IBU International Journal of Technical and Natural Sciences 2 (2021) 39-52

RESULTS OF COMPARISON BETWEEN DIFFERENT SORTING ALGORITHMS

Assoc. Prof. Dr. Festim Halili[1], Edlira Dika[2]

[1,2]Computer Engineering, Faculty of Engineering, International Balkan University, Macedonia

Abstract. *Analysis of algorithms is an issue that has always stimulate enormous curiosity. There are logical techniques for estimating the complexity of an algorithm. Generally, a more convenient solution is to estimate the run time analysis of the algorithm. The present piece of investigation documents the comparative analysis of five different sorting algorithms of data structures Bubble Sort, Heap Sort, Straight Insertion Sort, Shell Insertion Sort, and Quick Sort. The running time of these algorithms is calculated with the C++ language. These sorting algorithms are also compared on the basis of various parameters like complexity, method, memory, etc.*

Keywords: *sorting algorithms; sort; algorithm.*

1. INTRODUCTION

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.
In other words, sorting algorithms are ways to organize an array of items from smallest to largest. These algorithms can be used to organize messy data and make it easier to use.
Sorting properties:

A. Adaptive
A sort is adaptive if it runs faster on a partially sorted array.

B. Stable
A sort is stable if it preserves the relative order of equal keys in the database.

C. In Situ
An in situ ("in place") sort moves the items within the array itself and, thus, requires only a small O(1) amount of extra storage.

D. Online
An online sort can process its data piece-by-piece in serial fashion without having the entire array available from the beginning of the algorithm.

2. FIVE SORTING ALGORITHMS

In this paper, is discussed for five sorting algorithms:

1. Straight Insertion Sort
2. Shell Insertion Sort
3. Heap Sort
4. Quick Sort
5. Bubble Sort

2.1 Straight Insertion Sort

Straight insertion sort is one of the most basic sorting algorithms that essentially inserts an element into the right position of an already sorted list. It is usually added at the end of a new array and moves down until it finds an element smaller than itself (the desired position). The process repeats for all the elements in the unsorted array.
Consider the array {3,1,2,5,4}, we begin at 3, and since there are no other elements in the sorted array, the sorted array becomes just {3}. Afterward, we insert 1 which is smaller than 3, so it would move in front of 3 making the array {1,3}. This same process is repeated down the line until we get the array {1,2,3,4,5}.
The advantages of this process are that it is straightforward and easy to implement. Also, it is relatively quick when there are small amounts of elements to sort. It can also turn into binary insertion which is when you compare over longer distances and narrow it down to the right spot instead of comparing against every single element before the right place. However, a straight insertion sort is usually slow whenever the list becomes large.
Main Characteristics: Insertion sort family
Straightforward and simple
Worst case = O(n^2)

Table 1. Straight Insertion Sort Complexity

Time Complexity	
Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)
Stability	Yes

Time Complexities

- **Worst Case Complexity: $O(n^2)$**

Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.
Each element has to be compared with each of the other elements so, for every nth element, **(n-1)** number of comparisons are made.
Thus, the total number of comparisons = $\mathbf{n*(n-1) \sim n^2}$
To insert the last element, we need at most n-1n−1 comparisons and at most **n-1** swaps. To insert the second to last element, we need at most **n-2** comparisons and at most **n-2** swaps, and so on. The number of operations needed to perform insertion sort is therefore: $\mathbf{2\times(1+2+\cdots+n-2+n-1)}$. To calculate the recurrence relation for this algorithm, we use the following summation:

$$\sum_{q=1}^{p} q = p(p+1)/2$$

It follows that,

$$2(n-1)(n-1+1)/2 = n(n-1)$$

- **Best Case Complexity: O(n)**

When the array is already sorted, the outer loops run for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

- **Average Case Complexity: $O(n^2)$**

It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Space Complexity
Space complexity is **O(1)** because an extra variable **key** is used.

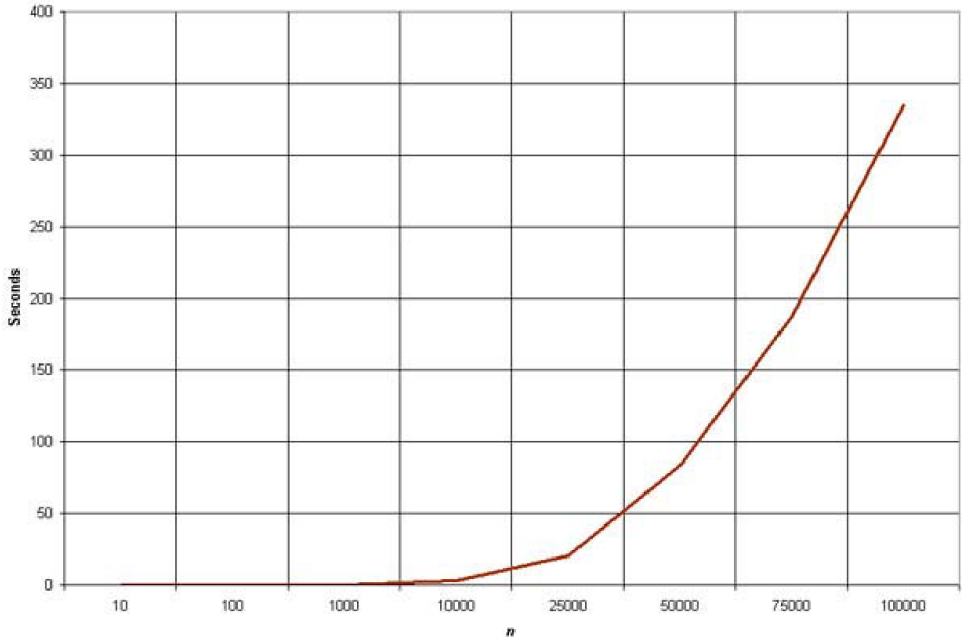

Figure 1. Insertion Sort Efficiency

The graph demonstrates the n2 complexity of the insertion sort.
The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

```
void insertionSort()
{
int temp;
for(long i = 1; i < length; i++)
{
temp = list[i];
long j;
for(j = i-1; j >= 0 && list[j] > temp; j--)
{
list[j+1] = list[j];
}
list[j+1] = temp;
}}
```

Code one

2.2 Shell Insertion Sort

Shell sort is an insertion sort that first partially sorts its data and then finishes the sort by running an insertion sort algorithm on the entire array. It generally starts by choosing small subsets of the array and sorting those arrays. Afterward, it repeats the same process with larger subsets until it reaches a point where the subset is the array, and the entire thing becomes sorted. The advantage of doing this is that having the array almost entirely sorted helps the final insertion sort achieve or be close to its most efficient scenario.

Furthermore, increasing the size of the subsets is achieved through a decreasing increment term. The increment term essentially chooses every kth element to put into the subset. It starts large, leading to smaller (more spread out) groups, and it becomes smaller until it becomes 1 (all of the array).
The main advantage of this sorting algorithm is that it is more efficient than a regular insertion sort. Also, there are a variety of different algorithms that seek to optimize shell sort by changing the way the increment decreases since the only restriction is that the last term in the sequence of increments is 1. (The most popular is usually Knuth's method which uses the formula h=((3^k)-1)/2 giving us a sequence of intervals of 1 (k=1),4 (k=2),13 (k=3), and so on. On the other hand, shell sort is not as efficient as other sorting algorithms such as quicksort and merge sort.)
Main Characteristics: Sorting by insertion
Can optimize algorithm by changing increments
Using Knuth's method, the worst case is O(n^(3/2))

Table 2. Shell Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	$O(n^2)$
Average	O(nlog n)
Space Complexity	O(1)
Stability	No

Time Complexities

- **Worst Case Complexity: $O(n^2)$**

It is always less than or equal to **$O(n^2)$**
In a very worst-case scenario (which doesn't exist), each sort would be quadratic time.
comparisons ≤ n2, for 1 sort with elements 1-apart (last step)
+ 3 * (n/3)2, for 3 sorts with elements 3-apart (next-to-last step)
+ 7 * (n/7)2, for 7 sorts with elements 7-apart
+ 15 * (n/15)2, for 15 sorts with elements 15-apart
+ ...
We can see that the number of comparisons is bounded by: n2 * (1 + 1/3 + 1/7 + 1/15 + 1/31 + ...) < n2 * (1 + 1/2 + 1/4 + 1/8 + 1/16 + ...) = n2 * 2
The last step uses the sum of the geometric series.

- **Best Case Complexity: O(n*log n)**

When the array is already sorted, the total number of comparisons for each interval or increment is equal to size of the array.
The best case, like insertion sort, is when the array is already sorted. Then the number of comparisons for each of the increment-based insertion sorts is the length of the array.
Therefore we can determine: comparisons =
n, for 1 sort with elements 1-apart (last step)
+ 3 * n/3, for 3 sorts with elements 3-apart (next-to-last step)
+ 7 * n/7, for 7 sorts with elements 7-apart
+ 15 * n/15, for 15 sorts with elements 15-apart
+ ...
Each term is n. The question is how many terms are there? The number of terms is the value k such that
2k - l < n
So k < log(n+1), meaning that the sorting time in the best case is less than n * log(n+1) = O(n*log(n)).

- **Average Case Complexity: O(n*log n)**

It is around **$O(n^{1.25})$.**
The complexity depends on the interval chosen. The above complexities differ for different increment sequences chosen. Best increment sequence is unknown.

Space Complexity
Space complexity is **O(1).**

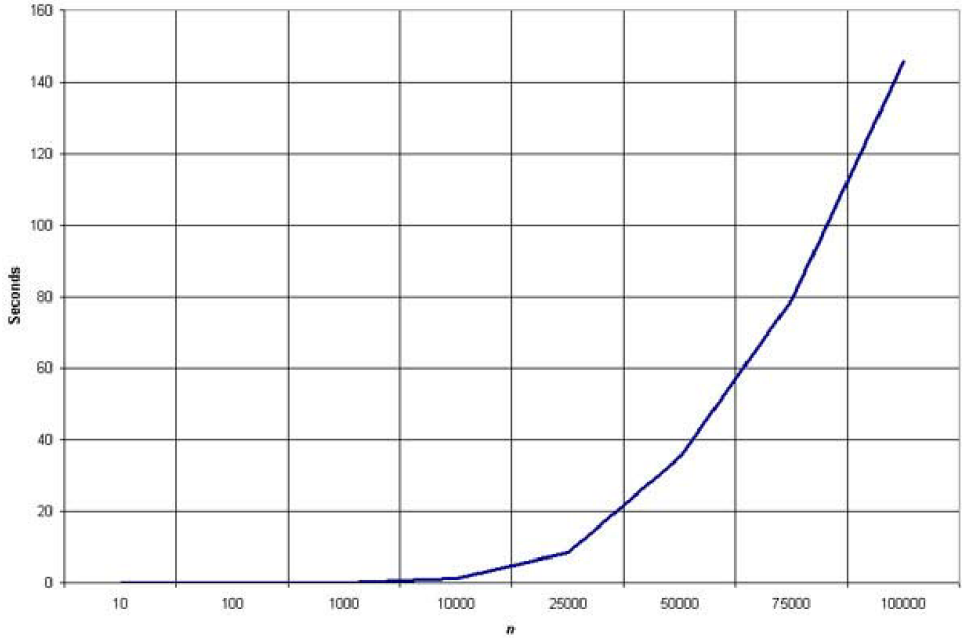

Figure 2. Shell Sort Efficiency

The shell sort is by far the fastest of the N2 class of sorting algorithms. It's more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor. The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed is hyper-critical. It's also an excellent choice for repetitive sorting of smaller lists.

```
void insertionSort()
{
    int temp;
    for(long i = 1; i < length; i++)
    {
        temp = list[i];
        long j;
        for(j = i-1; j >= 0 && list[j] > temp; j--)
        {
            list[j+1] = list[j];
        }
        list[j+1] = temp;
    }
}
```

Code two

2.3 Heap Sort

Heapsort is a sorting algorithm based on the structure of a heap. The heap is a specialized data structure found in a tree or a vector. In the first stage of the algorithm, a tree is created with the values to be sorted, starting from the left, we create the root node, with the first value. Now we create a left child node and insert the next value, at this moment we evaluate if the value set to the child node is bigger than the value at the root node, if yes, we change the values. We do this to all the trees. The initial idea is that the parent nodes always have bigger values than the child nodes.
At the end of the first step, we create a vector starting with the root value and walking from left to right filling the vector.
Now we start to compare parent and child nodes values looking for the biggest value between them, and when we find it, we change places reordering the values. In the first step, we compare the root node with the last leaf in the tree. If the root node is bigger, then we change the values and continue to repeat the process until the last leaf is the larger value. When there are no more values to rearrange, we add the last leaf to the vector and restart the process. We can see this in the image below.
Main characteristics: From the family of sorting by selection
Comparisons in the worst-case = O(n log n)
Not stable

Table 3. Heap Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	O(nlog n)
Average	O(nlog n)
Space Complexity	O(1)
Stability	No

Time Complexities
Heap Sort has **O(n*log n)** time complexities for all the cases (best case, average case, and worst case).
Let us understand the reason why. The height of a complete binary tree containing n elements is **log n**.
To fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it.In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of **log(n)** comparisons and swaps.

The run-time analysis of maxifying the heap depends on the number of "trickle-downs" per node being swapped. For example, if you have n = 6 total elements and are at level i = 0, you are at the level of the leaf nodes. By looking at the heap, you can see there are 3 nodes at this level i = 0. The expression used to prove this is **n/2^i+1**.
If we are at level i, we have to go a max of i levels down. This means that if we are at level 0, we go down 0 levels, which makes sense because this is the bottom-most level where all the leaf nodes are.
Similarly, at level i = 1, we "trickle down" a maximum of 1 level. At i = 2, we "trickle" a max 2 levels down. And so forth for any level i.

Space Complexity
Space complexity is **O(1).**

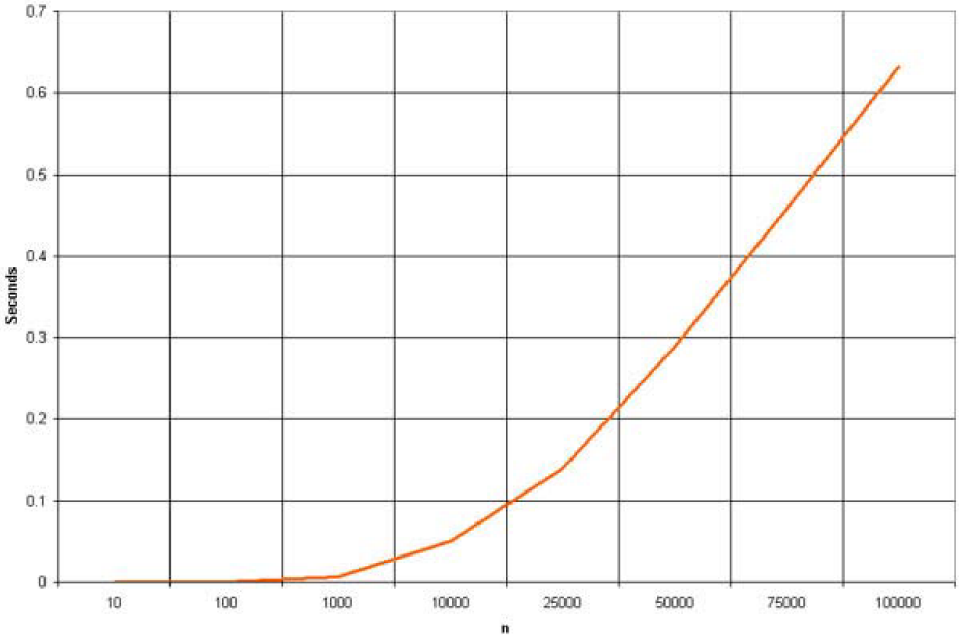

Figure 3. Heap Sort Efficiency

As mentioned above, the heap sort is slower than the merge and quick sorts but doesn't use multiple arrays or massive recursion like they do. This makes it a good choice for really large sets, but most modern computers have enough memory and processing power to handle the faster sorts unless over a million items are being sorted.
The "million item rule" is just a rule of thumb for common applications - high-end servers and workstations can probably safely handle sorting tens of millions of items with the quick or merge sorts. But if you're working on a common user-level application, there's always going to be some yahoo who tries to run it on junk machine older than the programmer who wrote it, so better safe than sorry.

```
void heapSort(long length)
{

    // Build heap (rearrange array)
    for (int i = length / 2 - 1; i >= 0; i--)
        heapify(list, length, i);

    // One by one extract an element from heap
    for (int i = length - 1; i > 0; i--) {
        // Move current root to end
        swap(list[0], list[i]);

        // call max heapify on the reduced heap
        heapify(list, i, 0);
    }
}
```

Code three

2.4 Quick Sort

Quicksort is one of the most efficient sorting algorithms, and this makes it one of the most used as well. The first thing to do is to select a pivot number, this number will separate the data, on its left are the numbers smaller than it and the greater numbers on the right. With this, we got the whole sequence partitioned. After the data is partitioned, we can assure that the partitions are oriented, we know that we have bigger values on the right and smaller values on the left. The quicksort uses this divide and conquers algorithm with recursion. So, now that we have the data divided we use recursion to call the same method and pass the left half of the data, and after the right half to keep separating and ordinating the data. At the end of the execution, we will have the data all sorted.

Main characteristics: From the family of Exchange Sort Algorithms
Divide and conquer paradigm
Worst-case complexity $O(n^2)$

Table 4. Quick Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	$O(n^2)$
Average	O(nlog n)
Space Complexity	O(log n)
Stability	No

Time Complexities

- **Worst Case Complexity [Big-O]: $O(n^2)$**

It occurs when the pivot element picked is either the greatest or the smallest element. This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains **n-1** elements. Thus, quicksort is called only on this sub-array. However, the quicksort algorithm has better performance for scattered pivots.

When quicksort always has the most unbalanced partitions possible, then the original call takes cncnc, n time for some constant ccc, the recursive call on n-1 elements takes c(n-1) time, the recursive call on n-2 elements takes c(n-2) time, and so on.

When we total up the partitioning times for each level, we get

$cn+c(n-1)+c(n-2)+\cdots+2c = c(n+(n-1)+(n-2)+\cdots+2) = c((n+1)(n/2)-1)$.

- **Best Case Complexity [Big-omega]: O(n*log n)**

It occurs when the pivot element is always the middle element or near to the middle element.

The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2 elements. The latter case occurs if the subarray has an even number nnn of elements and one partition has n/2 elements with the other having n/2-1. In either of these cases, each partition has at most n/2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times.

- **Average Case Complexity: O(n*log n)**

It occurs when the above conditions do not occur.

Space Complexity

Space complexity is **O(log n).**

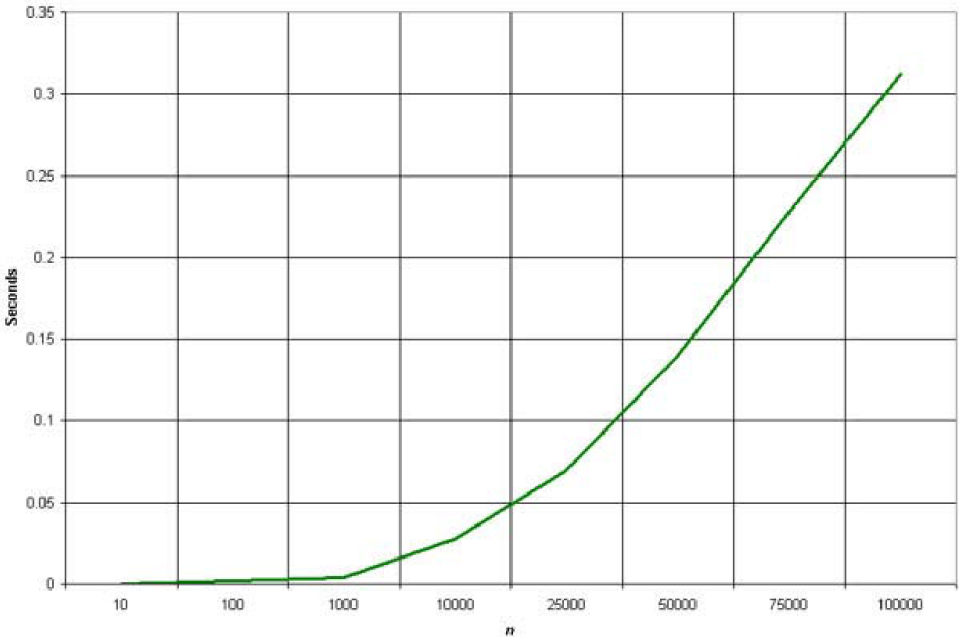

Figure 4. Quick Sort Efficiency

The quick sort is by far the fastest of the common sorting algorithms. It's possible to write a special-purpose sorting algorithm that can beat the quick sort for some data sets, but for general case sorting there isn't anything faster.

As soon as students figure this out, their immediate impulse is to use the quick sort for everything - after all, faster is better, right? It's important to resist this urge - the quick sort isn't always the best choice. As mentioned earlier, it's massively recursive (which means that for very large sorts, you can run the system out of stack space pretty easily). It's also a complex algorithm - a little too complex to make it practical for a one-time sort of 25 items, for example.

With that said, in most cases the quick sort is the best choice if speed is important (and it almost always is). Use it for repetitive sorting, sorting of medium to large lists, and as a default choice when you're not really sure which sorting algorithm to use. Ironically, the quick sort has horrible efficiency when operating on lists that are mostly sorted in either forward or reverse order - avoid it in those situations.

```
void quickSort(long left, long right)
{
    if (left < right)
    {
        long pivot = partition(left, right);
        quickSort(left, pivot-1);
        quickSort(pivot+1, right);
    }
}
```

Code four

2.5 Bubble Sort

Bubble sort compares adjacent elements of an array and organizes those elements. Its name comes from the fact that large numbers tend to “float” (bubble) to the top. It loops through an array and sees if the number at one position is greater than the number in the following position which would result in the number moving up. This cycle repeats until the algorithm has gone through the array without having to change the order. This method is advantageous

because it is simple and works very well for mostly sorted lists. As a result, programmers can quickly and easily implement this sorting algorithm. However, the tradeoff is that this is one of the slower sorting algorithms.

Main Characteristics: Exchange sorting
Easy to implement
Worst Case = O(n^2)

Table 5. Bubble Sort Complexity

Time Complexity	
Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)
Stability	Yes

Time Complexities

- **Worst Case Complexity: $O(n^2)$**

This is the case when the array is reversely sort i.e. in descending order but we require ascending order or ascending order when descending order is needed.
The number of swaps of two elements is equal to the number of comparisons in this case as every element is out of place.
$T(N) = C(N) = S(N) = \frac{N*(N-1)}{2}$, from equation 2 and 4
Therefore, in the worst case:
Number of Comparisons: O(N^2) time
Number of swaps: O(N^2) time

- **Best Case Complexity: O(n)**

This case occurs when the given array is already sorted.
For the algorithm to realise this, only one walk through of the array is required during which no swaps occur (lines 9-13) and the swapped variable (false) indicates that the array is already sorted.
$T(N) = C(N) = N$
$S(N) = 0$
Therefore, in the best case:
Number of Comparisons: N = O(N) time
Number of swaps: 0 = O(1) time

- **Average Case Complexity: $O(n^2)$**

The number of comparisons is constant in Bubble Sort so in average case, there is O(N^2) comparisons. This is because irrespective of the arrangement of elements, the number of comparisons C(N) is same.
For the number of swaps, consider the following points:
If an element is in index I1 but it should be in index I2, then it will take a minimum of I2-I1 swaps to bring the element to the correct position.
An element E will be at a distance of I3 from its position in sorted array. Maximum value of I3 will be N-1 for the edge elements and it will be N/2 for the elements at the middle.
The sum of maximum difference in position across all elements will be:
(N-1) + (N-3) + (N-5) ... + 0 + ... + (N-3) + (N-1)
= N x N - 2 x (1 + 3 + 5 + ... + N/2)
= N^2 - 2 x N^2 / 4
= N^2 - N^2 / 2
= N^2 / 2

Space Complexity

- Space complexity is **O(1)** because an extra variable is used for swapping.

- In the **optimized bubble sort algorithm**, two extra variables are used. Hence, the space complexity will be **O(2)**.

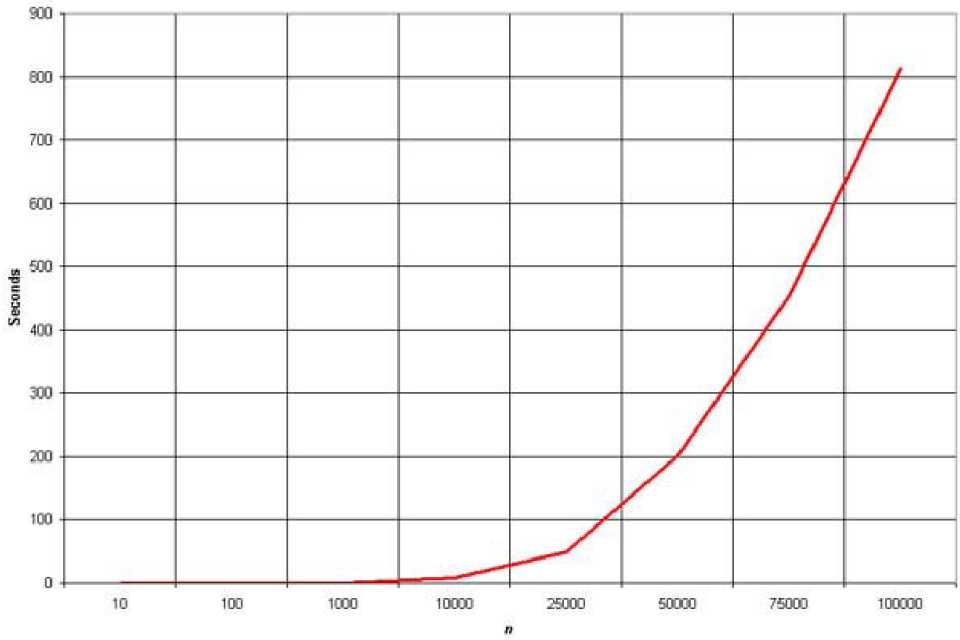

Figure 5. Bubble Sort Efficiency

The graph clearly shows the n2 nature of the bubble sort.
A fair number of algorithm purists (which means they've probably never written software for a living) claim that the bubble sort should never be used for any reason. Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items.

```
void bubbleSort()
{
    int temp;
    for(long i = 0; i < length; i++)
    {
        for(long j = 0; j < length-i-1; j++)
        {
            if (list[j] > list[j+1])
            {
                temp        = list[j];
                list[j]     = list[j+1];
                list[j+1]   = temp;
            }
        }
    }
}
```

Code five

3. RESULTS SORT COMPARISON

After the development of the algorithms, it is good for us to test how fast they can be. In this part we developed a simple program using the code above to generate a basic benchmark, just to see how much time they can use to sort a list of integers. An important observation about the code is that Shell sort and Heap Sort algorithms performed well despite the length of the lists, on the other side we found that Insertion sort and Bubble sort algorithms were far the worse, increasing largely the computing time.

ALGORITHM	BEST CASE	AVERAGE CASE	WORST CASE
Bubble Sort	Ω(n)	Ω(n^2)	Ω(n^2)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)

Figure 6. Results of comparison

In this section, we are going to conduct three sets of tests. The first will have 100 random numbers, the second will have 1000 and the third will have 10,000.

Table 6. Results of comparison

Sorting Algorithm	Test 1 (100)	Test 2 (1000)	Test 3 (10000)
Bubble Sort	Min: 0.01008 seconds Max: 0.0206 seconds	Min: 1.0242 seconds Max: 1.0558 seconds	Min: 100.922 seconds Max: 102.475 seconds
Insertion Sort	Min: 0.00306 seconds Max: 0.00650 seconds	Min: 0.0369 seconds Max: 0.0562 seconds	Min: 100.422 seconds Max: 102.344 seconds
Selection Sort	Min: 0.00556 seconds Max: 0.00946 seconds	Min: 0.4740 seconds Max: 0.4842 seconds	Min: 40.831 seconds Max: 41.218 seconds
Quick Sort	Min: 0.00482 seconds Max: 0.01141 seconds	Min: 0.0370 seconds Max: 0.0383 seconds	Min: 0.401 seconds Max: 0.420 seconds
Merge Sort	Min: 0.00444 seconds Max: 0.00460 seconds	Min: 0.0561 seconds Max: 0.0578 seconds	Min: 0.707 seconds Max: 0.726 seconds
Heap Sort	Min: 0.00489 seconds Max: 0.00510 seconds	Min: 0.0704 seconds Max: 0.0747 seconds	Min: 0.928 seconds Max: 0.949 seconds

The O(n2) Algorithms (Bubble and Insertion Sort) reacted very poorly as the number of tests went up to 10,000. At 10,000 numbers the other Algorithms were on average, over 100x times faster.

On the test cases with just 100 numbers, the O(n2) Algorithms were faster than the O(n.log(n)) Algorithms. With every 10x increase in the amount of numbers, the O(n2) Algorithms completion time increased by 100x.
Heapsort is fastest Algorithm with a space complexity of O(1).

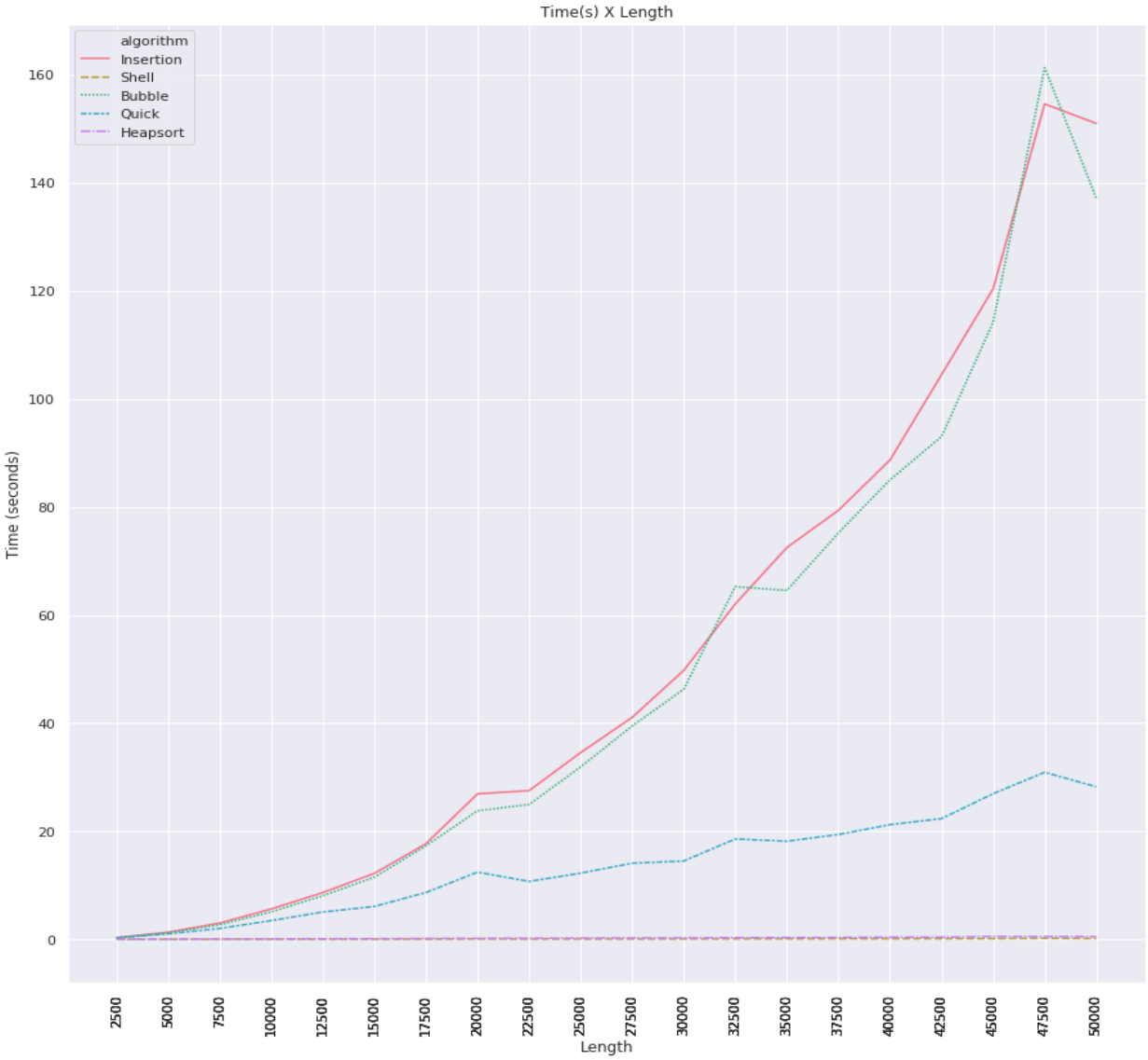

Figure 7. Results of comparison

4. CONCLUSION

In this study, we have studied various sorting algorithms and their comparison. There are advantages and disadvantages to all algorithms. To find the running time of all sorting algorithms, we used C++.

REFERENCES

1. Galler, L., Kimura, M. (2019). Sorting Algorithms, LAMFO [Online].Available at https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/ (Accessed 2 July 2021).
2. Kansas State Polytechnic (2015). Sorting Properties [Online], Available at http://faculty.salina.k-state.edu/tmertz/Java/324sorting/sortingproperties.pdf (Accessed 3 July 2021)
3. Wikipedia (2015). Sorting Algorithm [Online], Available at https://en.wikipedia.org/wiki/Sorting_algorithm (Accessed 1 July 2021)
4. Geeks for Geeks (2018). Sorting Algorithms [Online]. Available at https://www.geeksforgeeks.org/sorting-algorithms/ (Accessed 2 July 2021).
5. Moore, K., Pilling, G., Khim, J., Kappel, T., Takvam, K. (2021). Sorting Algorithms, Brilliant [Online]. Available at https://brilliant.org/wiki/sorting-algorithms/ (Accessed 4 July 2021).
6. Shell Sort Algorithm (2018). Programiz [Online]. Available at https://www.programiz.com/dsa/shell-sort (Accessed 4 July 2021)
7. Chauhan, Y., Duggal, A., (2020). Different Sorting Algorithms comparison based upon the Time Complexity. ResearchGate [Online]. Available at https://www.researchgate.net/publication/344280789_Different_Sorting_Algorithms_comparison_based_upon_the_Time_Complexity (Accessed 5 July 2021).
8. Software Testing Help (2021). Introduction To Sorting Techniques In C++ [Online]. Available at https://www.softwaretestinghelp.com/sorting-techniques-in-cpp/ (Accessed 5 July 2021).
9. Allain, A. (2019). Sorting Algorithm Comparison. CProgramming.com [Online]. Available at https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html (Accessed 6 July 2021).
10. SanFoundry (2021). C Program to Implement Shell Sort [Online]. Available at https://www.sanfoundry.com/c-program-implement-shell-sort/ (Accessed 7 July 2021).
11. Goodrich, T. M. (2018). Insertion Sort and Shell Sort [ebook] Irvine: University of California. Available at https://www.ics.uci.edu/~goodrich/teach/cs260P/notes/Shellsort.pdf (Accessed 7 July 2021).
12. Phongsai, J. (2009). Research Paper on Sorting Algorithms. Computer Science and Engineering, IIT Kanpur [Online]. Available at https://www.cse.iitk.ac.in/users/cs300/2014/home/~rahume/cs300A/techpaper-review/5A.pdf (Accessed 8 July 2021).
13. Programmer Sought (2021). Inserting Sort, Shell Sort, Heap Sort and Quick Sort [Online]. Available at https://www.programmersought.com/article/18836491585/ (Accessed 9 July 2021)