combining all the pros of the sorting algorithms, Introsort behaves based on the data set (geeksforgeeks.org, 2020d).

Characteristics:

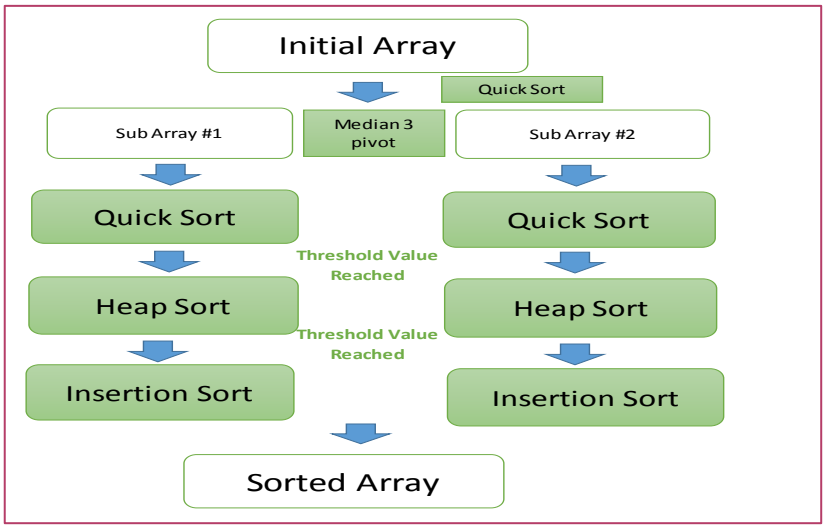
- Worst Case: O(n log n)
- Average Case Time Complexity: : O(n log n)
- Best Case Time Complexity: : O(n log n)
- Sorting In Place: Yes
- Stable: No

Here is how IntroSort is formulated:

As Musser (1997) describes himself, “IntroSort (for \introspective sort"), a new, hybrid sorting algorithm that behaves almost exactly like median-of-3 quicksort for most inputs (and is just as fast) but which is capable of detecting when partitioning is tending toward quadratic behavior. By switching to heapsort in those situations, introsort achieves the same O(N log N) time bound as heapsort but is almost always faster than just using heapsort in the first place. On a median-of-3 killer sequence of length 100,000, for example, introsort switches to heapsort after 32 partitions and has a total running time less than 1/200-th of that of median-of-3 quicksort. In this case it would be somewhat faster to use heapsort in the first place, but for almost all randomly-chosen integer sequences introsort is faster than heapsort by a factor of between 2 and 5.” (Musser, 1997).

Quicksort is faster on average, but can run slow on degenerate data. Heapsort is slower on average but doesn’t suffer by those degenerate cases. So these two combine quite well. Introsort is quite good choice for large number of situations. But unsuitable if stable sort is needed.

FIGURE 6 INTROSORT PROCEDURE[5].



3. IMPLEMENTATION AND RESULTS

The full Python code file is enclosed to this project separately.

Each section below contains code snippets. The code is commented throughout so no additional comments are offered. The RESULTs and discussion are offered in the Section 3.6. of this project.

3.1. BUBBLE SORT

```
# Adapted from: https://realpython.com/sorting-algorithms-python/
def bubble_sort(array):
    n = len(array)

    for i in range(n):
        # Look at each item of the list one by one,compare with the next value.
        # With each iteration, array to sort gets smaleer
        for j in range(n - i - 1):
            # Terminate if sorted
            already_sorted = True

            if array[j] > array[j + 1]:      # If the item is greater than the next -swap
                array[j], array[j + 1] = array[j + 1], array[j]

                already_sorted = False # chnage already sorted to False

        if already_sorted: # break when sorted
            break
```

[5] By Aksana Chyzheuskaya

3.2. QUICKSORT

```
# Adapted from: https://realpython.com/sorting-algorithms-python/
def quick_sort(array):

    if len(array) < 2: # if the input array contains fewer than two elements,
        return array   # return the array

    low, same, high = [], [], [] # create 3 empty arrays

    pivot = array[randint(0, len(array) - 1)] # randomely select pivot

    for item in array:

        if item < pivot:        # if an element < pivot
            low.append(item)    # append to "low" array
        elif item == pivot:     # if an element = pivot
            same.append(item)   # append to "same" array
        elif item > pivot:      # if an element > pivot
            high.append(item)   # append to "high array

    return quick_sort(low) + same + quick_sort(high) # combine results
    return array
```

3.3. BUCKET SORT

```
def bucket_sort(array):
    largest = max(array) # largest element
    length = len(array)  # length of an array
    size = largest/length

    buckets = [[] for _ in range(length)]  # create empty buckets as a list of empty lists
    for i in range(length): # determine what bucket array(i) belongs to
                            # an append value to the bucket
        j = int(array[i]/size)
        if j != length:
            buckets[j].append(array[i])
        else:
            buckets[length - 1].append(array[i])

    for i in range(length):
        insertion_sort(buckets[i]) # perfotm insertion sort on each bucket

    result = []
    for i in range(length):
        result = result + buckets[i] # concatenate all buckets together

    return result
```

```
def insertion_sort(array):
    for i in range(1, len(array)): # loop over the elements of the list
        temp = array[i]  # creat temp var equal to an element at index i
        j = i - 1        # set a counter
        while (j >= 0 and temp < array[j]): # if j is not negative and
                                            # an element at index i < than preceeding element
            array[j + 1] = array[j] # set the element at index j + 1 equal to the element at index j
            j = j - 1               # increment counter
        array[j + 1] = temp         # set the element at index j + 1 equal to temp
```

3.4. HEAPSORT

```
# Adapted from: https://www.tutorialspoint.com/python-program-for-heap-sort
def heapify(arr, n, i):
   largest = i # largest value
   l = 2 * i + 1 # left
   r = 2 * i + 2 # right
   if l < n and arr[i] < arr[l]: # if left child exists
      largest = l
   if r < n and arr[largest] < arr[r]: # if right child exits
      largest = r
   # root
   if largest != i:
      arr[i],arr[largest] = arr[largest],arr[i] # swap
      # root.
      heapify(arr, n, largest)
# sort
def heap_sort(arr):
   n = len(arr)
   # maxheap
   for i in range(n, -1, -1):
      heapify(arr, n, i)
   # element extraction
   for i in range(n-1, 0, -1):
      arr[i], arr[0] = arr[0], arr[i] # swap
      heapify(arr, i, 0)
```

3.5. INTROSORT

```
# Adapted from: https://www.sanfoundry.com/python-program-implement-introsort/
def intro_sort(alist):
    maxdepth = (len(alist).bit_length() - 1)*2 #maxdepth = to 2 times floor of log base 2 of the length of the list.
    introsort_helper(alist, 0, len(alist), maxdepth) # call introsort_helper

def introsort_helper(alist, start, end, maxdepth):
    if end - start <= 1: # if the length of the list to be sorted is not greater than 1
        return           # return
    elif maxdepth == 0:  # if maxdepth is 0
        heapsort(alist, start, end) # perform a heapsort
    else:
        p = partition(alist, start, end) # therwise, call partition
        introsort_helper(alist, start, p + 1, maxdepth - 1) # call introsort_helper on 2 halfs of the list
        introsort_helper(alist, p + 1, end, maxdepth - 1)
```

The rest of the IntroSort Algorithm's code can be found in the code file.

3.6. RESULTS AND DISCUSSION

The results below outline the running time of each of the algorithms chosen in the BEST case scenario (Table 1 and Figure 7) – sorting an array of all ones, in the AVERAGE case scenario (Table 2 and Figure 9) – sorting an array of randomly shuffled numbers , and the WORST case scenario (Table 3 Figure 10) – sorting through presorted array in descending order.

In the best case QuickSort and BubbleSort have the best performance, which is not surprising as best time complexity is for BubbleSort is ***O(n)***. QuickSort has a very similar performance to BubbleSort despite having tame complexity ***O(n log n)***. A closer look at these two sorting algorithms performances (see Figure 8) shows that Bubble Sort has time complexity function looks very similar to QuickSort and the line looks more ***O(n log n)*** than a straight line.

In the best case scenario IntroSort has the worst time complexity followed by BucketSort.

TABLE 1. BEST CASE – SORTING THROUGH ARRAY OF 1-S, TIME IN MILLISECONDS

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
bubble_sort	0	0.1	0.2	0.3	0.3	0.8	1.2	1.8	2.6	2.9	4.1	4.6	5.4
quick_sort	0.2	0	0.2	0.2	0.4	0.6	0.9	1.6	2.1	2.1	2.9	3.6	3.8
bucket_sort	0.3	1.9	2.7	4.7	6.5	8	16.2	22.5	29.8	36.1	40.1	48.1	53.5
heap_sort	0.4	0.6	1.6	2.8	3.1	3.8	7.4	12.1	16.1	19.6	23.1	29.6	34.3
intro_sort	0.3	2.2	4.2	6.5	9.7	12.1	25.2	38.4	50.6	65.8	77.8	92.1	113.9

FIGURE 7. TIME RUNNING ALGORITHMS – BEST CASE.

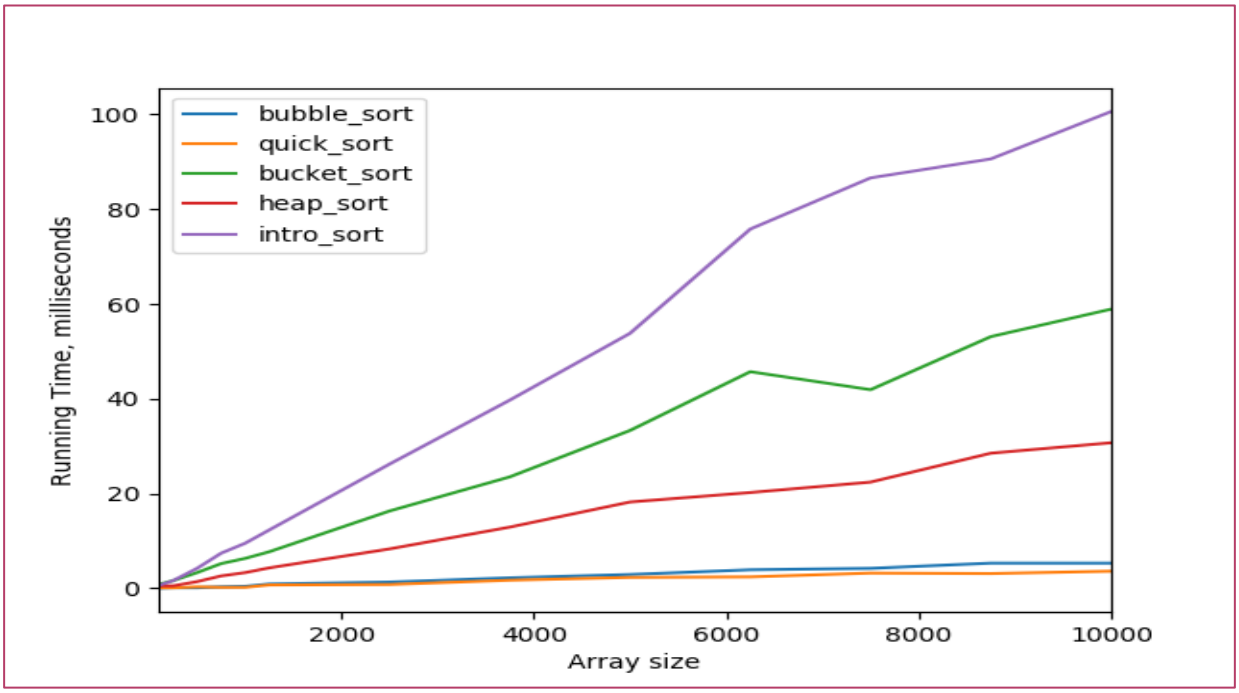
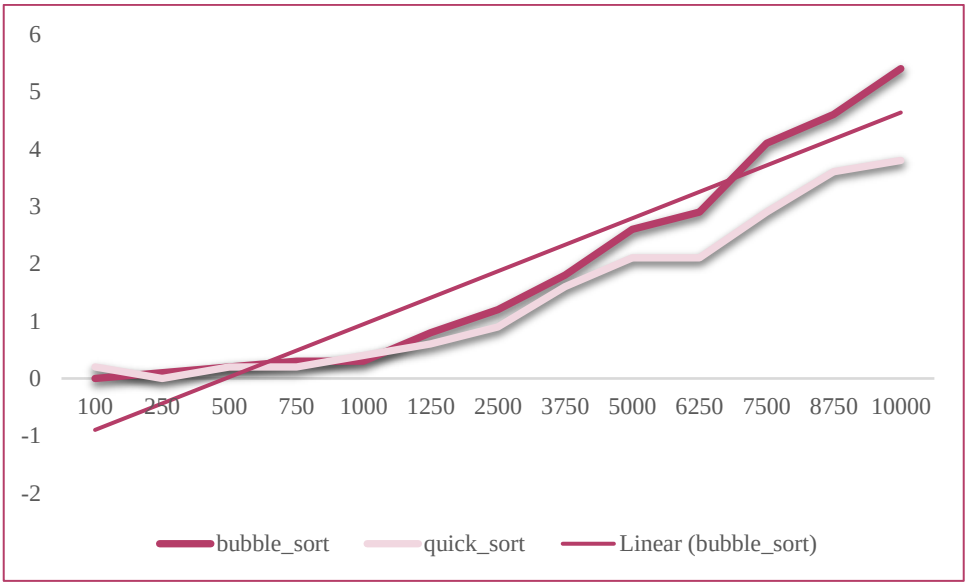


FIGURE 8. BEST CASE: BUBBLE_SORT VS QUICK_SORT.

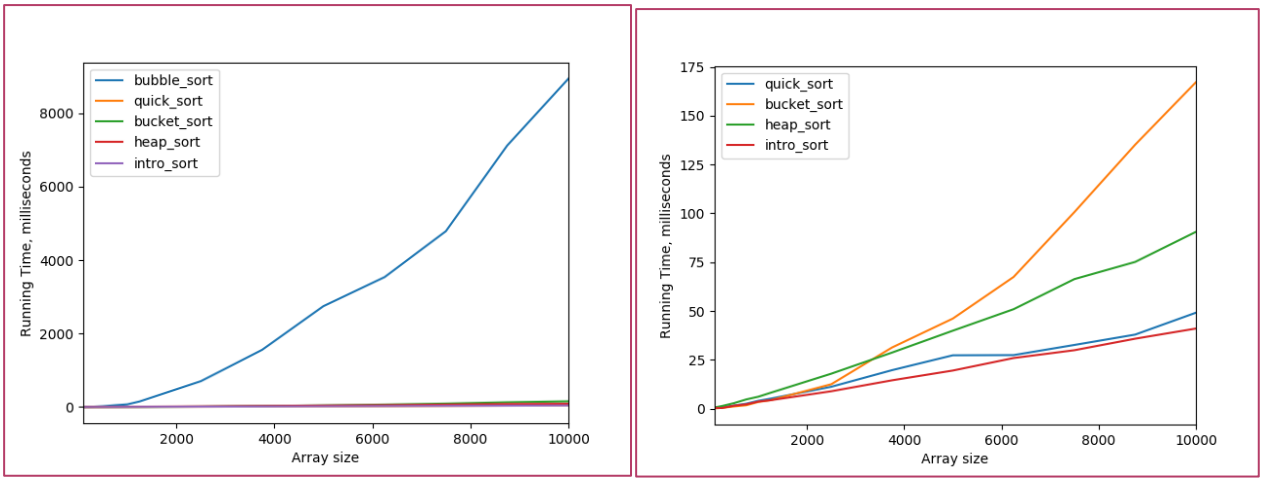


When the algorithms are sorting through the randomly shuffled arrays, BubbleSort stands out as the worst performing algorithm (Figure 9, on the left). It looks like the rest of the algorithms are performing in the similar way, however, when they are plotted separately (Figure 9, on the right), we can see, that the best AVERAGE case performance belongs to IntroSort, closely followed by QuickSort.

TABLE 2. AVERAGE CASE – SORTING A RANDOMLY SHUFFLES ARRAY:

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
bubble_sort	0.9	4.7	21.4	48	72.3	151.4	702.6	1557	2746	3538	4789	7119	8942
quick_sort	0.3	0.6	1.9	4	5.1	5.9	8.9	14.4	19.2	23.8	31	38	40.4
bucket_sort	0.1	0.2	1.3	2	3.3	3.8	12.3	26.9	44.7	65.9	92.9	128.9	155.8
heap_sort	0.3	1.2	2.7	4.5	6.2	8.3	18.4	29.2	40.7	51.9	66.6	81.4	89
intro_sort	0.3	1	1.4	2.3	3.3	4.3	9.6	15.4	19.9	25.9	31.5	36.9	42.2

FIGURE 9. AVERAGE CASE – SORTING A RANDOMLY SHUFFLES ARRAY (LEFT – WITH BUBBLE_SORT, RIGHT – WITHOUT BUBBLE_SORT)



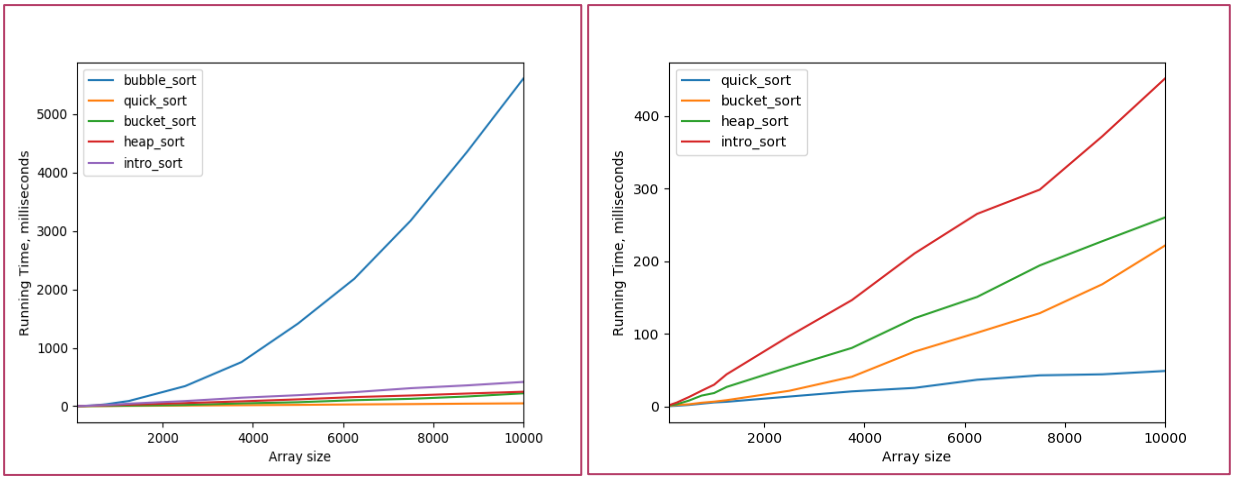
Finally in the so-called WORST case scenario, when the algorithms are sorting the presorted in a reverse order list, BubbleSort again comes up at the top with the worst (highest) time it takes to sort through the array (Figure 10, on the left). Figure 10 (on the right) also reveals that if the rest of the algorithms considered separately, the best performing of the remaining four algorithms is QuickSort, the worst is IntroSort, while BucketSort and HeapSort fall in the middle between the former two.

TABLE 3. WORST CASE –SORTING THROUGH PRESORTED ARRAY IN DESCENDING ORDER.

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
bubble_sort	0.5	4	13.9	32.4	60.9	88.5	343.9	755.9	1414	2182	3178	4355	5611
quick_sort	0.5	1.7	2.2	3	4.5	5.1	10.7	16.8	22.4	29.9	35.6	42.8	48.5
bucket_sort	0.7	1.3	2.8	4.4	6.3	8.6	21.3	48	67.7	102.8	126	165.9	220.4
heap_sort	1.1	3.2	9.2	14.9	19.8	25.1	53.1	83.1	116.8	155.8	182.9	216.4	246.9
intro_sort	1.6	5.2	12.4	20.6	30.6	43	87.6	145	188	240.1	307.8	355.9	414.8

It is a surprising outcome as QuickSort is expected to have an O(n2) in the worst case. However, in literature the worst performance for the QuickSort is noted as when pivot is badly chosen – either always the 1st element or the last.

FIGURE 10. WORST CASE –SORTING THROUGH PRESORTED ARRAY IN DESCENDING ORDER (LEFT – WITH BUBBLE_SORT, RIGHT – WITHOUT BUBBLE_SORT).



In the nutshell, performance of a sorting algorithms is not very easy to benchmark as different algorithms perform different on different sets of data. So the results of this Benchmarking exercise is can only be extrapolated on the similar data. Initially my hypothesis was that IntroSort would perform better than the other algorithms chosen in at least worst case scenario. However, QuickSort is a relative "winner" based on the results reported in this project.

Having said that, it is important to note that the QuickSort and HeapSort are unstable. If you a stable sort is needed neither of these or IntroSort would be suitable. If the data are almost sorted, the expert opinion is that, InsertionSort or BubbleSort will work much better than IntroSort. InsertSort has better memory access pattern and cache locality compared to IntroSort. If the data to be sorted consists of integers or floats, then use CountingSort or RadixSort would be used, that have better complexity.

The limitations of this study have been mentioned above. Further testing would be required in different types of data to explore the performance of the selected algorithms on variety of data. Comparison with a wider variety of algorithms would provide better insights into the topic.

4. REFERENCES

GEEKSFORGEEKS.ORG. 2020a. *Bubble Sort* [Online]. Available: https://www.geeksforgeeks.org/bubble-sort/ [Accessed].

GEEKSFORGEEKS.ORG. 2020b. *Bucket Sort* [Online]. Available: https://www.geeksforgeeks.org/bucket-sort-2/ [Accessed].

GEEKSFORGEEKS.ORG. 2020c. *HeapSort* [Online]. Available: https://www.geeksforgeeks.org/heap-sort/ [Accessed].

GEEKSFORGEEKS.ORG. 2020d. *IntroSort or Introspective sort.* [Online]. Available: https://www.geeksforgeeks.org/introsort-or-introspective-sort/ [Accessed].

GEEKSFORGEEKS.ORG. 2020e. *QuickSort* [Online]. Available: https://www.geeksforgeeks.org/quick-sort/ [Accessed].

HEINEMAN, G. T., POLLICE, G. & SELKOW, S. 2016. *Algorithms in a nutshell: A practical guide*, " O'Reilly Media, Inc.".

HUDGENS, J. 2016. *Why are Sorting Algorithms Important?* [Online]. Available: https://www.crondose.com/2016/07/sorting-algorithms-important/ [Accessed].

MANNION, P. 2017a. *Sorting Algorithms. Part1.* [Online]. Available: https://learnonline.gmit.ie/pluginfile.php/191466/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf [Accessed].

MANNION, P. 2017b. *Sorting Algorithms. Part 2.* [Online]. Available: https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf [Accessed].

MANNION, P. 2017c. *Vodeo: Sorting Algorithms: Part2.* [Online]. Available: https://galwaymayoinstitute.sharepoint.com/portals/hub/_layouts/15/PointPublishing.aspx?app=video&p=p&chid=7dca8d2a-5df7-4414-add4-80d7ef6c1473&vid=88aa6c0b-26c4-4be1-b132-e1ca5225e53c [Accessed].

MIYAKI, K. 2020. Basic Algorithms — Heapsort. Sorting an array with “Heap” data structure.

MUELLER, J. P. & MASSARON, L. 2017. *Algorithms for Dummies*, John Wiley & Sons.

MUSSER, D. R. 1997. Introspective sorting and selection algorithms. *Software: Practice and Experience,* 27**,** 983-993.

SEHGAL, K. 2018. An Introduction to Bucket Sort.

SKIENA, S. S. 2009. *The Algorithm Design Manual*, Springer London.

WIKIPEDIA.ORG. 2020a. *Bucket Sort* [Online]. Available: https://en.wikipedia.org/wiki/Bucket_sort [Accessed].

WIKIPEDIA.ORG. 2020b. *Heapsort* [Online]. Available: https://en.wikipedia.org/wiki/Heapsort [Accessed].

WIKIPEDIA.ORG 2020c. Introsort.

WIKIPEDIA.ORG. 2020d. *Sorting algorithm* [Online]. Available: https://en.wikipedia.org/wiki/Sorting_algorithm [Accessed].