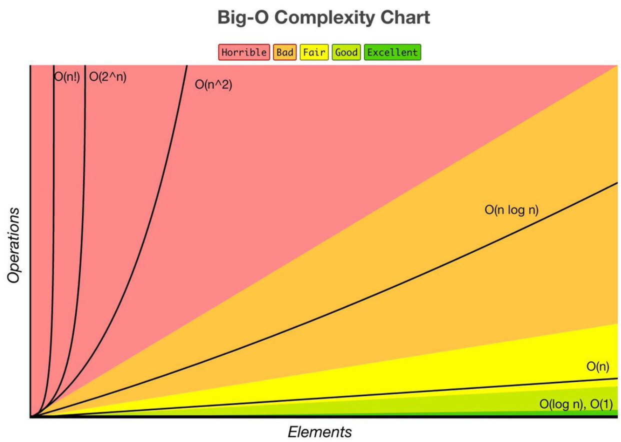
PROGRAMMING ASSIGNMENT 1

Subject: Algorithm Complexity Analysis

Course Instructors: Assoc. Prof. Dr. Erkut Erdem, Assoc. Prof. Dr. Hacer Yalım Keleş, Assoc. Prof. Dr. Adnan Özsoy
TAs: Sümeyye Meryem Taşyürek*, Hikmet Can Doğancı
Programming Language: Java (OpenJDK 11)
Due Date: **Friday, 21.03.2025 (23:59:59)**



1 Introduction

Analysis of algorithms is the area of computer science that provides tools to analyze the efficiency of different methods of solutions. Efficiency of an algorithm depends on these parameters; i) how much time, ii) memory space, iii) disk space it requires. Analysis of algorithms is mainly used to predict performance and compare algorithms that are developed for the same task. Also it provides guarantees for performance and helps to understand theoretical basis.

A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modeled input.
- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

In this experiment, you will analyze different sorting algorithms and compare their running times on a number of inputs with changing sizes.

2 Background and Problem Definition

Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be sorted. The efficiency of a sorting algorithm can be observed by applying it to sort datasets of varying sizes and other characteristics of the dataset instances that are to be sorted. In this assignment, you will be classifying the given sorting algorithms based on two criteria:

- **Computational (Time) Complexity**: Determining the best, worst and average case behavior in terms of the size of the dataset. Table 1 illustrates a comparison of computational complexity of some well-known sorting algorithms.

Table 1: Computational complexity comparison of some well-known algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$

- **Auxiliary Memory (Space) Complexity**: Some sorting algorithms are performed "in-place" using swapping. An in-place sort needs only $O(1)$ auxiliary memory apart from the memory used for the items being sorted. On the other hand, some algorithms may need $O(\log n)$ or $O(n)$ auxiliary memory for sorting operations. Table 2 illustrates an auxiliary space complexity comparison of the same well-known sorting algorithms.

Table 2: Auxiliary space complexity comparison of some well-known algorithms.

Algorithm	**Auxiliary Space Complexity**
Bubble Sort	$O(1)$
Insertion Sort	$O(1)$
Heap Sort	$O(1)$
Quick Sort	$O(\log n)$
Merge Sort	$O(n)$
Radix Sort	$O(k+n)$

A time complexity analysis focuses on gross differences in the efficiency of algorithms that are likely to dominate the overall cost of a solution. See the example given below:

Code	Unit Cost	Times
i=1;	c1	1
sum = 0;	c2	1
while (i ≤ n) {	c3	$n+1$
j=1;	c4	n
while (j ≤ n) {	c5	$n \cdot (n+1)$
sum = sum + i;	c6	$n \cdot n$
j = j + 1;	c7	$n \cdot n$
}		
i = i +1;	c8	n
}		

The total cost of the given algorithm is $c1 + c2 + (n+1) \cdot c3 + n \cdot c4 + n \cdot (n+1) \cdot c5 + n \cdot n \cdot c6 + n \cdot n \cdot c7 + n \cdot c8$. The running time required for this algorithm is proportional to n^2, which is determined as its growth rate, and it is usually denoted as $O(n^2)$.

3 Assignment Tasks

The main objective of this assignment is to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities. You are expected to implement the sorting algorithms given as pseudocodes, and perform a set of experiments on the given datasets to show that the empirical data follows the corresponding asymptotic growth functions. To do so, you will have to consider how to reduce the noise in your running time measurements, and plot the results to demonstrate and analyze the asymptotic complexities.

3.1 Sorting Algorithms to Implement

- **Comparison-based sorting algorithms**: In comparison-based sorting algorithms, the elements are compared to determine their order in the final sorted output. All comparison-based sorting algorithms have a complexity lower bound of $\Omega(n \log n)$. You will implement the following comparison-based sorting algorithms:
  - Comb sort given in Alg. 1
  - Insertion sort given in Alg. 2
  - Shaker sort given in Alg. 3
  - Shell sort given in Alg. 4
- **Non-comparison-based sorting algorithms**: Some algorithms sort data without direct element comparisons, often leveraging structure or predefined assumptions. Non-comparison-based sorting algorithms may perform in $O(n)$ time complexity. You will implement the following non-comparison-based sorting algorithm:
  - Radix sort given in Alg. 5

Algorithm 1 Comb Sort

```
procedure COMBSORT(A: array)
    gap ← length(A)
    shrink ← 1.3
    sorted ← false
    while sorted = false do
        gap ← max(1, floor(gap / shrink))
        sorted ← (gap == 1)
        for i ← 0 to length(A) - gap do
            if A[i] > A[i+gap] then
                swap(A[i], A[i+gap])
                sorted ← false
            end if
        end for
    end while
end procedure
```

Algorithm 2 Insertion Sort

```
procedure INSERTION-SORT(A: array)
    for j← 1, . . . , length(A) do
        key ← A[j]
        i ← j - 1
        while i >= 0 and A[i] > key do
            A[i + 1] ← A[i]
            i ← i - 1
        end while
        A[i + 1] ← key
    end for
end procedure
```

Algorithm 3 Shaker Sort

```
procedure SHAKERSORT(A: array)
    swapped ← true
    while swapped do
        swapped ← false
        for i ← 0 to length(A)-2 do
            if A[i] > A[i+1] then
                swap(A[i], A[i+1])
                swapped ← true
            end if
        end for
        if not swapped then
            break
        end if
        swapped ← false
        for i ← length(A)-2 to 0 do
            if A[i] > A[i+1] then
                swap(A[i], A[i+1])
                swapped ← true
            end if
        end for
    end while
end procedure
```

Algorithm 4 Shell Sort

```
procedure SHELLSORT(A: array)
    n ← length(A)
    gap ← n / 2
    while gap > 0 do
        for i ← gap to n-1 do
            temp ← A[i]
            j ← i
            while j ≥ gap and A[j - gap] > temp do
                A[j] ← A[j - gap]
                j ← j - gap
            end while
            A[j] ← temp
        end for
        gap ← gap / 2
    end while
end procedure
```

Algorithm 5 Radix Sort

```
procedure RADIXSORT(A: array, d: number of digits)
    for pos ← 1 to d do                                  ▷ Process each digit position
        A ← COUNTINGSORT(A, pos)
    end for
    return A
end procedure
procedure COUNTINGSORT(A: array, pos: digit position)
    count ← array of 10 zeros                            ▷ Assuming decimal digits (0-9)
    output ← array of the same length as A
    size ← length(A)
    for i ← 1, . . . , size do
        digit ← getDigit(A[i], pos)
        count[digit] ← count[digit] + 1
    end for
    for i ← 2, . . . , 10 do
        count[i] ← count[i] + count[i − 1]
    end for
    for i ← size, . . . , 1 do
        digit ← getDigit(A[i], pos)
        count[digit] ← count[digit] −1
        output[count[digit]] ← A[i]
    end for
    return output
end procedure
```

3.2 Dataset

You will test the given sorting algorithms on a shortened version of a real dataset that contains a great amount of recorded traffic flows of a test network, generated from a real network trace through FlowMeter. This dataset (TrafficFlowDataset.csv) includes more than 250,000 captures of communication packets sent in a bidirectional manner between senders and receivers over a certain period of time.

In order to be able to perform a comparative analysis of the performance of the given sorting algorithms over different data sizes, you will consider several smaller partitions of the dataset, that is, its subsets of sizes 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, and 250000 starting from the beginning of the file. You will be sorting the records based on the **Flow Duration** feature given in the 3^{rd} column (**Flow Duration**), which is of type *int* (see Fig. 1).

Flow ID	Timestamp	Flow Duration
192.168.1.101-67.212.184.66-2156-80-6	13.06.2010 06:01	2328040
192.168.1.101-67.212.184.66-2159-80-6	13.06.2010 06:01	2328006
192.168.2.106-192.168.2.113-3709-139-6	13.06.2010 06:01	7917
192.168.5.122-64.12.90.98-59707-25-6	13.06.2010 06:01	113992
192.168.5.122-64.12.90.98-59707-25-6	13.06.2010 06:01	3120
192.168.5.122-64.12.90.66-37678-25-6	13.06.2010 06:01	121910
192.168.5.122-64.12.90.66-37678-25-6	13.06.2010 06:01	4073
192.168.5.122-64.12.90.97-56782-25-6	13.06.2010 06:01	128308
192.168.5.122-64.12.90.97-56782-25-6	13.06.2010 06:01	2449
192.168.5.122-205.188.59.193-54493-25-6	13.06.2010 06:01	110814
192.168.5.122-205.188.59.193-54493-25-6	13.06.2010 06:01	2391
192.168.5.122-205.188.155.110-59130-25-6	13.06.2010 06:01	178255
192.168.5.122-205.188.155.110-59130-25-6	13.06.2010 06:01	2955
192.168.1.101-67.212.184.66-2159-80-6	13.06.2010 06:01	9624620

Figure 1: Sorting the data in the Flow Duration column only.

3.3 Experiments and Analysis Tasks

Assignment Steps Summarized:

- Implement the given algorithms in Java.
- Perform the experiments on the given dataset.
  - Tests with varying input sizes.
  - Tests on random, sorted, and reversely sorted inputs.
- Fill out the results tables and plot the results.
- Discuss the findings.

Once you have implemented the given algorithms in Java, you need to perform a set of experiments, report, illustrate, and analyze the results, and discuss your findings.

3.3.1 Experiments with Sorting Algorithms

In the given set of experiments, you will run the given sorting algorithms on different input types and sizes.

Experiments on the Given Random Data In the first set of experiments, you will test the algorithms on the given random datasets with varying input sizes. You are expected to measure the **average** running time of each algorithm **by running each experiment 10 times and taking the average of the recorded running times** for each input size. Make sure to save the sorted input data for the next set of experiments. Please be careful that you measure the running time of the sorting process only. To obtain the input numbers for each test case of size n, take the first n rows of the given dataset.

Report your findings in milliseconds (ms) by filling out the first five empty rows in Table 3.

Table 3: Results of the running time tests performed for varying input sizes (in ms).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Comb sort										
Insertion sort										
Shaker sort										
Shell sort										
Radix sort										
	Sorted Input Data Timing Results in ms									
Comb sort										
Insertion sort										
Shaker sort										
Shell sort										
Radix sort										
	Reversely Sorted Input Data Timing Results in ms									
Comb sort										
Insertion sort										
Shaker sort										
Shell sort										
Radix sort										

Experiments on the Sorted Data In this second set of experiments, you should run your algorithms all over again, but this time on the already sorted input data that you obtained in the previous experiments. **The same averaging rule over 10 tries should also be applied in this step**. Fill out the next five empty rows in Table 3 with the new measured running times for the sorted input data.

Experiments on the Reversely Sorted Data In this third set of experiments, you should run your algorithms on the reversely sorted input data. You should use the already sorted input data that you obtained in the previous experiments and reverse them first (or simply read the sorted array from the end). Please make sure that **you measure the running time of the sorting process only**. **The same averaging rule over 10 tries should also be applied in this step**. Fill out the final five empty rows in Table 3 with the new measured running times for the reversely sorted input data.

3.3.2 Complexity Analysis and Result Plotting

After completing all the tests, you should analyze the obtained results in terms of the computational and auxiliary space complexity of the given algorithms. First, complete Tables 4 and 5, and justify your answers in short. Note that we are not interested in the overall space complexity of these algorithms, only in the additional memory space they use while performing the sorting operations. Please state which lines from the given pseudo-codes you used to obtain the auxiliary space complexity answers.

Table 4: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Comb sort	$\Omega()$	$\Theta()$	$O()$
Insertion sort	$\Omega()$	$\Theta()$	$O()$
Shaker sort	$\Omega()$	$\Theta()$	$O()$
Shell sort	$\Omega()$	$\Theta()$	$O()$
Radix sort	$\Omega()$	$\Theta()$	$O()$

Table 5: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Comb sort	$O()$
Insertion sort	$O()$
Shaker sort	$O()$
Shell sort	$O()$
Radix sort	$O()$

Plot the results obtained from the experiments in 3.3.1. **You should obtain separate plots for each sorting algorithm, showing performance across random, sorted, and reversely sorted data.** Table 3 should result in three plots, comparing all sorting algorithms with each-other under each input condition. For all plots, X-axis should represent the input size (the number of input instances n), while Y-axis should represent the running time in ms. See Figure 2 to see an example of how you should demonstrate your results.

The sample chart includes two different plotted algorithms, and illustrate how the performance of a faster algorithm on average can degrade significantly in some worst case scenarios. Your plots must include the results of all given algorithms presented in a similar manner.

The plotting operation must be handled programmatically by using a readily-made *Java* libraries. You can leverage any library of your choice. You can also use the **XChart** library, which is open-source and pretty easy to use. To use the **XChart** library, you should first obtain the *.zip* file by using the download button from the following link `https://knowm.org/open-source/xchart/`. Then, you should extract the file and add **xchart-3.8.1.jar** to your project; i.e., include it in your classpath. You can check the example code provided by the authors from the link: `https://knowm.org/open-source/xchart/xchart-example-code/`.

3.3.3 Results Analysis and Discussion

Briefly discuss the obtained results by referring to Table 4 and the obtained plots in 3.3.2. Answer the following questions:

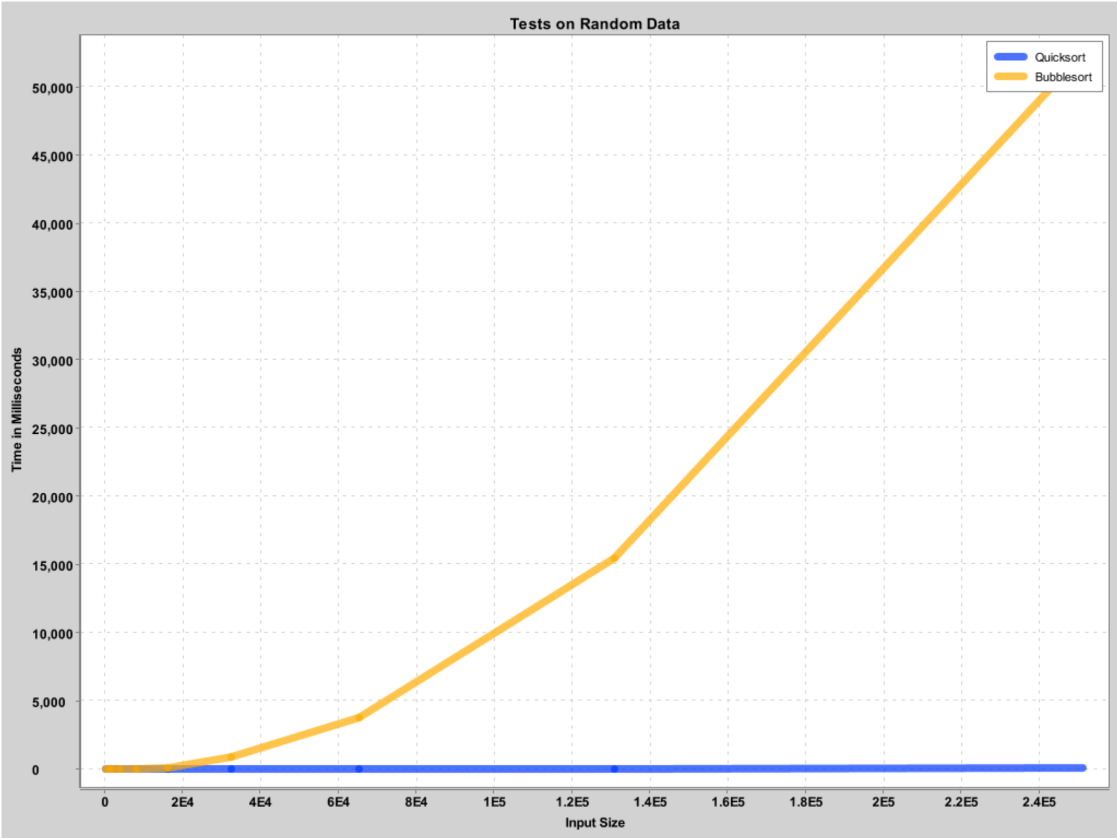


Figure 2: A sample plot showing running time results for varying input sizes on random input data for two sample sorting algorithms.

- What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted?
- Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?
- Do Shaker Sort and Comb Sort improve performance compared to Bubble Sort? If so, how do their approaches contribute to this improvement?
- Is Shell Sort perform better than Insertion Sort for larger datasets? Under what conditions does Insertion Sort still perform well?
- Given that Radix Sort is a non-comparison-based sorting algorithm, how does it handle large numerical ranges efficiently?

Grading Policy

- Implementation of the algorithms: 20%
- Performing the experiments and reporting the results (filling out the given results tables) and making comments: 25%

- Completing the given two computational and auxiliary space complexity tables and making comments: 15%
- Plotting the results: 20%
- Results analysis and discussion: 20%

What to Include in the Report

You are encouraged to use this **Programming Assignment Report Template** and create your reports in LaTeX. We suggest using Overleaf platform for this. This is not mandatory, but make sure your report has all necessary parts and information (click here to see the PDF example).

Your report needs to include the following:

1. Include a brief problem statement.
2. Include your Java codes corresponding to the given sorting algorithms.
3. Include all running time results tables corresponding to all given experiment sets performed on the given random, sorted, and reversely sorted data, for varying input sizes. All five algorithms must be tested.
4. Include two completed tables that show the theoretical computational and auxiliary space complexities of the given algorithms, with a brief justification of your answers.
5. Include at least four plots of the obtained results from step 3.
6. Briefly discuss the obtained results by answering the given questions.
7. In preparing the report, you may use any theoretical resources, online or otherwise, but make sure to include the references in your report. **Do not copy any ready codes from the internet as there is a big chance someone else could do the same thing, and you would get caught for cheating even if you don't know each other!**

Important Notes

- Do not miss the deadline: **Friday, 21.03.2025 (23:59:59)**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (`https://piazza.com/hacettepe.edu.tr/spring2025/bbm204`), and you are supposed to be aware of everything discussed on Piazza.
- You will submit your work via `https://submit.cs.hacettepe.edu.tr/` with the file hierarchy given below:

```
b<studentID>.zip
├─ src.zip <FILE>
│   ├─ Main.java <FILE>
│   └─ *.java <FILE>
└─ report.pdf <FILE>
```

- The name of the main class that contains the main method should be **Main.java**. You may use **this starter code** which has a helpful example of using **XChart** library. The main class and all other classes should be placed directly (no subfolders) into a zip file named **src.zip**.
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for online/AI sources. Make use of them responsibly, refrain from generating the code you are asked to implement. Remember that we also have have access to such tools, making it easier to detect such cases.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in suspension of the involved students.

References

[1] "Sorting algorithm." Wikipedia, https://en.wikipedia.org/wiki/Sorting_algorithm, Last Accessed: 10/02/2022.

[2] N. Faujdar and S. P. Ghrera, "Analysis and Testing of Sorting Algorithms on a Standard Dataset," 2015 Fifth International Conference on Communication Systems and Network Technologies, 2015, pp. 962-967.

[3] G. Batista, "Big O," Towards Data Science, Nov 5. 2018, https://towardsdatascience.com/big-o-d13a8b1068c8, Last Accessed: 10/03/2024.