PG4200-H: Data Structures and Algorithms 2025 Report

INTRODUCTION

This report presents our implementation and analysis of four fundamental sorting algorithms: Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort. The assignment requires implementing these algorithms on the Wine Quality dataset with time complexity analysis and evaluation of how shuffling affects performance. We fulfill these requirements through systematic implementation and complexity analysis.

Additionally, following the instruction to "show your work and be creative," we complement the required theoretical analysis with empirical execution time measurements. While Big O notation predicts how algorithms scale, execution time reveals practical performance characteristics that Big O notation alone cannot capture.

Note on execution time variability: Absolute times are hardware-dependent and show natural variation across runs due to JVM behavior, garbage collection, and system load. **All measurements were conducted on the same laptop (plugged in to ensure consistent power delivery) with 3 warm-up runs followed by 10 measured executions per algorithm.** We report representative ranges observed across these runs to illustrate practical performance characteristics, not definitive benchmarks.

As (Neapolitan, 2015, s. 25) explains, execution time depends on factors such as basic operations, overhead instructions, and computer implementation. We cannot eliminate these inherent factors, but we can minimize external variability through systematic testing methodology. This approach reduces noise from JVM state (Obregon, 2025), garbage collection, and system load, allowing execution time patterns to emerge that validate our theoretical complexity analysis.

Report Structure

For each problem (1-4), we follow a consistent structure:

- Part (a): Implementation explanation with code snippets and execution results
- Part (b): Time complexity analysis addressing the specific questions asked

While we answer the specific assignment questions about time complexity in sections 1-4, our conclusion first summarizes these findings before synthesizing them with our additional execution time analysis to explore the relationship between theoretical complexity and practical performance.

.

PREPARATIONS

Before implementing the sorting algorithms, we performed initial data preparation to ensure the dataset was in a suitable format for our analysis.

DATASET PREPARATION

The Wine Quality dataset is provided in two separate CSV files: one for red wine (winequality-red.csv) and one for white wine (winequality-white.csv). We merged these files and extracted the unique alcohol content values as specified in the exam's problem statements. After removing duplicates, our dataset contains 112 unique alcohol values (n=112).

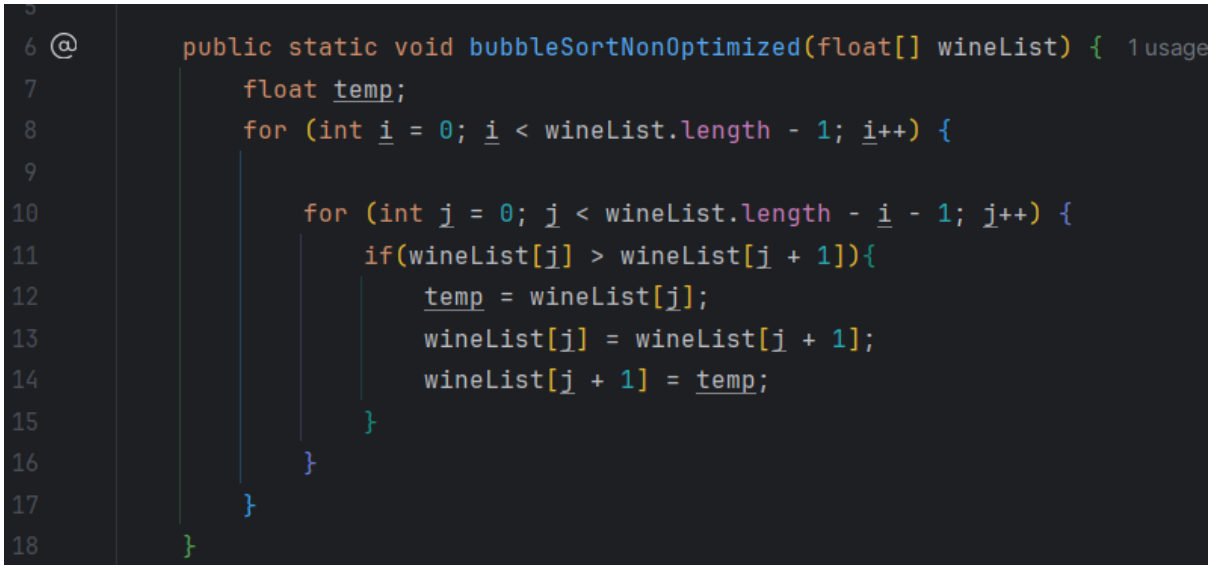
From a time complexity perspective, sorting only alcohol values versus complete wine objects does not change the theoretical complexity. Time complexity is determined by the number of comparisons and iterations relative to input size (n), not by the size of individual elements. As (Goodrich et al., 2014, s. 158) explains, algorithm analysis counts primitive operations as a function of input size, where each operation has constant execution time regardless of the data being processed.

This unique dataset (n=112) forms the basis for our primary analysis in sections 1-4. In our conclusion, we extend the analysis to the complete dataset (n=6,498) to demonstrate how algorithmic complexity and practical performance diverge at scale.

1A)

UN-OPTIMIZED BUBBLESORT

An un-optimized implementation of BubbleSort performs all n-1 passes through the array regardless of whether the list becomes sorted earlier. This means even if the list is partially or fully sorted, the algorithm completes every iteration (Programiz, 2025).



```
6  @    public static void bubbleSortNonOptimized(float[] wineList) {  1 usage
7           float temp;
8           for (int i = 0; i < wineList.length - 1; i++) {
9
10              for (int j = 0; j < wineList.length - i - 1; j++) {
11                  if(wineList[j] > wineList[j + 1]){
12                      temp = wineList[j];
13                      wineList[j] = wineList[j + 1];
14                      wineList[j + 1] = temp;
15                  }
16              }
17          }
18      }
```

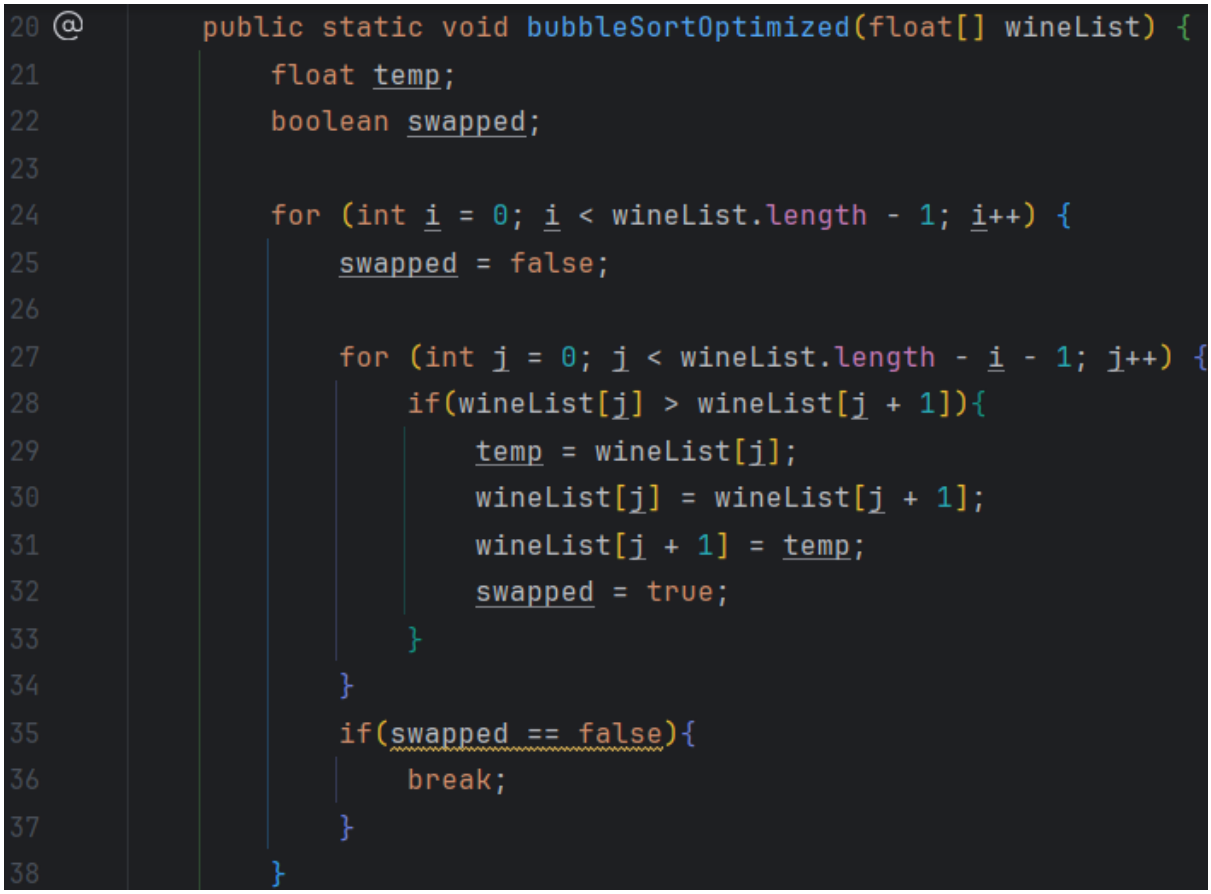
Code Snippet 1.1*: Un-optimized BubbleSort* (GeeksforGeeks, 2025a)

Key Components of the Implementation:

- **Outer loop (i) (line 8):** Iterates n-1 times, ensuring all elements are processed through multiple passes
- **Inner loop (j) (line 10):** Compares adjacent elements, iterating up to the unsorted portion of the array
- **Comparison and swap (lines 11-14):** When wineList[j] > wineList[j + 1], the elements are exchanged using a temporary variable
- **No early termination:** The algorithm always completes all n-1 passes, regardless of whether the list becomes sorted earlier

OPTIMIZED BUBBLESORT

An optimized implementation introduces an early termination mechanism using a boolean flag (swapped). If no swaps occur during a complete pass, the algorithm recognizes that the list is already sorted and stops immediately, avoiding unnecessary iterations (Programiz, 2025).



```
20 @    public static void bubbleSortOptimized(float[] wineList) {
21          float temp;
22          boolean swapped;
23
24          for (int i = 0; i < wineList.length - 1; i++) {
25              swapped = false;
26
27              for (int j = 0; j < wineList.length - i - 1; j++) {
28                  if(wineList[j] > wineList[j + 1]){
29                      temp = wineList[j];
30                      wineList[j] = wineList[j + 1];
31                      wineList[j + 1] = temp;
32                      swapped = true;
33                  }
34              }
35              if(swapped == false){
36                  break;
37              }
38          }
```

Code Snippet 1.2*: Optimized BubbleSort* (GeeksforGeeks, 2025a)

Key Components of the Implementation:

- **Swapped flag (line 22):** A boolean variable initialized before each pass to track whether any swaps occurred
- **Outer loop (i) (line 24):** Iterates up to n-1 times, but can terminate early based on the swapped flag
- **Inner loop (j) (line 27):** Compares adjacent elements, identical to the un-optimized version
- **Comparison and swap (lines 28-32):** When wineList [j] > wineList [j + 1], elements are exchanged and swapped is set to true

- **Early termination check (lines 35-37):** After each complete pass, if swapped == false, the algorithm breaks immediately as the list is already sorted

Execution Results:

```
Duration in nanoseconds: 464500
BubbleSort not optimized without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1,

Duration in nanoseconds: 149100
BubbleSort optimized without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1, 9.2,
```

Code Snippet 1.3 Bubble Sort *Execution Results*

Both implementations successfully sorted the dataset. The optimized version executed in 120,000-400,000 ns compared to 400,000-900,000 ns for the unoptimized version across multiple runs, demonstrating the practical benefit of early termination on partially sorted data.

1B)

TIME COMPLEXITY FOR BUBBLE SORT

```java
public static void bubbleSortOptimized(float[] wineList) {
    float temp;                                                 // O(1)
    boolean swapped;                                            // O(1)

    for (int i = 0; i < wineList.length - 1; i++) {             // Outer loop: O(N)
        swapped = false;                                        // O(1)

        for (int j = 0; j < wineList.length - i - 1; j++) {     // Inner loop: O(N)
            if(wineList[j] > wineList[j + 1]){                  // O(1)
                temp = wineList[j];                             // O(1)
                wineList[j] = wineList[j + 1];                  // O(1)
                wineList[j + 1] = temp;                         // O(1)
                swapped = true;                                 // O(1)
            }
        }
        if(swapped == false){                                   // O(1)
            break;
        }
    }
    // O(1) + O(1) + O(N) * [O(1) + O(N) * [O(1) + O(1) + O(1) + O(1) + O(1)] + O(1)]
    // Removing all constants since we're only interested in big O which is usually worst case.
    // O(N) * [O(N)] => O(N) * O(N) = O(N^2) WorstCase
}
```

Code Snippet 1.4 *BubbleSort Optimized* (GeeksforGeeks, 2025a)

TIME COMPLEXITY MEASUREMENT:

The time complexity of an algorithm is not assessed by the actual duration required to execute each statement within the code. Instead, it is evaluated based on the frequency with which each statement is executed. This distinction is important because the execution time of the algorithms is influenced by various factors, including hardware specifications, programming language used and the number of code statements. Meaning measuring algorithms by execution time is not a reliable method (GeeksforGeeks, 2025i; W3Schools, 2025).

CALCULATING THE TIME COMPLEXITY:

To calculate time complexity it is crucial to consider the frequency of statement execution, which is significantly affected by the size of the input data, in this case the winedataset. In the picture the total time complexity of the algorithm is expressed as O(1) + O(1) + O(N) * [O(1) + O(N) * [O(1) + O(1) + O(1) + O(1) + O(1)] + O(1)]. However, constants are typically disregarded in time complexity analysis,

as they become more insignificant as the input size (N) grows. Additionally, O(1) is a constant time complexity, which remains unchanged regardless of the input size of N which is winelist in this context.

We will be using worst, average and best case as examples for what happens with the time complexity if we shuffle the arrays before we run the algorithm, because they'll give us the upper bound(maximum) and lower bound(minimum) time the algorithm will take for the input size N while average case which is the average time complexity between upper bound and lower bound. Therefore worst case and best case would be two extremes of shuffling the array into a specific order like already sorted ascending or descending, while average case time complexity would be all other orders (GeeksforGeeks, 2025g).

We are usually only interested in the worst case which is the maximum time an algorithm takes and it's the easiest of the three to calculate, but in this case to show that shuffling may affect or not affect the algorithm we will add average and best case too (GeeksforGeeks, 2025g).

SIMPLIFY THE TIME COMPLEXITY:

If we simplify the time complexity by removing all the constants to O(N) * O(N) = O(N^2), this gives us a quadratic time complexity, due to the presence of nested loops (Rashmi Gupta, PowerPoint Slides, 15. September 2025). This means that the algorithm's execution time increases quadratic as input size N grows. The time complexity remains unaffected even if we shuffle the dataset, as it is dependent on the input size N, and not the elements inside.

BEST, AVERAGE AND WORST CASE TIME COMPLEXITY:

In the worst case scenario, the time complexity is O(N^2) for both optimized and unoptimized versions as it would run the outer loop N times the input size, and the inner loop will traverse and compare the elements and swap if needed N times the input size. In the best case scenario for an optimized sort, the time complexity becomes O(N) if the array it's supposed to shuffle is shuffled to an already sorted array. The reason is the outer loop only executes once, as the swapped flag is not activated in the inner loop, which results in the loop becoming O(1) a constant. This means that the time complexity becomes linear and scales linearly based on the input size N. But best case scenario for the unoptimized sort still remains O(N^2), as it lacks the mechanism to verify whether a swap has occurred, leading it to running the outer loop N amount of times even in the absence of swaps (GeeksforGeeks, 2025e). While shuffling may change the average case number of operations performed, it still does not change the time complexity it has as O(N^2) (GeeksforGeeks, 2025e).

To answer question about if the time complexity will change if we shuffle the array, it would not change since Bubble sort time complexity is similar on all cases, but in a case where the shuffling of the array results in an ascending order, the optimized Bubble Sort will change into O(N) because of best case scenario.

2A)

INSERTION SORT

Insertion Sort builds a sorted array by iterating through the input and inserting each element into its correct position within the already-sorted portion. The algorithm maintains a sorted subarray at the beginning and expands it by placing each new element in its appropriate location. For each element at position i (starting from index 1), the algorithm compares it with elements at position i-1, i-2, etc, shifting larger elements one position right until finding the correct insertion point (Neapolitan, 2015, s. 288-289).

```
7  @    public static void insertionSort(float[] wineList) {
8           for (int i = 1; i < wineList.length; i++) {
9               float temp = wineList[i];
10              int j = i - 1;
11
12              while (j >= 0 && wineList[j] > temp) {
13                  wineList[j + 1] = wineList[j];
14                  j = j - 1;
15              }
16              wineList[j + 1] = temp;
17          }
```

Code Snippet 1.5 *Insertion Sort* (GeeksforGeeks, 2025c)

Key Components of the Implementation:

- **Outer loop (i) (line 8):** Iterates from index 1 to the end of the array (i < wineList.length), treating the first element as already sorted. Each iteration expands the sorted portion by one element.
- **Temporary variable (temp) (line 9):** Stores the current element being inserted, allowing element shifts without data loss.
- **Inner while loop (line 12):** Moves backward through the sorted portion, shifting elements greater than temp one position right (lines 13-14).

- **Final insertion (line 16):** Places temp at position j+1 once the correct location is found (when j becomes -1 or wineList [j] ≤ temp).

```
Duration in nanoseconds: 610800
InsertionSort without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1, 9.2, 9.233334, 9.25,
```

Code Snippet 1.6 *Insertion Sort Execution Result*

One example run took 610,800 ns. Insertion Sort processed and sorted all 112 unique alcohol-content values in ascending order. Over several runs we observed execution-time variation of 600,000–900,000 ns.

2B)

TIME COMPLEXITY FOR INSERTION SORT

```java
public static void insertionSort(float[] wineList) {  1 usage
    for (int i = 1; i < wineList.length; i++) {          // Outer loop: O(N)
        float temp = wineList[i];                        // O(1)
        int j = i - 1;                                   // O(1)

        while (j >= 0 && wineList[j] > temp) {           // Inner loop: O(N)
            wineList[j + 1] = wineList[j];               // O(1)
            j = j - 1;                                   // O(1)
        }
        wineList[j + 1] = temp;                          // O(1)
    }
    // O(N) * [O(1) + O(1) + O(N) * [O(1) + O(1)] + O(1)]
    // Removing all constants since we're only interested in big O which is usually worst case.
    // O(N) * [O(N)] => O(N) * O(N) = O(N^2) WorstCase
}
```

Code Snippet 1.7 *Insertion Sort* (GeeksforGeeks, 2025c)

CALCULATING THE TIME COMPLEXITY:

The time complexity of the Insertion Sort algorithm can be expressed as O(N) * [O(1) + O(1) + O(N) * [O(1) + O(1)] + O(1)]. We can simplify this algorithm by removing the constants like we did in the Bubble Sort which gives us O(N) * O(N) = O(N^2). This quadratic time complexity is attributed to the presence of a nested loop, which in this case an outer for loop and inner while loop.This shows that it's similar to the Bubble Sort which the algorithm's execution time increases quadratic as input size N grows and even when the array is shuffled the worst case stays the same.

BEST, AVERAGE AND WORST CASE TIME COMPLEXITY:

The worst case time complexity for Insertion Sort is O(N^2), which represents the maximum time complexity the algorithm can achieve. This is due to the necessity of comparing each element with all preceding elements in a worst case setting which is an array already sorted in descending order (GeeksforGeeks, 2025d).

Contrary to the best case time complexity of Insertion Sort is O(N). This occurs when the array is already sorted when we shuffle it, which prevents the execution of the inner while loop. In this scenario, only the outer for loop will iterate over the input size resulting in a linear time complexity (GeeksforGeeks, 2025d).

Furthermore shuffling the array can improve the performance by reducing the number of operations, but regardless of how close it approaches the best case, the average case which encompasses all cases between best and worst remains O(N^2) (GeeksforGeeks, 2025d).

Therefore the time complexity will not change on shuffling the array except for the edge case of shuffling creating an already sorted array which results in becoming best case time complexity with O(N) time complexity.

3A)

DIVIDE AND CONQUER

Before examining Merge Sort and Quick sort, it is essential to understand the divide and conquer paradigm that both algorithms employ. The divide-and-conquer design strategy involved dividing a problem instance into one or more smaller instances, solving each smaller instance recursively, and if necessary, combining the solutions to obtain the solution to the original instance (Neapolitan, 2015, s. 57-58, 64).

For sorting algorithms, the divide-and-conquer pattern applies (Goodrich et al., 2014, s. 532):

1. **Divide**: If the input has zero or one element, it is already sorted and returned immediately. Otherwise, split the input into two roughly equal halves.

2. **Conquer:** Recursively sort each half independently.

3. **Combine:** Merge the two sorted halves back together to produce the final sorted sequence.

MERGE SORT

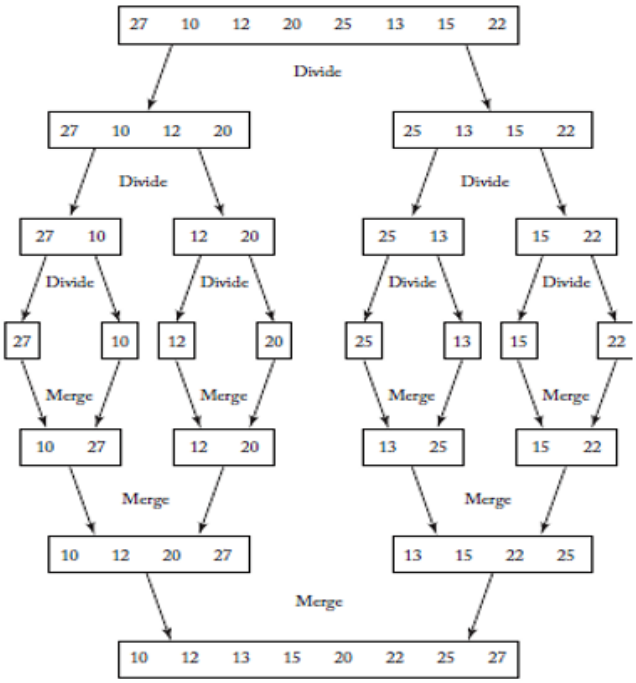


Figure 1.1 *The steps done by a human when sorting with Mergesort* (Neapolitan, 2015, s. 58).

As (Goodrich et al., 2014, s. 532) explains, Merge Sort applies the divide-and-conquer paradigm to sorting by dividing the input into halves, recursively sorting each half, and merging them back together.

Figure 1.1 demonstrates this process on the array [27, 10, 12, 20, 25, 13, 15, 22]. In the **divide phase**, the array is recursively split until each subarray contains a single element. During the **conquer phase**, these single-element arrays are already sorted. Finally, in the **combine phase**, sorted subarrays are merged pairwise, level by level, until the entire array is sorted to [10, 12, 13, 15, 20, 22, 25, 27].

Merge Sort Implementation

Our Merge Sort implementation follows the recursive divide-and-conquer pattern with two main methods: mergeSort() for division and merge() for combining.

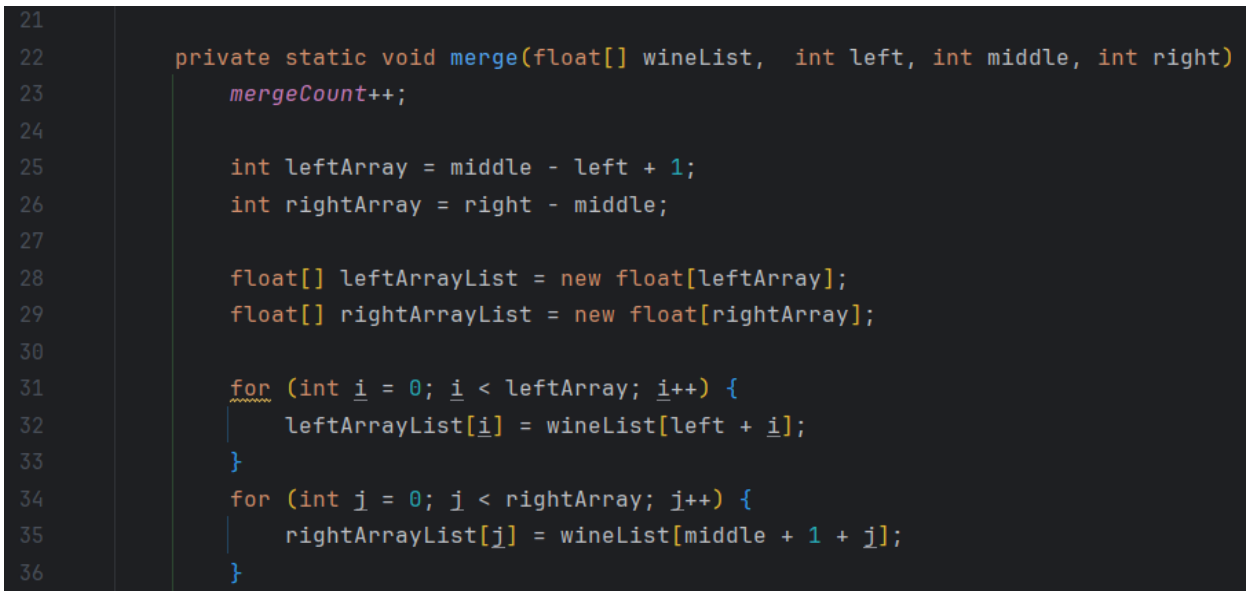
Key Components of the Implementation:

```
5   public class MergeSort {  7 usages  [redacted] *
6
7       private static int mergeCount = 0;  3 usages
8       private static int comparisonCount = 0;   3 usages
9
10      public static void mergeSort(float[] wineList, int left, int right) {  3 usages
11          if (left < right) {
12
13              int middle = left + (right - left) / 2;
14
15              mergeSort(wineList, left, middle);
16              mergeSort(wineList, middle + 1, right);
17
18              merge(wineList, left, middle, right);
19          }
20      }
21
```

Code Snippet 1.8 *Merge Sort First Part* (GeeksforGeeks, 2025k)

- **mergeSort() method (lines 10-17):** Recursively divides the array by calculating the middle point (line 11) and calling itself on the left half (line 13) and right half (line 14). After both halves are sorted, it calls merge() to combine them (line 16). The base case occurs when left >= right

.



```
21
22      private static void merge(float[] wineList,  int left, int middle, int right)
23          mergeCount++;
24
25          int leftArray = middle - left + 1;
26          int rightArray = right - middle;
27
28          float[] leftArrayList = new float[leftArray];
29          float[] rightArrayList = new float[rightArray];
30
31          for (int i = 0; i < leftArray; i++) {
32              leftArrayList[i] = wineList[left + i];
33          }
34          for (int j = 0; j < rightArray; j++) {
35              rightArrayList[j] = wineList[middle + 1 + j];
36          }
```

Code Snippet 1.81 *Merge Sort Second Part* (GeeksforGeeks, 2025k)

```
37
38          int i = 0;
39          int j = 0;
40          int k = left;
41
42          while (i < leftArray && j < rightArray) {
43              comparisonCount ++;
44              if (leftArrayList[i] <= rightArrayList[j]) {
45                  wineList[k] = leftArrayList[i];
46                  i++;
47              }else{
48                  wineList[k] = rightArrayList[j];
49                  j++;
50              }
51              k++;
52          }
53
54          while (i < leftArray) {
55              wineList[k] = leftArrayList[i];
56              i++;
57              k++;
58          }
59
60          while (j < rightArray) {
61              wineList[k] = rightArrayList[j];
62              j++;
63              k++;
64          }
```

Code Snippet 1.82 *Merge Sort Last Part* (GeeksforGeeks, 2025k)

- **merge() method (lines 22-63):** Combines two sorted subarrays into one. It creates temporary arrays for left and right halves (lines 27-28), copies data into them (lines 30-35), then merges by comparing elements and placing the smaller one first (lines 41-50). Remaining elements are copied at the end (lines 52-62).

```
Array to be sorted: [11.2, 0.0, 8.0, 10.133333, 11.94, 11.45, 11.7, 14.9, 8.5, 9.566667
Duration in nanoseconds: 1055300
MergeSort without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1, 9.2,
Amount of merges: 111
```

Code Snippet 1.9 *Merge Sort Execution Result*

As shown in our execution results, Merge Sort successfully sorted all 112 unique alcohol-content values in ascending order. One example run took 1,055,300 ns, with execution times across multiple runs ranging from approximately 700,000 to 1,400,000 ns.

3B)

AMOUNT OF MERGE OPERATIONS

```
Duration in nanoseconds: 155700
MergeSort already sorted array without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05,
Amount of merges: 111
Array to be sorted: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1, 9.2, 9.233334, 9.25,

Duration in nanoseconds: 227200
MergeSort already sorted array without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05,
Amount of merges: 111
```

Code Snippet 2.1 *Merge Sort Execution Results*

MERGE SORT TIME COMPLEXITY

The consistent nature of the merge amount is not affected by the shuffling of the array. This can be explained through an analysis of the time complexity associated with the Merge Sort algorithm. When the mergeSort function is called for the wineList array, it recursively divides the array into two halves until each subarray is reduced to a single element. This recursive division results in a logarithmic time complexity of O(log N). This comes from how many times we can divide the input size N with two until we reach one element (CodeAcademy Team, 2025).

Additionally the algorithm also uses a merge function which combines the single element into a sorted array of two elements. This merging process continues repeatedly, doubling the size of the sorted array on every step until it hits the original array size that it started as. During each merging step the elements in the array are handled once, resulting in a linear time complexity of O(N) based on the number of elements combined. Therefore the total time complexity of the Merge Sort algorithm is O(log n) * O(N) = O(N log N) (CodeAcademy Team, 2025).

BEST, AVERAGE AND WORST TIME COMPLEXITY

The consistency of the time complexity in worst, average and best case time complexity is O(N log N) which shows that the algorithm divides and merges the arrays regardless of the order of the elements as it lacks the mechanism to detect if sub arrays are sorted. Therefore the algorithm will have the same merge amount even if we shuffle the array (CodeAcademy Team, 2025).

Candidate number 37 and 39

4A)

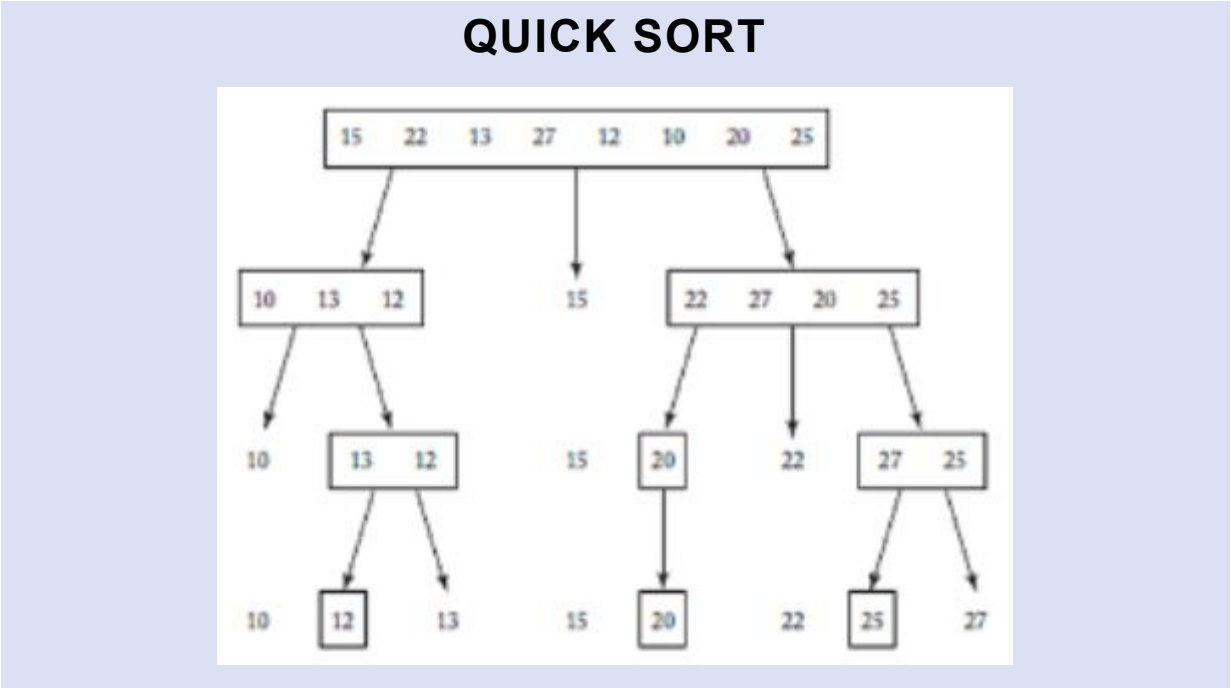
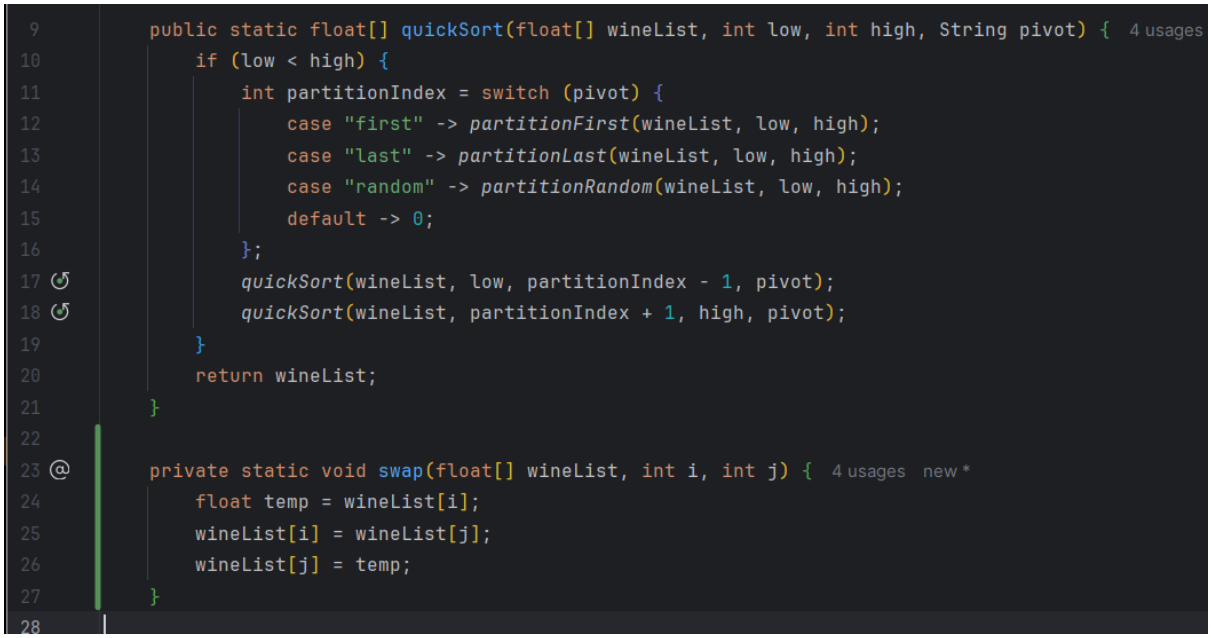


Figure 1.2 *The steps done by a human when sorting with Quick sort. The subarrays are enclosed in rectangles whereas the pivot points are free* (Neapolitan, 2015, s. 66).

Quick Sort also applies the divide-and-conquer paradigm, but uses it in a somewhat opposite manner to Merge Sort. All the partitioning work is done before the recursive calls, while the combine step requires no work as the array is sorted in-place (Goodrich et al., 2014, s. 544)

As (Neapolitan, 2015, s. 64–67) explains, Quick Sort partitions the array by placing all items smaller than a pivot element before it and all items larger after it. The pivot can be any item, and for convenience, the first element is commonly used. Figure 1.2 demonstrates this approach using the first element (15) as pivot, which results in [10, 13, 12] to its left and [22, 27, 20, 25] to its right.



```
9    public static float[] quickSort(float[] wineList, int low, int high, String pivot) {  4 usages
10       if (low < high) {
11           int partitionIndex = switch (pivot) {
12               case "first" -> partitionFirst(wineList, low, high);
13               case "last" -> partitionLast(wineList, low, high);
14               case "random" -> partitionRandom(wineList, low, high);
15               default -> 0;
16           };
17           quickSort(wineList, low, partitionIndex - 1, pivot);
18           quickSort(wineList, partitionIndex + 1, high, pivot);
19       }
20       return wineList;
21   }
22
23   private static void swap(float[] wineList, int i, int j) {  4 usages  new *
24       float temp = wineList[i];
25       wineList[i] = wineList[j];
26       wineList[j] = temp;
27   }
28
```

Code Snippet 2.2 *Quick Sort* (GeeksforGeeks, 2023, 2025d, 2025b)

Key Components of the Main implementation:

- **quickSort() method (lines 9-21):** The recursive method that controls the sorting process. It uses a switch statement (lines 11-16) to select the appropriate partition method based on the pivot strategy ("first", "last", or "random"). After partitioning returns the pivot's final position (line 11), the method recursively sorts the left subarray (line 17) and right subarray (line 18). The base case occurs when low >= high.
- **swap() method (lines 23-27):** A helper method that exchanges two elements in the array using a temporary variable. This method is used throughout the partition operations to rearrange elements around the pivot.

```
29   private static int partitionFirst(float[] wineList, int low, int high) {  1 usage
30       float pivot = wineList[low];
31       int i = low;
32       for (int j = low + 1; j <= high; j++) {
33           comparisonCount++;
34           if (wineList[j] < pivot) {
35               i++;
36               swap(wineList, i, j);
37           }
38       }
39       swap(wineList, low, i);
40       return i;
41   }
```

Code Snippet 2.21 *Quick Sort First Pivot* (GeeksforGeeks, 2025b)

partitionFirst() method (lines 29-41):

- Uses the first element as pivot (line 30)
- Iterates through the array from low + 1 to high (line 32), incrementing the comparison counter (line 33)
- When an element smaller than the pivot is found (line 34), it increments i (line 35) and swaps the element into position (line 36)
- After the loop, the pivot is swapped to its final position at index i (line 39), and this position is returned (line 40)

```
43 @    private static int partitionLast(float[] wineList, int low, int high) {  1 usage
44          float pivot = wineList[high];
45
46          int i = low - 1;
47
48          for (int j = low; j <= high - 1; j++) {
49              comparisonCount++;
50              if (wineList[j] < pivot) {
51                  i++;
52                  swap(wineList, i, j);
53              }
54          }
55          swap(wineList, i + 1, high);
56          return i + 1;
57      }
```

Code Snippet 2.22 *Quick Sort Last Pivot* (GeeksforGeeks, 2025d)

partitionLast() method (lines 43-57):

- Similar to partitionFirst(), but uses the last element as pivot (line 44)
- Initializes i to low - 1 (line 46) and iterates from low to high - 1 (line 48)
- Swaps elements smaller than pivot into position (lines 50-52)
- After the loop, swaps the pivot to its final position at i + 1 (line 55) and returns this index (line 56)

```
59      private static int partitionRandom(float[] wineList, int low, int high) {
60          randomElement(wineList, low, high);
61          float pivot = wineList[high];
62
63          int i = low - 1;
64          for (int j = low; j < high; j++) {
65              comparisonCount++;
66              if (wineList[j] < pivot) {
67                  i++;
68                  float temp = wineList[i];
69                  wineList[i] = wineList[j];
70                  wineList[j] = temp;
71              }
72          }
73          float temp = wineList[i + 1];
74          wineList[i + 1] = wineList[high];
75          wineList[high] = temp;
76          return i + 1;
77      }
78
79 @    static void randomElement(float[] wineList, int low, int high) {  1 usage
80          int pivot = rand.nextInt(high - low + 1) + low;
81
82          float temp = wineList[pivot];
83          wineList[pivot] = wineList[high];
84          wineList[high] = temp;
```

Code Snippet 2.23 *Quick Sort Random Pivot* (GeeksforGeeks, 2023)

partitionRandom() method (lines 59-77):

- Calls randomElement() helper method (line 60) to select and swap a random element to the high position.
- Then follows the same partitioning logic as partitionLast(): uses wineList[high] as pivot (line 61), initializes i to low - 1 (line 63), and iterates from low to high - 1 (line 64)
- Swaps elements smaller than pivot into position (lines 66-70) using inline swap instead of the swap() method
- After the loop, swaps the pivot to its final position at i + 1 (lines 73-75) and returns this index (line 76)

randomElement() helper method (lines 79-85):

- Generates a random index between low and high (line 80)
- Swaps the randomly selected element with the element at high position (lines 82-84), making it ready to be used as pivot

```
QuickSort First element without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9
Duration in nanoseconds: 939000

QuickSort Last element without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9
Duration in nanoseconds: 174800

QuickSort Random element without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05,
Duration in nanoseconds: 214600
```

Code Snippet 2.3 *Quick Sort First, Last and Random Pivot Execution Time Results*

As shown in our execution results, all three pivot strategies successfully sorted the alcohol content values in ascending order. The execution times varied across the different pivot selection methods:

- First element pivot: 939,000 ns
- Last element pivot: 174,600 ns
- Random element pivot: 214,600 ns

For this particular dataset, the last element pivot strategy achieved the fastest execution time, while the first element pivot was considerably slower. Across multiple runs, the random element pivot occasionally outperformed the last element strategy when a particularly well-positioned element was selected as pivot.
First element pivot : 800,000 – 1300,000 ns
Last element pivot: 150,000 – 350,000 ns
Random element pivot 120,000 – 450,000 ns

4B)

QUICK SORT TIME COMPLEXITY:

The number of comparisons in the Quick Sort algorithm is influenced by the choice of pivot, which is the main point of how the algorithm operates. Similar to the Merge Sort algorithm we can explain this with an analysis of the time complexity the Quick Sort has. Initially, the array is partitioned based on the selected pivot, which may be the first, last or a randomly chosen element in the array.

Partitioning involves rearranging the array so that all elements less than the pivot are positioned to the left side of the array and all elements greater than the pivot are positioned towards the right (Neapolitan, 2015, s. 66-67). It does this by comparing the two elements at each end of the array excluding the pivot element. If the right element is smaller and the right element is greater than the pivot, they switch places. But if only one of them is true, the element which does not fulfill the requirement gets ignored and the comparer moves to the next element towards the other comparer. When both the comparer cross each other that's when the partitioning stops (Interview Cake, 2025).

After rearranging the other elements the pivot will be placed in its correct position in the array. If the pivot was the first element, it will be swapped with the small element closest positioned to the large elements, while if the pivot is the last element, it is swapped with the large element that is closest is closest positioned to the small elements. In the case of a random pivot, it will swap itself with the last element in the array before partitioning and then follow the same steps as the last element pivot does. It is important to note that during all this, the array will not be sorting in ascending or descending order, but rather if the elements are smaller or greater than the pivot (Interview Cake, 2025).

This process divides the input array into two segments and the time complexity of O(N) can be employed to determine the number of comparisons made between the elements and the pivot during this process. After partitioning, the array is divided into two smaller subarrays based on the pivot as its dividing point. This process of partitioning and dividing into subarrays happens recursively until the subarrays are reduced to single element arrays. This type of recursive division is similar to the approach in Merge Sort, and they share a similar case of O(log N) in the dividing time complexity (Interview Cake, 2025).

BEST, AVERAGE AND WORST CASE TIME COMPLEXITY:

The total time complexity of Quick Sort would be O(N) * O(log N) = O(N log N). The O(N) complexity comes from the partitioning at each level of recursion where it will compare the elements with the pivot, while O(log N) represents the number of recursive calls. So O(N) happens N amount of time based on O(log N). The performance of the Quick Sort algorithm is based on the pivot its picking,

when the pivot is closer to the median of the array the more balanced are the subarrays divided on each side, which reduces the number of recursive calls. At the same time smaller subarrays means less times it has to compare in the partition process. Therefore the best and average case time complexity will be O(N log N) which is a log linear time complexity (Neapolitan, 2015, s. 69-71).

In contrast, the worst case time complexity occurs when the pivot is poorly chosen, an example is selecting the last element of an already sorted array. In this scenario the pivot is greater than all the other elements, resulting in an empty subarray on one side and a full subarray on the other. The time complexity will then change to O(N^2) as the partitioning process will repeatedly compare the last pivot with all the elements. It will then just remove one element at a time increasing the recursive calls (Neapolitan, 2015, s. 67-69).

```
QuickSort First element without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1, 9
Duration in nanoseconds: 978800
How many comparisons: 694

QuickSort Last element without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1, 9.
Duration in nanoseconds: 150100
How many comparisons: 639

QuickSort Random element without dupes: [0.0, 8.0, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.05, 9.1,
Duration in nanoseconds: 141100
How many comparisons: 704
```

Code Snippet 2.4 *Quick Sort First, Last And Random Comparison Results*

4B.A) IMPACT OF PIVOT STRATEGY ON COMPARISON COUNT

Code Snippet 2.4 results demonstrate that pivot selection strategy significantly impacts the number of comparisons required to sort our wine dataset:
- Last element pivot: 639 comparisons
- First element pivot: 694 comparisons
- Random element pivot: 692 comparisons (variation: 650-800 across runs)

The variation in comparison counts (from 639 to 694) confirms that the number of comparisons does change depending on pivot strategy. This occurs because different pivot choices create different partition balances, leading to varying recursion depths and thus different total comparison counts during the sorting process.

4B.B) OPTIMAL PIVOT STRATEGY FOR THIS DATASET

The last pivot strategy performs best for this dataset due to its pivot selection sequence. Examining the actual pivots (Code Snippets 2.4), the key difference lies in the initial selections:
- Last pivot: 11.05 → 10.3 → 9.05 → 8.0..
- First pivot: 11.2 → 11.05 → 10.3 → 9.05…
- Random pivot: 11.333333 → 10.033334 → 9.9...

The last pivot strategy quickly establishes balanced partitions with pivots well-distributed across the 8.0-14.2 range. Starting with 11.05, it rapidly moves to lower values (10.3, 9.05, 8.0), creating more balanced subarrays early in the recursion.

In contrast, the first pivot (11.2) and random pivot (11.333333) start higher, placing more elements in the left partition initially. This slightly unbalanced early partitioning accumulates additional comparisons throughout the recursion, resulting in 694 and 692 comparisons respectively.

Therefore the last pivot strategy becomes the “best” pivot strategy for this dataset.
However, for unknown datasets, the random pivot strategy remains the most robust choice.It guarantees O(N log N) expected time complexity regardless of input ordering and protects against worst-case O(N^2) scenarios that deterministic pivot selection can encounter with adversarial input patterns (GeeksforGeeks, 2025h; Tutorialspoint, 2025).

CONCLUSION

This assignment deepened our understanding of algorithm complexity through systematic implementation and analysis of four sorting algorithms: Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort. By complementing the required complexity analysis with execution time measurements, we observed how Big O notation provides a theoretical framework for scalability while practical performance reveals implementation-specific characteristics that theory abstracts away.

ANSWERS TO ASSIGNMENT QUESTIONS

Problem 1b & 2b - Does shuffling change time complexity?

As demonstrated in sections 1b and 2b, shuffling does not change the theoretical time complexity for Bubble Sort ($O(N^2)$) or Insertion Sort ($O(N^2)$) in the general case. The exception occurs only in edge cases: if shuffling produces an already-sorted array, optimized Bubble Sort achieves O(N) best-case complexity, and Insertion Sort similarly achieves O(N).

Problem 3b - Do merge operations change with shuffling?

No. As shown in section 3b, Merge Sort performs exactly 111 merge operations regardless of input ordering, because the algorithm's divide-and-conquer structure depends solely on input size (n=112), not element arrangement.

Problem 4b - Do comparisons change with pivot strategy?

Yes. Section 4b demonstrates that comparison counts vary significantly with pivot selection:
- Last element pivot: 639 comparisons
- First element pivot: 694 comparisons
- Random element pivot: 650-800 comparisons (varies across runs)

For this specific dataset (n=112), the last-pivot strategy performed best due to its initial pivot selections creating more balanced partitions early in the recursion.

Having addressed these specific questions, we now turn to the broader insight emerging from our analysis: the relationship between theoretical complexity and practical performance.

THEORY VS PRACTICE: BRIDGING THE GAP

Table 1.1 summarizes our empirical findings across all four sorting algorithms, showing theoretical complexity alongside measured comparisons and execution times on the primary dataset (n=112):

Algorithm	**Time Complexity (average)**	**Comparisons**	**Execution Time (ns)**
Bubble Sort (unoptimized)	$O(N^2)$	6216	400, 000 - 900, 000
Bubble Sort (optimized)	$O(N^2)$	6080	120,000 - 400,000
Insertion Sort	$O(N^2)$	2910	600,000 - 900,000
Merge Sort	O(N log N)	629 (111 merges)	700,000 - 1,400,000
Quick Sort (first pivot)	O(N log N)	694	800,000 - 1,300,000
Quick Sort (last pivot)	O(N log N)	639	150,000 - 350,000
Quick Sort (random pivot)	O(N log N)	600-900	120,000 - 450,000

Table 1.1 *Algorithm Performance Summary on Primary Dataset (n=112) Across Multiple Runs*

Our analysis demonstrates a critical distinction between time complexity and execution time. While Big O notation predicts algorithmic behavior as input size grows, execution time measurements reveal nuances that Big O analysis alone cannot capture.

Quick Sort with random pivoting sometimes performed more comparisons yet executed faster, demonstrating that factors like branch prediction, cache locality, and memory access patterns significantly impact performance beyond comparison counts. Similarly, optimized Bubble Sort outperformed expectations on small datasets (n=112) due to minimal overhead and excellent cache behavior, while Merge Sort's lower comparison count was offset by memory allocation and array copying operations (Ada Computer Science, 2025; GeeksforGeeks, 2025h).

As discussed in our introduction, execution time depends on factors such as basic operations, overhead instructions, and implementation details that Big O notation abstracts away (Neapolitan, 2015, s. 25). This explains why Merge Sort and Quick Sort, despite sharing O(N log N) complexity

and similar comparison counts (629 vs 639), exhibit different execution times, Quick Sort's superior constant factors and in-place sorting provide practical advantages (Thakrani, 2024).

KEY FINDINGS AND PRACTICAL IMPLICATIONS

Our analysis yields three insights for algorithm selection:

1. CONTEXT-DEPENDENT SELECTION:

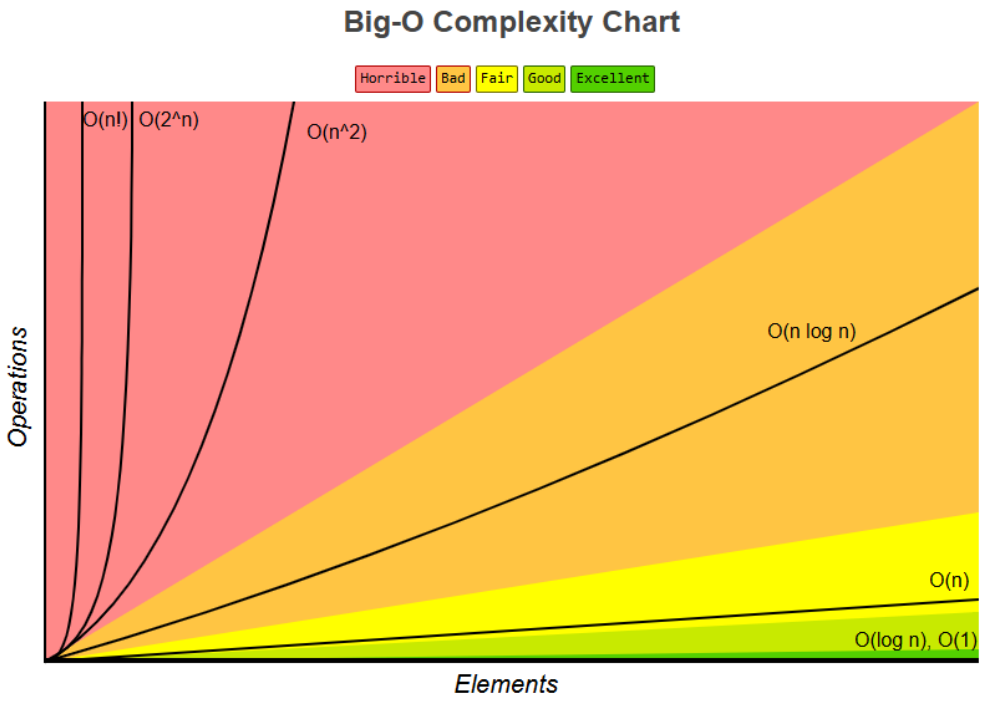


Figure 1.3: *Big-O Complexity Growth Rates,How operations scale with input size*(Rowell, 2025)

For small datasets (n < 200), O(N²) algorithms like Bubble Sort remain practical, achieving competitive execution times of 120,000-400,000 ns at n=112 (Table 1.1). At this scale, simple implementation and excellent cache behavior offset the theoretical complexity disadvantage (Amini, 2021).

However, for larger datasets (n > 1,000), the efficiency gap becomes dramatic. At n=6,498 (our complete dataset with duplicates), Quick Sort executes in 4.0 ms versus Bubble Sort's 29.7 ms, demonstrating how O(N log N) scaling becomes superior as input size grows (Figure 1.3).

```
Duration in nanoseconds: 29677100
BubbleSort optimized with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5, 8.5,
Amount of comparison: 21077877

QuickSort First element with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5, 8
Duration in nanoseconds: 4493100
Comparisons: 612254

QuickSort Last element with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5, 8.
Duration in nanoseconds: 4226500
Comparisons: 611070

QuickSort Random element with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5,
Duration in nanoseconds: 4029400
Comparisons: 602853
```

Code Snippet 2.5 *Bubble Sort And Quick Sort Comparison Results With Duplicates*

2. DATASET CHARACTERISTICS MATTER:

```
Duration in nanoseconds: 1284600
MergeSort with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5, 8.5, 8.5, 8.5,
Amount of merges: 6497
Comparisons: 74837

QuickSort First element with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5, 8
Duration in nanoseconds: 3745400
Comparisons: 612254

QuickSort Last element with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5, 8.
Duration in nanoseconds: 3650600
Comparisons: 611070

QuickSort Random element with dupes: [0.0, 8.0, 8.0, 8.4, 8.4, 8.4, 8.4, 8.4, 8.5, 8.5, 8.5, 8.5, 8.5,
Duration in nanoseconds: 4463100
Comparisons: 611852
```

Code Snippet 2.6 *Merge Sort And Quick Sort Comparison Results With Duplicates*

Even among algorithms with O(N log N) complexity the performance varies significantly. Quick Sort outperformed Merge Sort on small unique datasets with (n=112) despite having the same complexity (Table 1.1), but Merge Sort reversed this advantage on larger datasets where (n=6,498) (Code Snippet 2.6), achieving both faster execution and fewer comparisons. The duplicate values in the array caused unbalanced partitions in Quick Sort, leading to deeper recursions and more comparisons (WsCube Tech, 2025).

3. BIG O NOTATION IS NECESSARY BUT INSUFFICIENT:

Time complexity correctly predicted fundamental scaling behaviors (Figure 1.3), where O(N^2) algorithms degraded with growth while O(N log N) algorithms scaled efficiently with the input size. However Big O cannot predict practical performance for specific datasets and hardware configurations. Constant factors like implementation details, cache behavior and data characteristics determine the real world execution time which require both theoretical understanding and empirical validation for informed algorithm selection (Amini, 2021).

REFLECTION AND LEARNING OUTCOME

Our assignment reinforced the complementary relationship between theory and practice in algorithm analysis. Big O notation provides a hardware-independent framework for comparing algorithmic scalability and abstracting away implementation details to reveal fundamental growth rates of an algorithm. However, execution time reveals critical practical details that time complexity analysis cannot capture. Our testing demonstrated that algorithms with identical complexity classes showed vastly different performance due to cache behavior, memory overhead, and control flow complexity.

The “best” algorithm selection depends on the context of several values such as input size, data distribution, memory constraints, and performance requirements. For instance, while Merge Sort and Quick Sort share O(N log N) time complexity, Merge Sort requires O(N) auxiliary space for temporary arrays, whereas Quick Sort operates in-place with only O(log N) stack space for recursion (Silvwal, 2024).

This difference can be decisive in memory-constrained environments. By combining theoretical time complexity analysis with empirical execution time measurements, we gain a comprehensive understanding that extends beyond textbook descriptions. Big O notation predicts which algorithms scale to production workloads, while execution time reveals which perform best for specific use cases. This experience provided critical insight for making informed algorithm selection decisions in real world scenarios.

During our analysis, we identified that the dataset contains 112 unique alcohol values, including one zero value, rather than the expected 111. This does not impact our findings, as an additional element does not significantly alter algorithmic behavior.

REFERENCES

Ada Computer Science. (2025, november 4). *Sorting algorithms compared*. Ada Computer Science. https://adacomputerscience.org/concepts/sort_sorting_compared

Amini, P. (2021, august 4). 5 Factors to Consider Before Choosing a Sorting Algorithm. *Towards Data Science*. https://towardsdatascience.com/5-factors-to-consider-before-choosing-a-sorting-algorithm-5b079db7912c/

CodeAcademy Team. (2025, oktober 31). *Time Complexity of Merge Sort: A Detailed Analysis*. Codecademy. https://www.codecademy.com/article/time-complexity-of-merge-sort

GeeksforGeeks. (2023, september 14). *QuickSort using Random Pivoting*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/quicksort-using-random-pivoting/

GeeksforGeeks. (2025a, juli 23). *Bubble Sort Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/bubble-sort-algorithm/

GeeksforGeeks. (2025b, juli 23). *Implement Quicksort with first element as pivot*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/implement-quicksort-with-first-element-as-pivot/

GeeksforGeeks. (2025c, juli 23). *Insertion Sort Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/

GeeksforGeeks. (2025d, juli 23). *Quick Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/

GeeksforGeeks. (2025e, juli 23). *Time and Space Complexity Analysis of Bubble Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-analysis-of-bubble-sort/

GeeksforGeeks. (2025f, juli 23). *Time and Space Complexity of Insertion Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-insertion-sort-algorithm/

GeeksforGeeks. (2025g, juli 23). *Worst, Average and Best Case Analysis of Algorithms*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/worst-average-and-best-case-analysis-of-algorithms/

GeeksforGeeks. (2025h, august 11). *Which Sorting Algorithm is Best and Why?* GeeksforGeeks. https://www.geeksforgeeks.org/dsa/gfact-which-sorting-algorithm-is-best-and-why/

GeeksforGeeks. (2025i, august 28). *Understanding Time Complexity with Simple Examples*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/understanding-time-complexity-simple-examples/

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures & Algorithms* (Sixth Edition). Wiley.

Gupta, R. (2025, september 15). *Learning Outcome 5: Understanding Algorithm Efficiency* [Personlig kommunikasjon].

Interview Cake. (2025, november 4). *Quicksort Algorithm | Interview Cake*. Interview Cake: Programming Interview Questions and Tips. https://www.interviewcake.com/concept/java/quicksort

Neapolitan, R. E. (2015). *Foundation of Algorithms* (5. utg.). Jones & Bartlett Learning.

Programiz. (2025, oktober 28). *Bubble Sort*. https://www.programiz.com/dsa/bubble-sort

Rowell, E. (2025, november 11). *Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!)*. Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. https://www.bigocheatsheet.com/

Silvwal, S. (2024, desember 24). *Comparing Quick Sort, Merge Sort, and Insertion Sort*. https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/

Thakrani, S. (2024, juni 6). What are stable sorting algorithms and in-place sorting algorithms? *Medium*. https://medium.com/@suhailthakrani12/what-are-stable-sorting-algorithms-and-in-place-sorting-algorithms-672820a8e36c

Tutorialspoint. (2025, november 4). *Randomized Quick Sort Algorithm*. Randomized Quick Sort Algorithm. https://www.tutorialspoint.com/data_structures_algorithms/dsa_randomized_quick_sort_algorithm.htm

W3Schools. (2025, november 11). *DSA Time Complexity*. W3Schools. https://www.w3schools.com/dsa/dsa_timecomplexity_theory.php

WsCube Tech. (2025, februar 28). *Difference Between Quick Sort and Merge Sort (Comparison)*. Difference Between Quick Sort and Merge Sort (Comparison). https://www.wscubetech.com/resources/dsa/quick-sort-vs-merge-sort