Balancing Performance Trade-offs in Modern Sorting Methodologies

Muyang Li

Shenzhen College of International Education, Guangzhou 518043, China.

Abstract. The study of sorting algorithms has always been a key topic. This paper thoroughly explains and investigates the time complexity of six classic sorting algorithms through theoretical analysis and experimental comparison. We implemented insertion sort, selection sort, bubble sort, shell sort, quicksort, and heapsort. By controlling data scale and distribution, we systematically tested the performance of different algorithms under various scenarios. The results show that there are significant efficiency differences between algorithms on small-scale data, and the advantages of quicksort and heapsort become more obvious as data size increases. Through extensive comparative experiments, this paper identifies the application scenarios of each algorithm, providing a theoretical basis for algorithm design and selection.

Keywords: Algorithm analysis; Sorting algorithm; Efficiency.

1. Related Work

There have been many papers analyzing sorting algorithms, but they often focus on just theoretical complexity or lack extensive experimental results. Prajapati (2017)[1] provides a theoretical analysis of time complexity for 8 sorts. However, there is no experimental data to verify the theory. Gill et al. (2018)[2] implements 6 algorithms experimentally but only tests on small datasets of 20 elements. In comparison, my experiments scale up to 100,000 elements to better highlight performance differences. Bhalchandra et al. (2009)[3] deeply analyzes complexity but does not implement any experiments. My work bridges theory and practice through both complexity analysis and extensive timed tests. Bei (2018)[4] experimentally tests 8 algorithms in C++ but does not analyze complexity theory. In contrast, my work combines both theoretical analysis and experimental validation for a comprehensive comparison.

Overall, this paper provides a more in-depth experimental analysis than Gill et al. by testing larger datasets. It also combines theory and experimentation better than works of Prajapati, Bhalchandra, and Bei. By including both complexity analysis and thorough experimental validation on varied data sizes, my comparative study provides a solid theoretical and practical basis for choosing the right sorting algorithm for a given application.

2. Introduction

Sorting is an important operation in computer programming. Its function is to rearrange an arbitrary sequence of data elements into an ordered sequence. It is an important foundation for computer programming, databases, operating systems, compilation principles, and artificial intelligence. The speed or efficiency of the algorithm is essential for today's exponentially growing data, choosing the right sorting algorithm therefore is critical for optimized performance. Simple sorts like insertion don't scale, while fast algorithms like quicksort have higher implementation complexity. The optimal choice depends on data volume, memory size, stability requirements and data characteristics. Selecting the best algorithm improves speed and responsiveness when sorting large datasets. Understanding sorting algorithms and their individual benefit allows the user to choose the most suitable algorithm for their project, enabling faster information process ability.

This paper presents a focused comparison of six classic sorting techniques: insertion sort, selection sort, bubble sort, shell sort, quicksort, and heapsort. Though hundreds of sorting algorithms have been devised, these represent the foundational methods every computer scientist should understand. I

hypothesize advanced algorithms like quicksort will demonstrate clear speed advantages over simpler sorting algorithms for larger datasets. But simple sorts like bubble sort or selection sort may excel for small lists of data.

This essay will begin with an overview of each algorithm's mechanics and expected efficiencies. Followed by the description of the experimental setup to test each algorithm with numerous datasets. Finally, performance results are presented, analyzed, and discussed. Through such an approach, I believe my findings can provide a solid basis for algorithm selection and design decisions in practical applications.

2.1 Direct Insertion Sort

2.1.1 Basic Principle

Suppose there are n records $\{R0, R1, \ldots, Rn\}$ to be sorted stored in order in an array. The direct insertion method divides the records into two intervals when inserting record Ri $(i = 1,2, \ldots, n-1)$: $[R0, Ri-1]$ and $[Ri+1, Rn-1]$. The first interval is the sorted subsection, and the second interval is the unsorted subsection. The key Ki is compared sequentially with $Ki-1, Ki-2, \ldots, K0$ to find the proper insertion position, then record Ri is inserted. The remaining $i-1$ elements are inserted into the ordered sequence one by one according to the size of the keys, maintaining the ordered sequence after each insertion. After $i-1$ iterations, the sequence becomes ordered. Each time, a record to be sorted is inserted into the proper position in the previously sorted subsection according to its key value, until all records are inserted.

2.1.2 Sorting Process

Now, let's take the list of number: [10 3 25 20 8] as an example to see how insertion sort works in step-by-step breakdown, shown by Table.2.1.2.

Table. 2.1.2

Initial data:	[**10 3 25 20 8**]	Movement
First iteration:	[3 10] [25 20 8]	first two data sorted
Second iteration:	[3 10 25] [20 8]	25 inserted into proper position
Third iteration:	[3 10 20 25] [8]	20 inserted into proper position
Fourth iteration:	[3 8 10 20 25]	8 inserted into proper position

2.1.3 Time Complexity & Stability Analysis

The direct insertion sort algorithm must conduct $n-1$ iterations. In the best case, the initial sequence is already ordered, so the total comparisons are $n-1$ and total element moves is 0 in the best case. In the worst case where all elements are in the opposite order, total comparisons is $(n-1)$ and total element moves is $2(n-1)$. Therefore, the best-case time complexity is $O(n)$. In the worst case (reverse ordered), the maximum comparisons are i per iteration. Therefore, the time complexity is $O(n^2)$. Furthermore, in insertion sort, elements are shifted to the right until the correct position is found to insert the element. Equal keys will remain in the same order, so insertion sort is stable.

2.2 Direct Selection Sort

2.2.1 Basic Principle

For the data elements to be sorted, select the smallest element from the n elements and swap it with the first element. Then in the remaining $n-1$ elements, find the smallest and swap it with the second element. Repeat this process until only one element remains. In the first iteration, find the minimal key record from the nth record and swap it with the nth record. In the second iteration, find

the minimal key record from the 2^{nd} to $(n-1)th$ records and swap it with the 2^{nd} record. In the i^{th} iteration, find the minimal key record from the $(i+1)^{th}$ to n^{th} records and swap it with the ith record. Repeat until all records are sorted.

2.2.2 Sorting Process

Now, let's take the list of number: [49 38 65 97 76 13 27 49] as an example to see how direct selection sort works in step-by-step breakdown, shown by Table.2.2.2.

Table. 2.2.2

Initial data:	[**49 38 65 97 76 13 27 49**]	Movement
First iteration:	13 [**49 38 65 97 76 27 49**]	13 extracted
Second iteration:	13 27 [**49 38 65 97 76 49**]	27 extracted & linked to end
Third iteration:	13 27 38 [**49 65 97 76 49**]	38 extracted & linked to end
Fourth iteration:	13 27 38 49 [65 97 76 49]	38 extracted & linked to end
Fifth iteration:	13 27 38 49 49 [65 97 76]	38 extracted & linked to end
Sixth iteration:	13 27 38 49 49 65 [97 76]	38 extracted & linked to end
Seventh iteration:	13 27 38 49 49 65 76 [97]	76 extracted, finished sorting

2.2.3 Time Complexity & Stability Analysis

The running time of this algorithm does not depend on the initial arrangement of the elements. No matter the initial arrangement, the algorithm must conduct $n-1$ iterations, comparing $n-i-1$ keys in each iteration. Therefore, the best, worst, and average case time complexities of simple selection sort are all $O(n^2)$. Furthermore, selection sort repeatedly finds the minimum element and swaps it to the front. Swapping destroys the original order of equal keys, so selection sort is unstable.

2.3 Bubble Sort

2.3.1 Basic Principle

In each bubble sort iteration, adjacent keys are compared sequentially and swapped if in reverse order. After one pass of bubbling, the largest key must move to the end. In the first pass on the sequence $(I[0]\sim I[n-1])$, adjacent elements are compared from front to back, swapped if the latter is smaller, for a total of n-1 comparisons. After the first pass, the maximum element is ensured to be swapped to $I[n-1]$. Thus, bubble sort increases the length of the ordered sequence and decreases the length of the unsorted sequence. After each pass, the unsorted sequence decreases by 1. The next pass only needs to process the subsequence $(I[0]\sim I[n-2])$. If no swap occurred during a pass, no further pass is needed as it shows all elements are in the right position.

2.3.2 Sorting Process

Now, let's take the list of number: [10 3 25 20 8] as an example to see how bubble sort works in step-by-step breakdown, shown by Table.2.3.2.

Table. 2.3.2

Initial data:	[10 3 25 20 8]	Movement
First iteration:	[3 10] 25 20 8	10 & 3 swapped
	3 [10 25] 20 8	10 & 25 not swapped
	3 10 [20 25] 8	25 & 20 swapped

	3 10 20 [8 25]	25 & 8 swapped
Second iteration:	[3 10] 20 8 25	3 & 10 not swapped
	3 [10 20] 8 25	10 & 20 not swapped
	3 10 [8 20] 25	20 & 8 swapped
Third iteration:	[3 10] 8 20 25	3 & 10 not swapped
	3 [8 10] 20 25	10 & 8 swapped
Fourth iteration:	3 [8 10] 20 25	**8 & 10 not swapped**

2.3.3 Time Complexity & Stability Analysis

In the best case, all initial data is already ordered. Only 1 pass is needed, with $n-1$ comparisons, so best case complexity is $O(n)$. It is worst when initial data is reverse ordered. $n-1$ passes are needed, with $(n-i)$ comparisons and swaps per pass. Therefore, worst case complexity is $O(n^2)$. Furthermore, in bubble sort, equal keys will never swap positions with each other. Larger elements simply bubble up past smaller elements. So the relative order never changes and bubble sort is stable.

2.4 Shell Sort

2.4.1 Basic Principle

Shell sort is also called diminishing increment sort. It compares elements at a specified interval (called gap) and sorts them. It continually reduces the gap and sorts until the gap is 1 and no more swapping occurs. First take an integer $d1 \leq n/2$, divide the records into $d1$ groups with records at increments of $d1$ in a group, sort within each group by using direct insertion sort. Then take $d2 = d1 - 2$ and repeat the grouping and sorting. Keep reducing the increment until $dt = 1$ all records are in one group, and sort is organized.

2.4.2 Sorting Process

Now, let's take the list of number: 9 10 3 25 7 20 8 6 as an example to see how shell sort works in step-by-step breakdown, shown by Table.2.4.2.

Table. 2.4.2

Initial data:	9 10 3 25 7 20 8 6	Movement
First iteration: (d1=4)	[9 7] [10 20] [3 8] [25 6] 7 10 3 6 9 20 8 25	9 & 7 swapped 25 & 6 swapped
Second iteration: (d2=2)	[7 3 9 8] [10 6 20 25] 3 6 7 10 8 20 9 25	1st group changed to 3 7 8 9 2nd group changed to 6 10 20 25
Third iteration: (d3=1)	[3 6 7 10 8 20 9 25]	**Using basic insertion sort to reach result**
	3 6 7 8 9 10 20 25	

2.4.3 Time Complexity & Stability Analysis

Shell sort evolve from direct insertion sort. Initially the elements are very disordered, so the large gap results in few elements to insertion sort, making it fast. When elements are basically ordered, the reduction in intervals allows elements to move to their correct positions faster, as elements can 'jump' over several positions in a single step when the intervals are large. As the interval reduces, the list becomes more sorted, making the process more efficient. So, Shell sort is better than $O(n^2)$. Furthermore, shell sort is unstable because elements can swap positions during the different passes based on gaps. Equal elements can move out of sequence.

2.5 Quicksort

2.5.1 Basic Principle

First select a pivot element, put all elements less than the pivot to its left, and greater than to its right. This partitioning is done recursively. On an arbitrary given sequence, after one partition based on a chosen element, the original sequence is divided into two subsequences $(R_{(p_0)}, R_{(p_1)}, \dots, R_{(p_{s-1})})$ and $(R_{(p_{s+1})}, R_{(p_{s+2})}, \dots, R_{(p_{n-1})})$. All keys in the first subsequence are less than or equal to the pivot key $R_{(p_s)}$, and those in the second greater than or equal to it. This pivot element $R_{(p_s)}$ splits the low end and high end subsequences. Obviously, quicksort can now recursively operate on these two subsequences $(R_{(p_0)}, R_{(p_1)}, \dots, R_{(p_{s-1})})$ and $(R_{(p_{s+1})}, R_{(p_{s+2})}, \dots, R_{(p_{n-1})})$ until the subsequences have length 0 or 1. In summary, each iteration places the first element of the sequence in the proper position, dividing the sequence in two, repeating recursively on the sub-sequences until they contain only 1 element.

2.5.2 Sorting Process

Now, let's take the list of number: 49 38 65 97 76 13 27 as an example to see how quicksort works in step-by-step breakdown, shown by Table.2.5.2.

Table. 2.5.2

Initial data:	49 38 65 97 76 13 27	Movement
After first partition: (pivot=49)	[27 38 65 97 76 13] 49	
After second partition:	[27 38] 49 [97 76 13 65]	
After third partition:	[27 38 13 97 76]49 [65]	
After fourth partition:	[27 38 13] 49 [97 76 65]	
Second iteration: (pivot=27 & 97)	[13] 27 [38] [55 76] 97	
Together:	13 27 38 49 55 76 97	**Link the two smaller subsequences together with 49**

2.5.3 Time Complexity & Stability Analysis

For n elements, quicksort requires around $n * \log(n)$ comparisons and $(n * logn)/6$ swaps. The performance of quicksort is highly sensitive to the choice of the pivot element. If the pivot happens to be the smallest or largest element consistently, the partitioning becomes unbalanced, resulting in a skewed divide of the array. This can degrade the performance, leading to a worst-case time complexity of $O(n^2)$, but average is $O(n * logn)$. Furthermore, quicksort partitions elements around a pivot. Elements equal to the pivot can end up on either side, so the original order is not preserved.

2.6 Heap Sort

2.6.1 Basic Principle

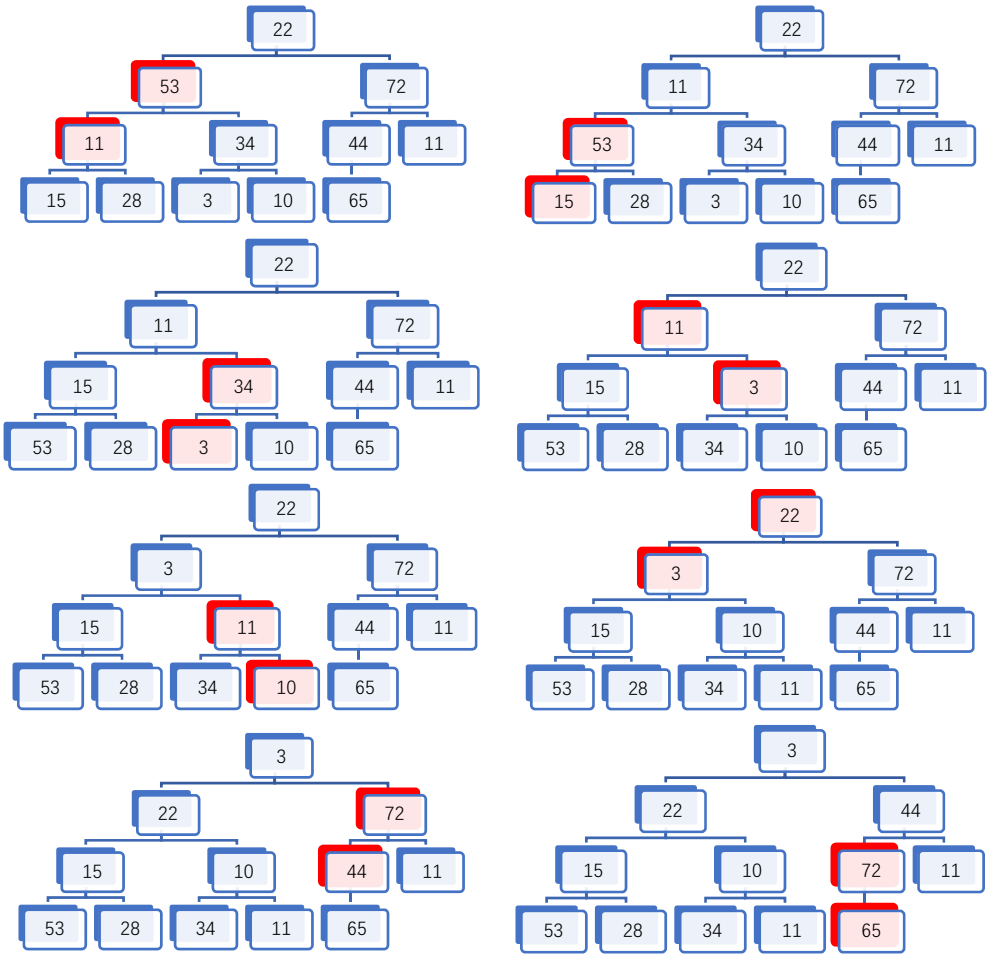
The idea of heap sort is simple. It repeatedly turn the array into a max heap, extract the root element and swap it with the last, shrinking the heap size by 1, then recreate the heap with the remaining elements. Repeat until only one element remains.

Given a sequence of keys $k1, k2, \ldots, kn$, it forms a heap if it satisfies:
(1) $k_i \leq k_{2i}$ and $k_i \leq k_{2i}+1$ or
(2) $k_i \geq k_{2i}$ and $k_i \geq k_{2i}+1$

If we view the sequence stored in array $R[1\ldots n]$ as a complete binary tree, then a heap is a complete binary tree that satisfies: each non-leaf node key is greater than (or less than) that of its children. A min heap has the root as minimum key, a max heap has the root as maximum key.

2.6.2 Sorting Process

Now, let's take the list of number: 22 53 72 11 34 44 11 15 28 3 10 65 as an example to see how quicksort works in step-by-step breakdown, shown by Figure.2.6.2.



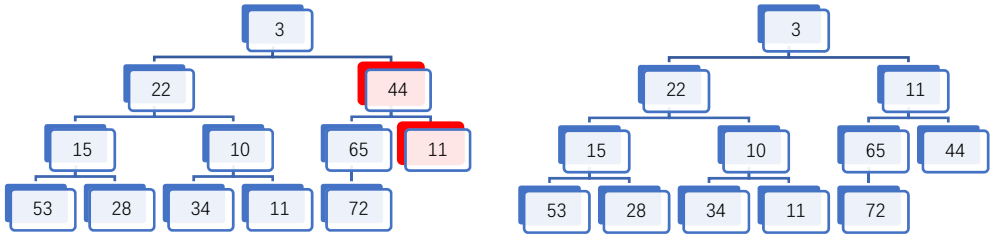


Figure. 2.6.2

Done swapping, output minimum value 3 and remove from heap. Repeat building heap for the remaining part of the heap (22 to 11) and extracting min/max until sorted.

2.6.3 Time Complexity & Stability Analysis

Heap sort consists of building the initial heap and reheapifying. Both are done by calling up the heapify function. Worst case time complexity is $O(n * logn)$. Average performance is close to worst case. The heap build has many comparisons, so heap sort is not suitable for small n. Furthermore, heapsort repeatedly extracts the max by swapping it with the last element. This destroys the original ordering of keys, including equal keys. Thus, heapsort is unstable.

3.1 Data Definition

3.1.1 Input data:

Since the running time of most sorting algorithms depends not only on the problem size but also the input instance, we generate random test data of fixed sizes of 1000. To ensure the reliability of the test and reduce the effect of a random error, we will repeat sorting for 3 times for each algorithm.

3.1.2 Output data:

The generated random numbers are sorted by direct insertion sort, direct selection sort, bubble sort, shell sort, quicksort, and heapsort. The number of key comparisons and moves will be recorded for three times and the mean value for the number of comparison and number of exchanges will be output.

3.2 Program Flowchart

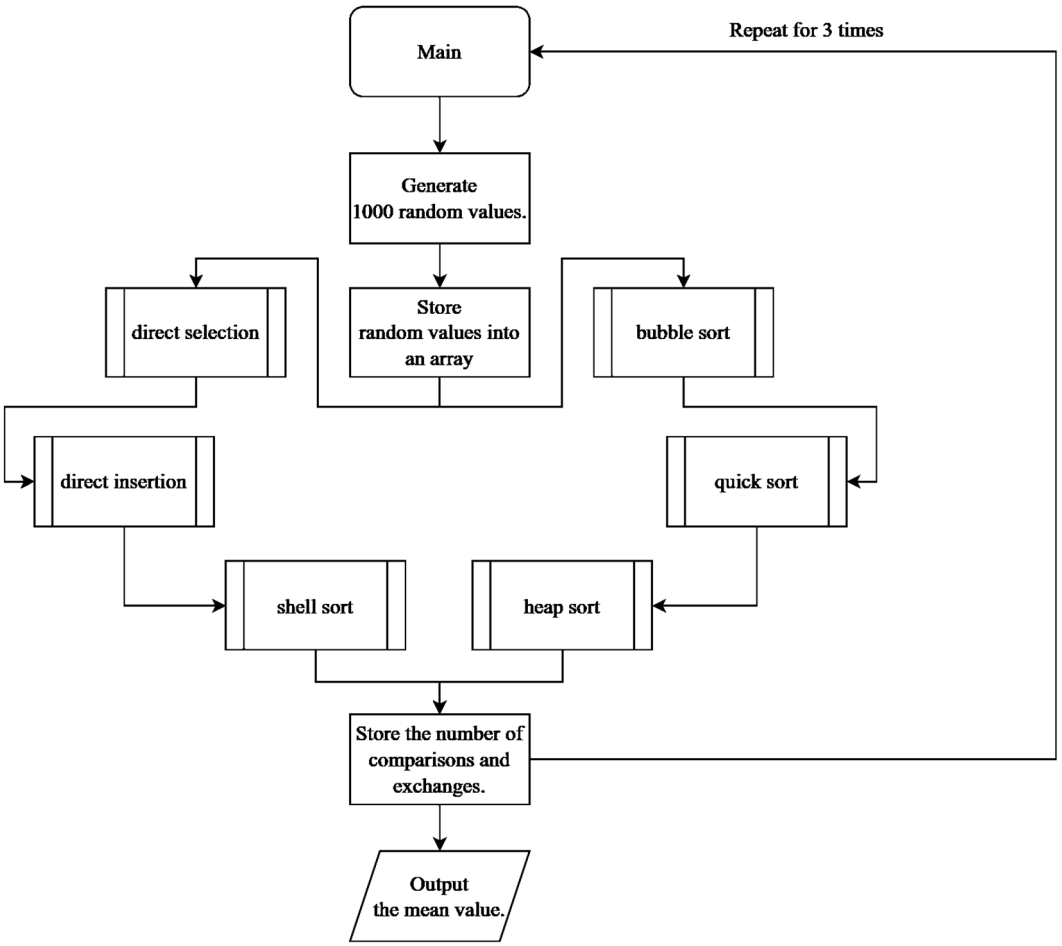


Figure. 3.2

4.1 Running and Testing

The program (see more in Appendix) functionality is to compare the time complexities of various sorting algorithms. A fixed length array consisting of 1000 data is tested, with random number generated data, without need for manual input. The program interface is shown in Table. 4.1-1:

Table. 4.1-1

Algorithm	Number of Comparison	Number of Exchange	Time/ms
Direct Insertion Sort	239396	243368	9
Direct Selection Sort	499000	5916	4
Bubble Sort	492388	718188	15
Shell Sort	74108	86052	2
Quick Sort	12496	9148	6
Heap Sort	21228	26164	2

From Figure 4.1, we can see that of these six algorithms, shell sort and heap sort took the least amount of time. Quick sort has the fewest comparisons and moves, making it one of the fastest sorting methods. Heap sort is close to quicksort. Direct selection sort has few swaps but more comparisons.

Bubble sort on the other hand has the largest number of comparisons and largest number of swaps making is the least efficient sorting algorithm.

Now, taking this experiment even further. We decided to increase the length of the test data to 100000 elements. Due to the number of exchanges and comparison will be too large under this scenario, we will only record the time taken for algorithm to sort the data (shown by Table. 4.1-2).

Table. 4.1-2

	Direct Insertion Sort	Direct Selection Sort	Bubble Sort	Shell Sort	Quick Sort	Heap Sort
Time/ms	606	4200	17141	22	19	23

Through this extreme test, we found that under such large data volumes, the differences between these six sorting algorithms widened further. Insertion sort, selection sort, and bubble sort became even slower, with bubble sort taking 17141 milliseconds to complete sorting. Meanwhile, shell sort, quicksort, and heapsort all finished in under 25 milliseconds. Quicksort, which was slower than heapsort and shell sort for 1000 elements, overtook them for 100000 elements, finishing 4 and 3 milliseconds faster respectively. Overall, insertion sort, selection sort, and bubble sort remain stable and simple but relatively slow, while shell sort, quicksort and heapsort are more efficient.

4.2 Result Analysis

Through the above steps and analysis, we can collate the data and summarize it in the following table (Table 4.2):

Table. 4.2

Algorithm	Average Time Complexity	Worst Time Complexity	Stability	Notes:
Direct Insertion Sort	$O(n^2)$	$O(n^2)$	Stable	Good performance when most are initially in order
Direct Selection Sort	$O(n^2)$	$O(n^2)$	Not Stable	Good performance when n is small
Bubble Sort	$O(n^2)$	$O(n^2)$	Stable	Good performance when n is small
Shell Sort	$O(n * logn)$	$O(n^2)$	Not Stable	Good performance when most are initially in order
Quick Sort	$O(n * logn)$	$O(n^2)$	Not Stable	Good performance when n is big
Heap Sort	$O(n * logn)$	$O(n * logn)$	Not Stable	Good performance when n is big

5.1 Conclusion

Simple Quadratic Sorts:

Bubble sort, insertion sort, and selection sort all exhibit average and worst time complexities of $O(n^2)$. Due to this quadratic nature, their performance diminishes significantly for larger datasets. Among these, bubble sort tends to be the least efficient, necessitating the maximum number of comparisons and swaps. Consequently, these algorithms are primarily beneficial for small-scale sorting tasks.

Gap-Based Improvement - Shell Sort:

Shell sort, which strategically employs gaps, offers an enhanced average time complexity of $O(n * logn)$. But its worst-case scenario still aligns with $O(n^2)$. Given its design, Shell sort is particularly effective when dealing with inputs that are already semi-ordered.

Advanced Logarithmic Sorts:

Quicksort and heapsort are clearly the fastest algorithms with average case complexity of $O(n * logn)$. Their superiority stems from a 'divide and conquer' approach applied recursively. While quicksort often outperforms heapsort due to its in-place sorting capability, its efficiency can be hampered if the pivot selection is consistently suboptimal. In contrast, heapsort offers a more stable O(n log n) performance across both average and worst cases, rendering these logarithmic sorts especially apt for extensive sorting challenges.

Stability Analysis:

Stability in sorting algorithms implies that the relative order of equal keys remains unchanged after sorting. This feature is crucial in scenarios where the initial order carries meaningful information. In our set of algorithms, insertion sort, bubble sort, and shell sort are inherently stable. On the other hand, the remaining algorithms — notably quicksort, heapsort, and selection sort — lack this stability. This instability arises because these algorithms might swap distant elements, thereby disrupting the original order.

Performance Recommendations:

For smaller datasets, specifically with fewer than 1,000 elements, simpler algorithms like insertion or selection sort are typically adequate due to their straightforward nature and lower overhead. However, as the dataset size extends beyond this threshold, relying on more advanced algorithms, such as shell sort, quicksort, or heapsort, becomes imperative to ensure timely and efficient sorting.

Appendix:

a) Code for direct insertion sort

```
public class Insertionsort {

    /**
     * The main function where the Insertionsort algorithm is defined.
     *
     * @param a An array that needs to be sorted
     * @param n The size of the array being passed from Main
     */
    public static void sort(int[] a, int n) {                    // Insertion implementation
        for (int i = 1; i <= n - 1; ++i) {                       // Find a number and moves it back until
            int x = a[i];                                        // it finds its perfect place
            int j = i;

            while (j > 0 && a[j - 1] > x) {                      // Move element one ahead to create space
                a[j] = a[j - 1];                                 // for the coming element
                j = j - 1;
            }
            a[j] = x;
        }
    }
}
```

b) Code for direct selection sort

```
public class Selectionsort {

    /**
     * Main functions where selection sort is performed. Goes through the array constantly checking if an
     * index is smaller that the one started with. The index could be replaced n times if the array is reversed.
     *
     * @param a An array that is needed to be sorted
     * @param n Holds the size of the array
     * @return Returns the array to the caller.
     */
    public static int[] selectionsort(int[] a, int n) {
        for (int i = 0; i < n - 1; ++i) {                        // First loop
```

```
            int index = i;
            for (int j = i + 1; j < n; j++) {                 // While holding on i, go through the array
                if (a[j] < a[index]) {
                    index = j;
                }
            }
            swap(a, index, i);                                // Swap in the smaller number
            //System.out.print("Hang tight, we are " + loading(i, n) + "% done. " + (i + 2) + "/" + n + "\r");
        }                                                     // Loading commented since it slowed the
        return a;                                             // algorithm down too much
    }

    /**
     * Swap is a basic function that exchanges the value of two array indexes. This is done by the
     * help of a temporary variable holding the value of one while they get exchanged.
     *
     * @param a An array that we are sorting
     * @param i The position that should hold a smaller int but doesn't at the moment
     * @param j The position that should hold a bigger  int but doesn't at the moment
     */
    private static void swap(int[] a, int i, int j) {          // Swaps the smallest number found in the array
        int temp = a[i];                                      // to where j was at in selectionsort()
        a[i] = a[j];
        a[j] = temp;
    }

     * @param i Array index currently at
     * @param n The size of the array
     * @return Returns a percentage using numberformat to only be in an int format.
    private static String loading(int i, int n) {              // Selectionsort can take a long time
        float x = ((float) (i + 2) / (float) n) * 100f;        // Progress bar wouldn't hurt.
        NumberFormat numberformat = new DecimalFormat("#");
        return numberformat.format(x);
    }
}
```

c) Code for bubble sort

```
public class Bubblesort {

    /**
     * Bubblesort is the main function that follows the Bubble sort algorithm. It starts with a flag of
     * false and will it set it to true if it ever had to swap to values. If no swaps were done, the loop
     * will then exit.
     *
     * @param a An array that needs to be sorted
     * @param n The size of the array
     */
    public static void BubbleSort(int[] a, int n) {
        boolean flag;                                         // Loop condition controller

        do {
            flag = false;                                     // Start with a flag of false
            for (int i = 0; i < n - 1; ++i) {
                if (a[i] > a[i + 1]) {                        // Array is still not sorted
                    swap(a, i, i + 1);                        // will have to loop again
                    flag = true;
                }
            }
        } while (flag);
    }

    /**
     * Swap is a basic function that exchanges the value of two array indexes. This is done by the
     * help of a temporary variable holding the value of one while they get exchanged.
     *
     * @param a An array that we are sorting
     * @param i The position that should hold a smaller int but doesn't at the moment
     * @param j The position that should hold a bigger  int but doesn't at the moment
     */
    private static void swap(int[] a, int i, int j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

d) Code for shell sort

```
import java.util.Arrays;

public class ShellSort {
     * @param array
     * @return
     */
    public static void ShellSort(int[] array) {
        int len = array.length;
        int temp, gap = len / 2;
        while (gap > 0) {
            for (int i = gap; i < len; i++) {
                temp = array[i];
                int preIndex = i - gap;
                while (preIndex >= 0 && array[preIndex] < temp) {
                    array[preIndex + gap] = array[preIndex];
                    preIndex -= gap;
                }
                array[preIndex + gap] = temp;
            }
            gap /= 2;
        }
    }
}
```

e) Code for quick sort

```
public class Quicksort {
    private static int partition(int a[], int low, int high) {
        int pivot = a[(low + high) / 2];                          // Finding the pivot

        while (low <= high) {
            while (a[low] < pivot)
                low++;
            while (a[high] > pivot)
                high--;
            if (low <= high) {
                swap(a, low++, high--);
            }
        }
        return low;
    }

    public static void quickSort(int a[], int low, int high) {
        int index = partition(a, low, high);
        if (low < index - 1)
            quickSort(a, low, index - 1);
        if (index < high)
            quickSort(a, index, high);
    }

    /**
     * Swap is a basic function that exchanges the value of two array indexes. This is done by the
     * help of a temporary variable holding the value of one while they get exchanged.
     *
     * @param a    An array that we are sorting
     * @param low  The position that should hold a smaller int but doesn't at the moment
     * @param high The position that should hold a bigger  int but doesn't at the moment
     */
    private static void swap(int[] a, int low, int high) {        // Swaps the smallest number found in the array
        int temp = a[low];                                       // to where j was
        a[low] = a[high];
        a[high] = temp;
    }
}
```

f) Code for heap sort

```
import java.util.Arrays;

public class HeapSort {

     * @param array
     * @param parentIndex
     * @param length
     */

public static void downAdjust(int[] array, int parentIndex, int length) {
    int temp = array[parentIndex];
    int childIndex = 2 * parentIndex + 1;

    while (childIndex < length) {
        if (childIndex + 1 < length && array[childIndex + 1] > array[childIndex]) {
            childIndex++;
        }

        if (temp >= array[childIndex])
            break;

        array[parentIndex] = array[childIndex];
        parentIndex = childIndex;
        childIndex = 2 * childIndex + 1;
    }
    array[parentIndex] = temp;
}

 * @param array
 */

public static void heapSort(int[] array) {

    for (int i = (array.length-2)/2; i >= 0; i--) {
        downAdjust(array, i, array.length);
    }
    for (int i = array.length - 1; i > 0; i--) {

        int temp = array[i];
        array[0] = array[i];
        array[0] = temp;

        downAdjust(array, 0, i);
    }
}
}
```

g) Code for the main testing program

```
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.HashMap;
import java.util.Map;

 */
public class Main {
    private static Map<Integer, String> data = new HashMap<>();
    private final static int SORT_IMPLEMENTED = 6;
    private static int leaderinsertion = 0;
    private static int leaderselection = 0;
    private static int leaderbubble = 0;
    private static int leadermerge = 0;
```

```
    private static int leaderquick = 0;
    private static int leadershell = 0;
    private static int size = 10;
    private static int loop = 5;
    private static int current = 0;
    private static int[][] a = new int[loop][];
    private static int[][] b = new int[loop][];


    public static void main(String[] args) {
        try {
            if (args.length != 0) {
                throw new Exception("Program can not accept additional arguments.");
            }

            /**
             * We will loop four times to check each sorting algorithm with four different array types.
             * 1. Random
             * 2. Sorted
             * 3. Sorted-reverse
             * 4. Few unique hours
             */
            for (int arraytype = 0; arraytype < 4; ++arraytype) {
                size = 10;
                long totalTime;
                long endTime;
                long startTime;
                long Selection = 0;
                long Merge = 0;
                long Insertion = 0;
                long Bubble = 0;
                long Quick = 0;
                long shell = 0;

                if (arraytype == 0) {
                    for (int i = 0; i < loop; ++i) {
                        a[i] = randomarray();
                        size *= 10;
                    }
                } else if (arraytype == 1) {
                    for (int i = 0; i < loop; ++i) {
                        a[i] = inorderarray();
                        size *= 10;
                    }
                } else if (arraytype == 2) {
                    for (int i = 0; i < loop; ++i) {
                        a[i] = reversearray();
                        size *= 10;
                    }
                } else if (arraytype == 3) {
                    for (int i = 0; i < loop; ++i) {
                        a[i] = similararray();
                        size *= 10;
                    }
                }

                for (int i = 0; i < loop; i++) {
                    b[i] = new int[a[i].length];
                    System.arraycopy(a[i], 0, b[i], 0, a[i].length);
                }

                ready("First", "Selection");

                for (int i = 0; i < loop; ++i) {
                    startTime = System.currentTimeMillis();
                    Selectionsort.selectionsort(a[i], a[i].length);
                    endTime = System.currentTimeMillis();
                    totalTime = endTime - startTime;
                    Selection += totalTime;
                    System.out.println("Time for array of size " + a[i].length + ": " + totalTime + " milliseconds.");
                }
                System.out.println("Total Time: " + Selection + " milliseconds.");

                recopy();
                //ready("Second", "Mergesort");
                ready("Second", "HeapSort");

                for (int i = 0; i < loop; ++i) {
                    startTime = System.currentTimeMillis();
                    //Mergesort.divide(a[i]);
                    HeapSort.heapSort(a[i]);
                    endTime = System.currentTimeMillis();
                    totalTime = endTime - startTime;
                    Merge += totalTime;
                    System.out.println("Time for array of size " + a[i].length + ": " + totalTime + " milliseconds.");
                }
                System.out.println("Total Time: " + Merge + " milliseconds.");

                recopy();
                ready("Third", "Insertion");

                for (int i = 0; i < loop; ++i) {
                    startTime = System.currentTimeMillis();
                    Insertionsort.sort(a[i], a[i].length);
                    endTime = System.currentTimeMillis();
                    totalTime = endTime - startTime;
                    Insertion += totalTime;
                    System.out.println("Time for array of size " + a[i].length + ": " + totalTime + " milliseconds.");
                }
                System.out.println("Total Time: " + Insertion + " milliseconds.");

                recopy();
```

```
            ready("Fourth", "Bubble");

            for (int i = 0; i < loop; ++i) {
                startTime = System.currentTimeMillis();
                Bubblesort.BubbleSort(a[i], a[i].length);
                endTime = System.currentTimeMillis();
                totalTime = endTime - startTime;
                Bubble += totalTime;
                System.out.println("Time for array of size " + a[i].length + ": " + totalTime + " milliseconds.");
            }
            System.out.println("Total Time: " + Bubble + " milliseconds.");

            recopy();
            ready("Fifth", "Quick");

            for (int i = 0; i < loop; ++i) {
                startTime = System.currentTimeMillis();
                Quicksort.quickSort(a[i], 0, a[i].length - 1);
                endTime = System.currentTimeMillis();
                totalTime = endTime - startTime;
                Quick += totalTime;
                System.out.println("Time for array of size " + a[i].length + ": " + totalTime + " milliseconds.");
            }
            System.out.println("Total Time: " + Quick + " milliseconds.");
            recopy();
            ready("Sixth", "Shell");

            for (int i = 0; i < loop; ++i) {
                startTime = System.currentTimeMillis();
                ShellSort.ShellSort(a[i]);
                endTime = System.currentTimeMillis();
                totalTime = endTime - startTime;
                shell += totalTime;
                System.out.println("Time for array of size " + a[i].length + ": " + totalTime + " milliseconds.");
            }
            System.out.println("Total Time: " + shell + " milliseconds.");

            compare("Selection", Selection, "Heap", Merge);
            compare("Insertion", Insertion, "Heap", Merge);
            compare("Bubblesort", Bubble, "Heap", Merge);
            compare("Quick", Quick, "Heap", Merge);
            compare("Shell", shell, "Heap", Merge);
            System.out.println();
            compare("Selection", Selection, "Bubblesort", Bubble);
            compare("Insertion", Insertion, "Bubblesort", Bubble);
            compare("Heap", Merge, "Bubblesort", Bubble);
            compare("Quick", Quick, "Bubblesort", Bubble);
            compare("Shell", shell, "Bubblesort", Bubble);
            System.out.println();
            compare("Insertion", Insertion, "Selection", Selection);
            compare("Bubblesort", Bubble, "Selection", Selection);
            compare("Heap", Merge, "Selection", Selection);
            compare("Quick", Quick, "Selection", Selection);
            compare("Shell", shell, "Selection", Selection);
            System.out.println();
            compare("Selection", Selection, "Insertion", Insertion);
            compare("Bubblesort", Bubble, "Insertion", Insertion);
            compare("Heap", Merge, "Insertion", Insertion);
            compare("Quick", Quick, "Insertion", Insertion);
            compare("Shell", shell, "Insertion", Insertion);
            System.out.println();
            compare("Selection", Selection, "Quick", Quick);
            compare("Insertion", Insertion, "Quick", Quick);
            compare("Bubblesort", Bubble, "Quick", Quick);
            compare("Heap", Merge, "Quick", Quick);
            compare("Shell", shell, "Quick", Quick);
            System.out.println();
            compare("Selection", Selection, "Shell", shell);
            compare("Insertion", Insertion, "Shell", shell);
            compare("Bubblesort", Bubble, "Shell", shell);
            compare("Heap", Merge, "Shell", shell);
            compare("Quick", Quick,"Shell", shell);

            data.put(leaderinsertion, "Insertion");
            data.put(leaderselection, "Selection");
            data.put(leaderbubble, "Bubble");
            data.put(leadermerge, "Heap");
            data.put(leaderquick, "Quick");
            data.put(leadershell, "Shell");

            leaderboard();
        }
    } catch (InterruptedException e) {
        System.out.println("I had trouble using the sleep function. Here is what happened:");
        System.out.println(e.getMessage());
        System.exit(1);
    } catch (Exception e) {
        System.out.println("I had trouble running the main function. Here is what happened:");
        System.out.println(e.getMessage());
        System.exit(1);
    }
}

/**
 * Sorts and prints algorithms based on which ever performed the best to which ever performed
 * the worse. Uses the map data type to link an Integer value to its String.
 */
private static void leaderboard() {
    int order = 1;
    System.out.println("\n");
    System.out.println("Leaderboard:\n");
```

```
        for (int i = 0; i < SORT_IMPLEMENTED; ++i) {
            String result = data.get(i);
            System.out.println(order + ") " + result);
            ++order;
        }
    }

    /**
     * Compares two sorting algorithms using their specific recorded timings. A specific
     * print out is done depending which is bigger or smaller than the other. A number
     * formatter is used to force only to the 4th zero position if more were available.
     *
     * @param First  The name of the first sorting algorithm.
     * @param first  The time it took to finish the first sorting algorithm.
     * @param Second The name of the second sorting algorithm.
     * @param second The time it took to finish the second sorting algorithm.
     */
    private static void compare(String First, long first, String Second, long second) {
        float x = ((float) first / (float) second);
        NumberFormat numberformat = new DecimalFormat("##.####");
        String s = numberformat.format(x);

        if (first > second) {
            System.out.print("\n" + Second + " was " + s + " times faster than " + First + ".");
            switch (First) {
                case "Shell":
                    ++leadershell;
                case "Selection":
                    ++leaderselection;
                    break;
                case "Insertion":
                    ++leaderinsertion;
                    break;
                case "Bubblesort":
                    ++leaderbubble;
                    break;
                case "Heap":
                    ++leadermerge;
                    break;
                case "Quick":
                    ++leaderquick;
            }
        } else {
            System.out.print("\n" + Second + " was " + s + " the speed of " + First + ".");
        }
    }

    /**
     * Creates a random array using the current size which should be multiples of 10.
     *
     * @return Returns a pointer to the newly created array.
     */
    static int[] randomarray() {
        int[] a = new int[size];

        for (int i = 0; i < a.length; ++i) {
            a[i] = randomnumber();
        }
        return a;
    }

    static int[] inorderarray() {
        int[] a = new int[size];

        for (int i = 0; i < size; ++i) {
            a[i] = i;
        }
        return a;
    }

    static int[] reversearray() {
        int[] a = new int[size];
        int k = 0;

        for (int i = size; i > 0; --i) {
            a[k++] = i-1;
        }

        return a;
    }

    static int[] similararray() {
        int[] a = new int[size];

        for (int i = 0; i < size; ++i) {
            a[i] = (int) ((Math.random() * 10) + 1) * size;
        }

        return a;
    }

    /**
     * randomnumber returns an int proportional to the size of an array. If size is 10 for example, the range
     * is within 10 to 109 and 100 to 1099 and etc...
     *
     * @return An int that is randomly generated and returned to the caller
     */
    static int randomnumber() {
        return (int) (((Math.random() * 10) + 1) * size);
    }

    /**
```

```
    * Ready is a setup function that only deals with print outs and nothing else. It has a feeling of a
    * "ready, set, go" just for a more user friendly print out.
    *
    * @param position The rank it is being applied. ex: First, Second, Third, Fourth, Fifth, etc...
    * @param sort     The name of the sorting algorithm that will be applied in a moment
    * @throws InterruptedException If there are any issues with sleep, function will throw its exceptions
    */
   static void ready(String position, String sort) throws InterruptedException {
       if (current == 0) {
           ++current;
       } else {
           System.out.println();
       }

       System.out.print(position + ", " + sort + ".");
       Thread.sleep(1000);
       System.out.print(".");
       Thread.sleep(1000);
       System.out.print(".\n");
       Thread.sleep(1000);
   }

   /**
    * Recopy takes array b that holds an array of arrays that was randomly generated and
    * copies it back into array a after each sorting algorithm been applied. This way the same
    * array is applied to each sorting algorithm.
    */
   static void recopy() {
       for (int i = 0; i < loop; i++) {
           System.arraycopy(b[i], 0, a[i], 0, a[i].length);
       }
   }
}
```

References

[1] Bhalchandra, P. et al. (2009) ‘A Comprehensive Note on Complexity Issues in Sorting Algorithms’, Advances in Computational Research, 1(2). doi:10.9735/0975-3273.

[2] Gill, S.K. et al. (2018) ‘A Comparative Study of Various Sorting Algorithms’, INTERNATIONAL JOURNAL OF ADVANCED STUDIES OF SCIENTIFIC RESEARCH, Special Issure based on proceedings of 4th International Conference on Cyber Security (ICCS).

[3] Prajapati, P. (2017) ‘Performance Comparison of Different Sorting Algorithms’, International Journal of Latest Technology in Engineering, Management & Applied Science, VI(VI). doi:10.51583/ijltemas.

[4] Bei, Q. (2022) Detailed Description of Eight Different Sorting Algorithm, CSDN. Available at: https://blog.csdn.net/weixin_61661271/article/details/126144187 (Accessed: 04 October 2023).