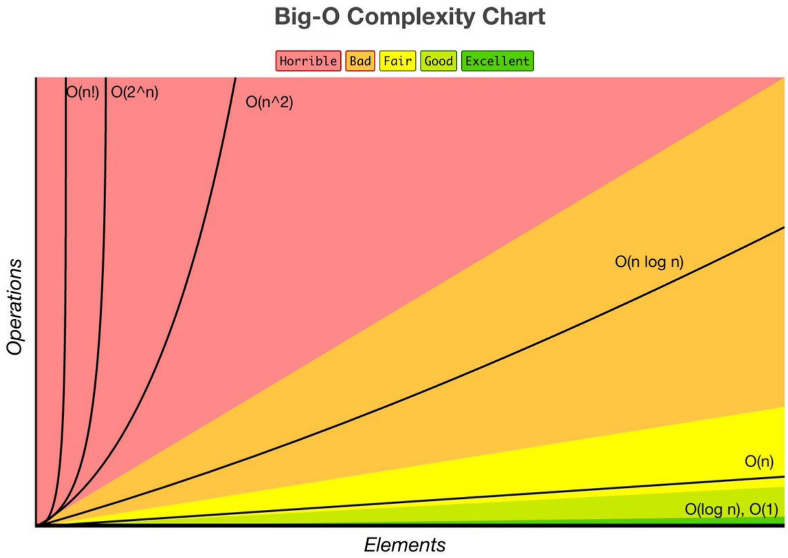
PROGRAMMING ASSIGNMENT 1

Subject: Algorithm Complexity Analysis

Course Instructors: Assoc. Prof. Dr. Erkut Erdem, Prof. Dr. Suat Özdemir, Asst. Prof. Dr. Adnan Özsoy
TAs: Alperen Çakın, Selma Dilek
Programming Language: Java 1.8.0
Due Date: **Wednesday, 09.03.2022 (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 sorting algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion 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 sorting algorithms.

Algorithm	**Auxiliary Space Complexity**
Selection Sort	$O(1)$
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 $\leq$ n) {	c3	$n+1$
j=1;	c4	n
while (j $\leq$ 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 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

You are given four different sorting algorithms to implement in this assignment. The sorting should be implemented in ascending order.

- **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 two comparison based sorting algorithms:
  - Insertion sort given in Alg. 1
  - Merge sort given in Alg. 2
- **Non-comparison based sorting algorithms**: There are sorting algorithms that can run faster than $O(n \log n)$ time complexity, but they require special assumptions about the input sequence to determine the sorted order of the elements. Non-comparison based sorting algorithms use operations other than comparisons to determine the sorted order

and may perform in $O(n)$ time complexity. You will implement the following two non-comparison based sorting algorithms:

- Pigeonhole sort given in Alg. 3
- Counting sort given in Alg. 4

The pseudocodes of the given sorting algorithms are given below.

Algorithm 1 Insertion Sort

```
procedure INSERTION-SORT(A: array)
    for j← 2, . . . , 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 2 Merge Sort

```
procedure MERGESORT(A: array)
    n ← length(A)
    if n ≤ 1 then
        return A
    end if
    left ← A[1]...A[n/2]
    right ← A[n/2+1]...A[n]
    left ← MERGESORT(left)
    right ← MERGESORT(right)
    return MERGE(left, right)
end procedure
procedure MERGE(A: array, B: array)
    C: array
    while A and B have elements do
        if A[1] > B[1] then
            add B[1] to the end of C
            remove B[1] from B
        else
            add A[1] to the end of C
            remove A[1] from A
        end if
    end while
    while A has elements do
        add A[1] to the end of C
        remove A[1] from A
    end while
    while B has elements do
        add B[1] to the end of C
        remove B[1] from B
    end while
    return C
end procedure
```

Algorithm 3 Pigeonhole Sort

```
procedure PIGEONHOLESORT(A: array)
    n ← length(A)
    min ← minimum(A)
    max ← maximum(A)
    range ← max − min + 1
    output ← [ ]
    holes: array of arrays ← [[ ], [ ], . . ., [ ]] (total of range elements)
    for $i \leftarrow 1, \dots, n$ do
        holes[A[i] - min].add(A[i])
    end for
    for i ← 1, . . . , range do
        output.addAll(holes[i])
    end for
    return output
end procedure
```

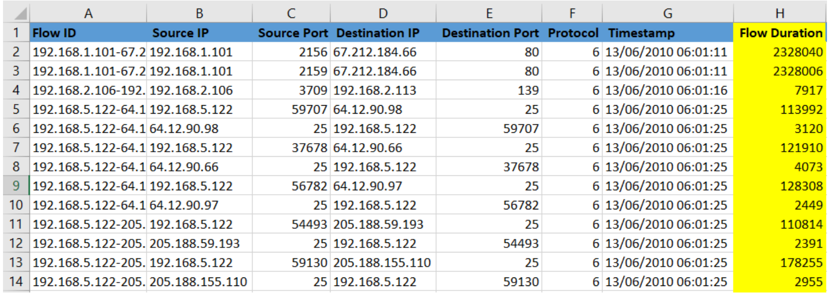
Algorithm 4 Counting Sort

```
procedure COUNTINGSORT(A, k)
    count ← array of k + 1 zeros
    output ← array of the same length as A
    size ← length(A)
    for i ← 1, . . . , size do
        j ← key(input[i])
        count[j] ← count[j] + 1
    end for
    for i ← 2, . . . , k + 1 do
        count[i] ← count[i] + count[i − 1]
    end for
    for i ← $size, \dots,$ 1 do
        j ← key(input[i])
        count[j] ← count[j] −1
        output[count[j]] ← input[i]
    end for
    return output
end procedure
```

3.2 Dataset

You will test the given sorting algorithms on 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 (e.g., header, payload, etc.) sent in a bidirectional manner between senders and receivers over a certain period of time. FlowMeter generates more than 80 record features including Flow ID, source and destionation IPs, etc. As it is out of the scope of this assignment, you do not need to know the details about the type of information recorded in these features, so do not get confused by them.

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 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and 131072 starting from the beginning of the file, as well as the whole dataset of 251281 records. You will be sorting the records based on the **Flow Duration** feature given in the 8^{th} column H, which is of type *int* (see Fig. 1).



	A	B	C	D	E	F	G	H
1	Flow ID	Source IP	Source Port	Destination IP	Destination Port	Protocol	Timestamp	Flow Duration
2	192.168.1.101-67.2	192.168.1.101	2156	67.212.184.66	80	6	13/06/2010 06:01:11	2328040
3	192.168.1.101-67.2	192.168.1.101	2159	67.212.184.66	80	6	13/06/2010 06:01:11	2328006
4	192.168.2.106-192.	192.168.2.106	3709	192.168.2.113	139	6	13/06/2010 06:01:16	7917
5	192.168.5.122-64.1	192.168.5.122	59707	64.12.90.98	25	6	13/06/2010 06:01:25	113992
6	192.168.5.122-64.1	64.12.90.98	25	192.168.5.122	59707	6	13/06/2010 06:01:25	3120
7	192.168.5.122-64.1	192.168.5.122	37678	64.12.90.66	25	6	13/06/2010 06:01:25	121910
8	192.168.5.122-64.1	64.12.90.66	25	192.168.5.122	37678	6	13/06/2010 06:01:25	4073
9	192.168.5.122-64.1	192.168.5.122	56782	64.12.90.97	25	6	13/06/2010 06:01:25	128308
10	192.168.5.122-64.1	64.12.90.97	25	192.168.5.122	56782	6	13/06/2010 06:01:25	2449
11	192.168.5.122-205.	192.168.5.122	54493	205.188.59.193	25	6	13/06/2010 06:01:25	110814
12	192.168.5.122-205.	205.188.59.193	25	192.168.5.122	54493	6	13/06/2010 06:01:25	2391
13	192.168.5.122-205.	192.168.5.122	59130	205.188.155.110	25	6	13/06/2010 06:01:25	178255
14	192.168.5.122-205.	205.188.155.110	25	192.168.5.122	59130	6	13/06/2010 06:01:25	2955

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

3.3 Experiments and Analysis Tasks

Assignment Steps Summarized:

- Implement the given sorting algorithms in Java.
- Perform the experiments on the given datasets.
  - 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 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, and measure the average running time of each algorithm by running each experiment 10 times and taking the average of 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. Report your findings in milliseconds (ms) in Table 3.

Table 3: Results of the running time tests performed on the random data of varying sizes (in ms).

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort										
Merge sort										
Pigeonhole sort										
Counting sort										

3.3.2 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 from Section 3.3.1 should also be applied in this step. Fill out a similar table as Table 3 with the new measured running times.

3.3.3 Experiments on the Reversely Sorted Data

In this final 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 from Section 3.3.1 should also be applied in this step. Fill out a similar table as Table 3 with the new measured running times.

3.3.4 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 pseudocodes 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**
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Pigeonhole Sort			
Counting Sort			

Table 5: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion Sort	$O(1)$
Merge Sort	$O(n)$
Pigeonhole Sort	
Counting Sort	

Plot the results obtained from the experiments in 3.3.1, 3.3.2, and 3.3.3. You should obtain three separate plots for each set of experiments (random, sorted, and reversely sorted data), each of which should include the results for all four given algorithms. X-axis should represent the input size (the number of input instances n), while Y-axis should represent the running time in ms. See Figures 2, 3, and 4 to see an example of how you should demonstrate your results.

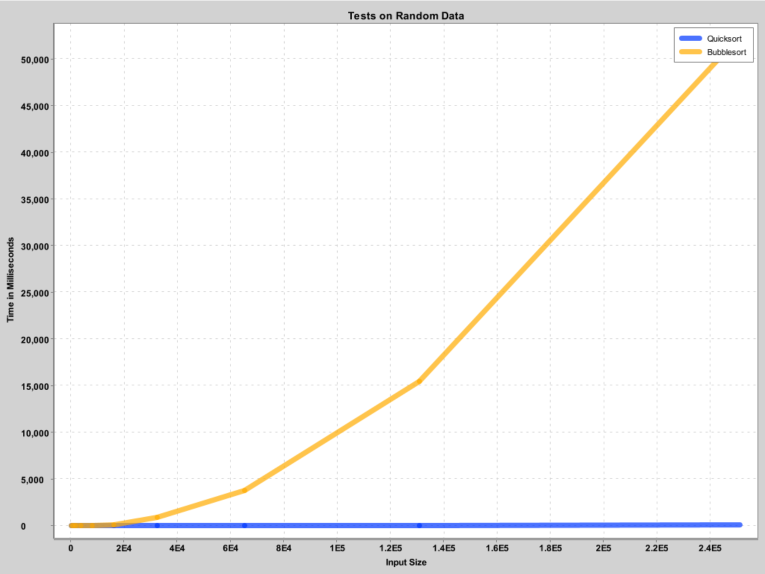


Figure 2: Running time results for varying input sizes on random input data.

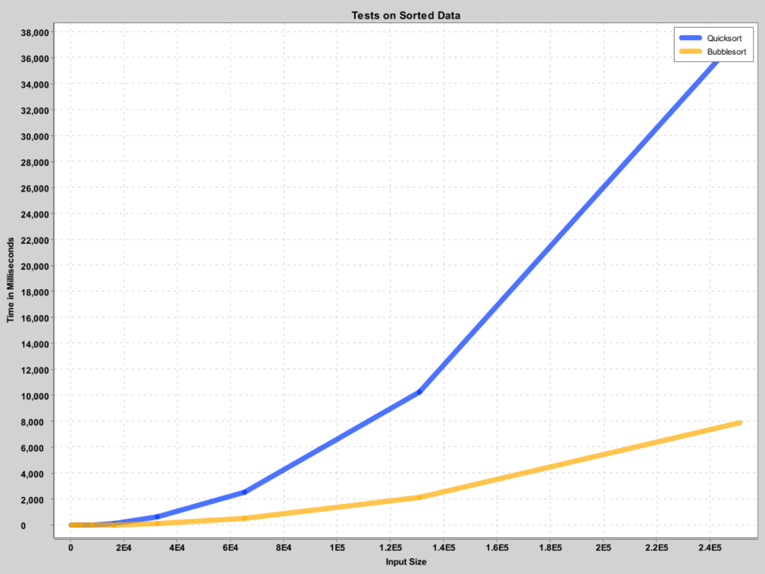


Figure 3: Running time results for varying input sizes on sorted input data.

The sample charts include 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 the four given algorithms in a similar manner.

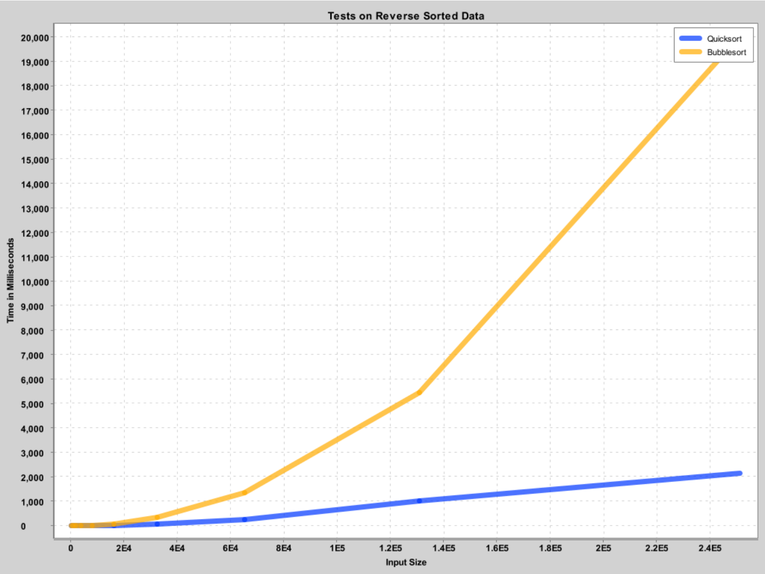


Figure 4: Running time results for varying input sizes on reverse sorted input data.

The plotting operation must be handled programmatically by using a readily-made *Java* library. You are encouraged to 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/.

3.3.5 Results Analysis and Discussion

Briefly discuss the obtained results by referring to Table 4 and the obtained plots in 3.3.4. Answer the following questions:

- 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?

Grading Policy

- Submission: 1%
- Implementation of the algorithms: 20%
- Performing the experiments and reporting the results (three tables): 30%
- Completing the computational and auxiliary space complexity tables: 9%
- Plotting the results (three plots): 30%
- Results analysis and discussion: 10%

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.

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 three running time results tables corresponding to the three experiment sets performed on the given random, sorted, and reversely sorted data, for varying input sizes. All four 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 three plots of the obtained results from step 3.
6. Briefly discuss the obtained results by answering the given questions.
7. You may use any resources, online or otherwise, but make sure to include the references in your report.

Important Notes

- Do not miss the deadline: **Wednesday, 09.03.2022 (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/spring2022/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 <DIR>
    - 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 in **src** directory. Feel free to create subdirectories if necessary, but they should also be in **src** directory.
- 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 the material found on the web as everything on the web has been written by someone else.

> **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: 11/02/2022.