Name-Surname: Nouredeen Ahmed Mahmoud Ali HAMMAD
Student no: 2121221362
Class: BLM19307E Algorithm Analysis

Project 1 - Analysis of Sorting Algorithms

Theoretical Section

I - Selection Sort

Time Complexity:

The time complexity of selection sort is determined by the number of comparisons and swaps it makes.

Comparisons: Selection sort makes n−1 comparisons for the first element, n−2 for the second, and so on, up to 1 comparison for the last element. The total number of comparisons is the sum of the first n−1 integers, which is 2n(n−1), i.e., $O(n^2)$.

Swaps: For each element, there is at most one swap. In the worst case, this results in n swaps. Therefore, the worst-case time complexity for swaps is O(n).

Therefore, the worst-case time complexity of selection sort is $O(n^2)$.

Iterative Equation:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

Space Complexity:

It is an in place algorithm. It doesn't require any additional space, so its space complexity is O(1).

Pseudocode:

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

II - Insertion Sort

Time Complexity:

The time complexity of insertion sort is determined by the number of comparisons and swaps it makes.

Comparisons: In the worst case scenario (when the array is reverse sorted), each insertion takes i steps where i is the current index. Therefore, the worst-case time complexity is the sum of first n natural numbers, minus 1 (since indexing starts from 0), which is $2n(n-1)$, i.e., $O(n^2)$.

Swaps: Each swap operation occurs after a comparison where a 'greater' element is found. Therefore, in the worst case, the number of swaps is also $2n(n-1)$, i.e., $O(n^2)$.

In the best case scenario (when the array is already sorted), there are $n-1$ comparisons and 0 swaps, so the best-case time complexity is $O(n)$[1][2].

Iterative Equations:

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$$

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

Space Complexity:

It is an in place algorithm. It doesn't require any additional space, so its space complexity is O(1).

Pseudocode:

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

III - Shell Sort

Time Complexity:

The time complexity of Shell Sort is heavily dependent on the gap sequence it uses. However, here are some general observations:

1. Worst-case performance: The worst-case time complexity of Shell Sort is $O(n^2)$ for the worst known gap sequence, and $O(n\log^2 n)$ for the best known gap sequence.
2. Best-case performance: The best-case time complexity of Shell Sort is O(nlogn) for most gap sequences, and $O(n\log^2 n)$ for the best known worst-case gap sequence.
3. Average performance: The average time complexity of Shell Sort depends on the gap sequence. As per big-O notation, Shell Sort has $O(n^{1.25})$ average time complexity.

Space Complexity:

It is an in place algorithm. It doesn't require any additional space, so its space complexity is O(1).

Pseudocode:

Algorithm 1: Shellsort

Data: $x = [x_1, x_2, \ldots, x_n]$: the array to sort, $h = [h_1, h_2, \ldots, h_m]$: the gap sequence.

Result: x is sorted.

for $k = 1, 2, \ldots, m$ **do**

  for $i = 1, 2, \ldots, h_k + 1$ **do**

    Apply Insertion Sort to chain

    $\left[x[i + t(h_k + 1)] \mid t = 0, 1, \ldots, \lfloor \frac{n-i}{h_k+1} \rfloor\right]$.

  end

end

IV - Merge Sort

Time Complexity:

The time complexity of merge sort can be analyzed as follows:

Divide: The divide step computes the middle of the subarray, which takes constant time. Thus, the divide step has a time complexity of O(1).

Conquer: The conquer step recursively sorts two subarrays of n/2 elements each. If T(n) is the time complexity of sorting n elements, then this step is 2T(n/2).

Combine: The combine step merges n elements, taking O(n) time.

Therefore, the time complexity T(n) satisfies the recurrence relation:
T(n)=2T(n/2)+O(n)

By the master theorem, the solution to this recurrence is O(nlogn).

$$C_{worst}(n) = n \log_2 n - n + 1$$

Space Complexity:

Merge sort requires additional space to hold the two halves while merging. In the worst case, a temporary array of size n is needed. Therefore, the space complexity of merge sort is O(n).

Pseudocode:

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)    //see below
```

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

V - 3 Way Merge Sort

Time Complexity:

The time complexity of 3-Way Merge Sort can be analyzed as follows:

Divide: The divide step computes the two midpoints of the subarray, which takes constant time. Thus, the divide step has a time complexity of O(1).

Conquer: The conquer step recursively sorts three subarrays of approximately n/3 elements each. If T(n) is the time complexity of sorting n elements, then this step is 3T(n/3).

Combine: The combine step merges a total of n elements, taking O(n) time.

Therefore, the time complexity T(n) satisfies the recurrence relation:
$T(n)=3T(n/3)+O(n)$

By the master theorem, the solution to this recurrence is O(nlogn).

Space Complexity:

3-Way Merge Sort requires additional space to hold the three halves while merging. In the worst case, a temporary array of size n is needed. Therefore, the space complexity of 3-Way Merge Sort is O(n).

Pseudocode: (insert 3 in place of K)

Algorithm 1: Pseudocode example for K-Way merge

```
Data: Initial state
Result: What you want to obtain
Int minItem = MinIndex(Item, k);
processItem(minItem);
if item(minItem) =Item(i) then
    More Items[i]= NextItemIntheList(i);
else
    return to the first step;
end
```

VI - Quick Sort

Lomuto's Partitioning:

In Lomuto's partition scheme, the pivot element is assumed to be the last element. If any other element is given as a pivot element then it is swapped first with the last element.
Time Complexity

Worst-case performance: The worst-case time complexity of QuickSort with Lomuto partitioning is $O(n^2)$, which occurs when the input array is already sorted or reverse sorted.
Average performance: On average, the time complexity of QuickSort with Lomuto partitioning is O(nlogn).

The space complexity of QuickSort with Lomuto partitioning is O(1), as it is an in-place sorting algorithm.

Hoare's Partitioning:

Hoare's partition scheme works by initializing two indexes that start at two ends, the two indexes move toward each other until an inversion is found. When an inversion is found, two values are swapped and the process is repeated.

Time Complexity

Worst-case performance: The worst-case time complexity of QuickSort with Hoare partitioning is O(n2), which occurs when the input array is already sorted or reverse sorted.
Average performance: On average, the time complexity of QuickSort with Hoare partitioning is O(nlogn).

The space complexity of QuickSort with Hoare partitioning is O(logn), as it requires logn amount of additional space for recursive function calls.

Pseudocode:

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$*Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//        as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])  //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

ALGORITHM *LomutoPartition*$(A[l..r])$

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
//        indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;  swap(A[s], A[i])
swap(A[l], A[s])
return s
```

VII - Heap Sort

Time Complexity:

The time complexity of HeapSort can be analyzed as follows:

Building a Heap: Building a heap from an array of n elements takes O(n) time.

Heapify: The Heapify procedure, which is used to maintain the heap property (where parent node is either greater than or equal to its children for max heap or less than or equal to its children for min heap), takes O(logn) time in the worst case, as it's called once for each node of the heap.

Sorting: Finally, we perform n delete operations, removing the root each time and calling Heapify. Since we're doing n deletions, each taking O(logn) time, this step contributes O(nlogn) to the overall time complexity.

Therefore, the total time complexity of HeapSort is O(n)+O(nlogn)=O(nlogn).

Space Complexity:

HeapSort is an in-place sorting algorithm. It does not require any additional space that scales with the size of the input list, so its space complexity is O(1).

Pseudocode:

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

Empirical Section

Experimental design:

Sorting Algorithm Interface:

To be able to perform the experiment in a way that is easily scalable with clean, easy to read code, a Java Interface named **SortingAlgorithm** that contains one non-static method named **run** was created. Every sorting algorithm class implements this interface.

```
public interface SortingAlgorithm {
    void run(int[] arr);
}
```

The main function does the following.

Array Generation:

The main function generates three types of arrays: random, reverse sorted, and almost sorted. These arrays are used as input to compare the sorting algorithms.

Dataset Initialization:

A DefaultCategoryDataset named dataSet is created. This dataset will store the results of the execution times for each sorting algorithm on different types of arrays.

Algorithm Execution and Filling Dataset:

The **getExecTimesAndAddToDataset** function is called three times, once for each type of array (random, reverse sorted, and almost sorted). This function calculates the execution times of each sorting algorithm on the given array and adds the results to the dataset.

Chart Creation and Display:

After populating the dataset, a bar chart (**BarChart** class from the JFreeChart library) is created with relevant labels and titles.
The chart is packed and then set to be visible, displaying the comparison of execution times for different sorting algorithms on different types of arrays.

Utility Functions:

The program utilizes utility functions from the **Utils** class to generate arrays, measure execution times, and convert data types.

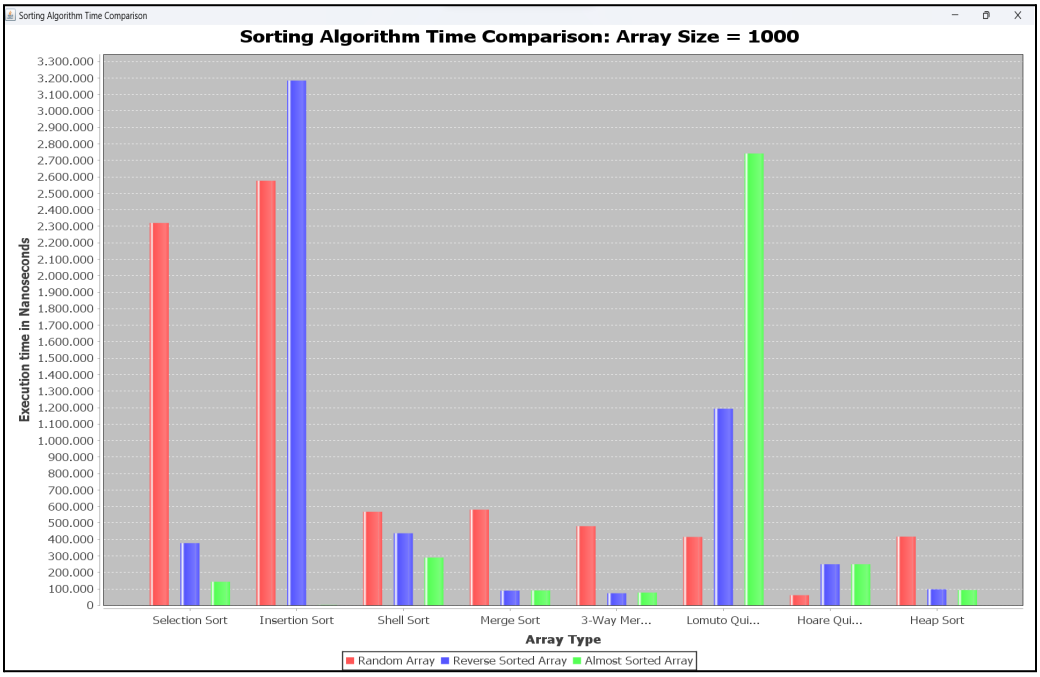
Unit Tests:

Unit tests have been made using JUnit 4 to make sure all the sorting algorithms produce correctly sorted arrays.

Result Analysis:

This experiment was performed using integer arrays of sizes **1000**, **5000** and **15000**.

Array of size 1000:

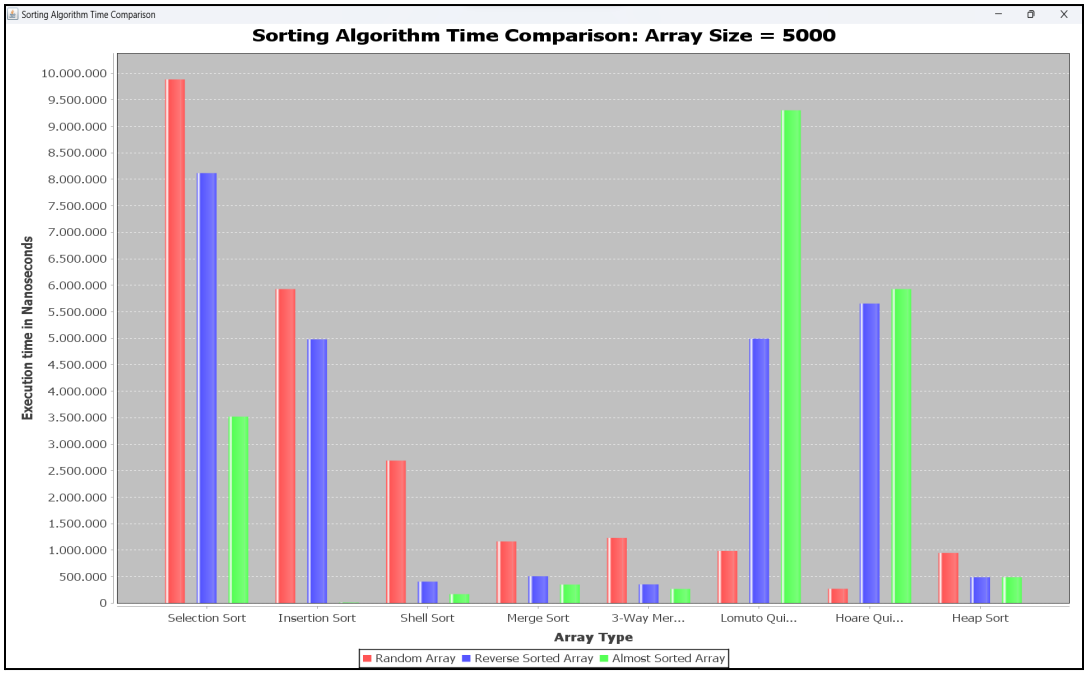


```
run:
Random Array
Execution time for SelectionSort: 2261800 nanoseconds
Execution time for InsertionSort: 2501700 nanoseconds
Execution time for ShellSort: 616300 nanoseconds
Execution time for MergeSort: 525500 nanoseconds
Execution time for ThreeWayMergeSort: 423400 nanoseconds
Execution time for QuickSortLomuto: 456900 nanoseconds
Execution time for QuickSortHoare: 69400 nanoseconds
Execution time for HeapSort: 394300 nanoseconds

Reverse Sorted Array
Execution time for SelectionSort: 373100 nanoseconds
Execution time for InsertionSort: 4644400 nanoseconds
Execution time for ShellSort: 429900 nanoseconds
Execution time for MergeSort: 78000 nanoseconds
Execution time for ThreeWayMergeSort: 70900 nanoseconds
Execution time for QuickSortLomuto: 1193000 nanoseconds
Execution time for QuickSortHoare: 195000 nanoseconds
Execution time for HeapSort: 97000 nanoseconds

Almost Sorted Array
Execution time for SelectionSort: 143400 nanoseconds
Execution time for InsertionSort: 1500 nanoseconds
Execution time for ShellSort: 297200 nanoseconds
Execution time for MergeSort: 79600 nanoseconds
Execution time for ThreeWayMergeSort: 90900 nanoseconds
Execution time for QuickSortLomuto: 4167600 nanoseconds
Execution time for QuickSortHoare: 614000 nanoseconds
Execution time for HeapSort: 131200 nanoseconds
```

Array of size 5000:

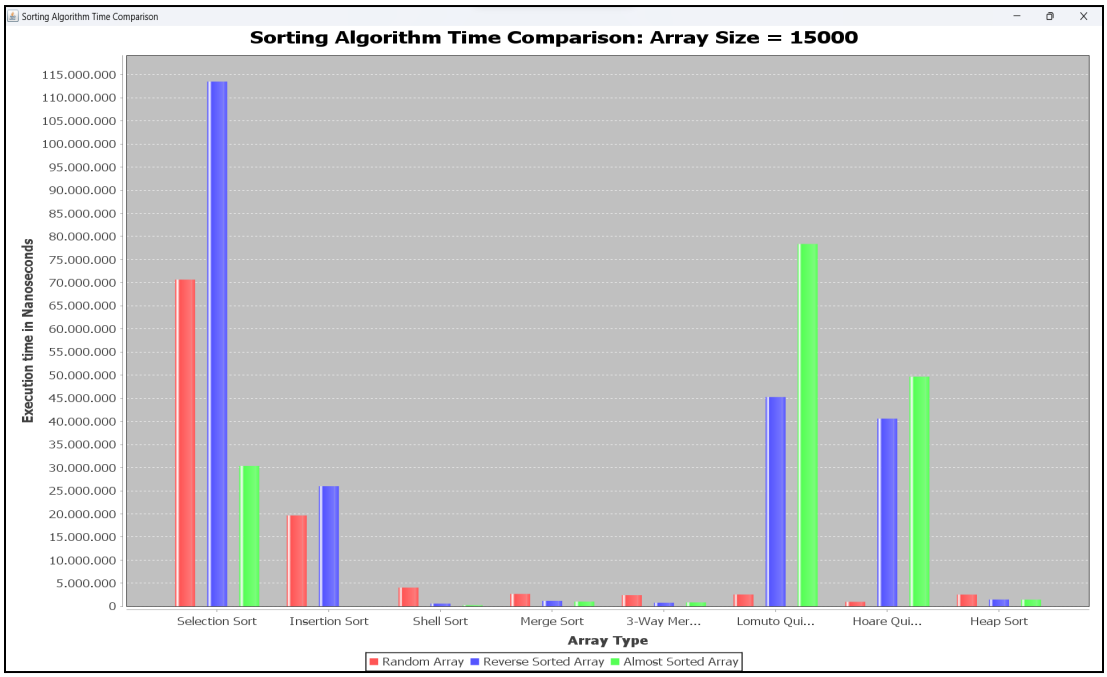



```
run:
Random Array
Execution time for SelectionSort: 10296900 nanoseconds
Execution time for InsertionSort: 5958100 nanoseconds
Execution time for ShellSort: 2704100 nanoseconds
Execution time for MergeSort: 1185900 nanoseconds
Execution time for ThreeWayMergeSort: 1285700 nanoseconds
Execution time for QuickSortLomuto: 914100 nanoseconds
Execution time for QuickSortHoare: 281600 nanoseconds
Execution time for HeapSort: 990400 nanoseconds

Reverse Sorted Array
Execution time for SelectionSort: 8352700 nanoseconds
Execution time for InsertionSort: 5091300 nanoseconds
Execution time for ShellSort: 256800 nanoseconds
Execution time for MergeSort: 2371900 nanoseconds
Execution time for ThreeWayMergeSort: 347600 nanoseconds
Execution time for QuickSortLomuto: 5269200 nanoseconds
Execution time for QuickSortHoare: 6249000 nanoseconds
Execution time for HeapSort: 476000 nanoseconds

Almost Sorted Array
Execution time for SelectionSort: 3431000 nanoseconds
Execution time for InsertionSort: 6800 nanoseconds
Execution time for ShellSort: 117400 nanoseconds
Execution time for MergeSort: 1199600 nanoseconds
Execution time for ThreeWayMergeSort: 262400 nanoseconds
Execution time for QuickSortLomuto: 8362500 nanoseconds
Execution time for QuickSortHoare: 5886300 nanoseconds
Execution time for HeapSort: 472100 nanoseconds
```

Array of size 15000:




```
run:
Random Array
Execution time for SelectionSort: 75720000 nanoseconds
Execution time for InsertionSort: 17110000 nanoseconds
Execution time for ShellSort: 3978300 nanoseconds
Execution time for MergeSort: 2745500 nanoseconds
Execution time for ThreeWayMergeSort: 2493100 nanoseconds
Execution time for QuickSortLomuto: 2780500 nanoseconds
Execution time for QuickSortHoare: 890200 nanoseconds
Execution time for HeapSort: 2843500 nanoseconds

Reverse Sorted Array
Execution time for SelectionSort: 100672000 nanoseconds
Execution time for InsertionSort: 25482000 nanoseconds
Execution time for ShellSort: 610000 nanoseconds
Execution time for MergeSort: 1085500 nanoseconds
Execution time for ThreeWayMergeSort: 769900 nanoseconds
Execution time for QuickSortLomuto: 41249500 nanoseconds
Execution time for QuickSortHoare: 40555900 nanoseconds
Execution time for HeapSort: 2031900 nanoseconds

Almost Sorted Array
Execution time for SelectionSort: 31605000 nanoseconds
Execution time for InsertionSort: 19300 nanoseconds
Execution time for ShellSort: 107400 nanoseconds
Execution time for MergeSort: 1023000 nanoseconds
Execution time for ThreeWayMergeSort: 625600 nanoseconds
Execution time for QuickSortLomuto: 80179200 nanoseconds
Execution time for QuickSortHoare: 49939200 nanoseconds
Execution time for HeapSort: 1476300 nanoseconds
```

Conclusions:

Random Array:

Quick Sort with Hoare's partitioning shows the best performance on random arrays. Shell Sort, Merge Sort, Three-Way Merge Sort and Quick Sort with Lumoto's partitioning algorithms also perform well.

Reverse Sorted Array:

Shell Sort and Three-Way Merge Sort are the fastest in this scenario. Both versions of Quick Sort perform very badly in this case. Selection sort remains the worst with $O(n^2)$ time complexity.

Almost Sorted Array:

Insertion Sort, Shell Sort, and Merge Sort outperform the other algorithms on almost sorted arrays. Both Quick Sort types show much higher execution time here.

Overall Performance:

Merge Sort, Three-Way Merge Sort, Shell Sort and Heap Sort consistently demonstrate good performance across all array types. They are reliable choices for general-purpose sorting.

For random arrays:
Quick Sort with Hoare's partitioning is the absolute best, Lumoto's partitioning system is slightly worse.

Special Cases:
For almost sorted arrays, Insertion Sort shines. Shell Sort also performs well in this scenario.

Reverse sorted arrays favor algorithms with good adaptability, such as Shell Sort and Merge Sort.

In conclusion, the choice of sorting algorithm depends on the characteristics of the input data. No single algorithm is the best for all scenarios, but understanding their behavior in different situations allows for informed decision-making based on the specific requirements of the application.

Sources:

Khan Academy
Introduction To The Design And Analysis Of Algorithms 3rd edition
Selection Sort – Data Structure and Algorithm Tutorials - GeeksforGeeks
Insertion Sort - Data Structure and Algorithm Tutorials - GeeksforGeeks
ShellSort - GeeksforGeeks
Merge Sort - Data Structure and Algorithms Tutorials - GeeksforGeeks
3-way Merge Sort - GeeksforGeeks
Hoare's vs Lomuto partition scheme in QuickSort - GeeksforGeeks
Heap Sort - Data Structures and Algorithms Tutorials - GeeksforGeeks