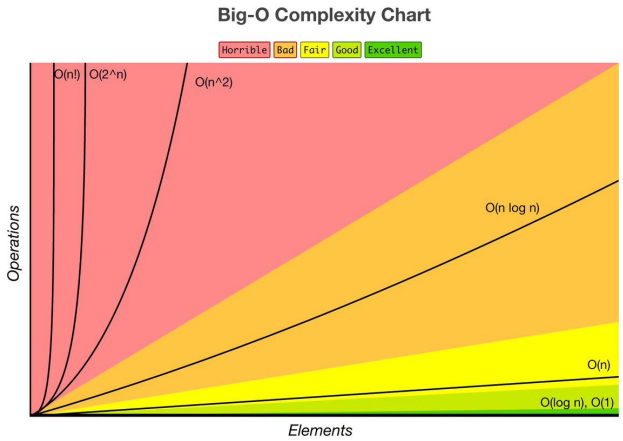
PROGRAMMING ASSIGNMENT 1

Subject: Algorithm Complexity Analysis

Course Instructors: Assoc. Prof. Dr. Erkut Erdem, Prof. Dr. Suat Özdemir, Asst. Prof. Dr. Adnan Özsoy
TAs: Ali Burak Erdoğan, Alperen Çakın, Dr. Selma Dilek
Programming Language: Java (OpenJDK 11)
Due Date: **Thursday, 30.03.2023 (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 and searching 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. Furthermore, modern computing and the internet have made accessible a vast amount of information. The ability to efficiently search through this information is fundamental to computation. 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 and searching 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 and searching algorithms.

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

Algorithm	**Best Case**	**Average Case**	**Worst Case**
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	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

- **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. Searching algorithms usually do not require additional memory space. Table 2 illustrates an auxiliary space complexity comparison of the same well-known sorting and searching 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	$O(k+n)$
Linear Search	$O(1)$
Binary Search	$O(1)$

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 three 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. You will implement the following two comparison based sorting algorithms:
  - Selection sort given in Alg. 1
  - Quick sort (iterative implementation) 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 non-comparison based sorting algorithm:
  - Bucket sort given in Alg. 3

Algorithm 1 Selection Sort

```
1: procedure SELECTION-SORT(list: array of items, n: size of list)
2:     for i from 1 to n - 1 do
3:         min ← i
4:         for j from i+1 to n do
5:             if list[j] < list[min] then
6:                 min ← j
7:             end if
8:         end for
9:         if min ≠ i then
10:            swap list[min] and list[i]
11:        end if
12:    end for
13: end procedure
```

Algorithm 2 Quick Sort (iterative implementation)

```
1: procedure QUICK-SORT(array : list of sortable items, low : first element of list, high : last element of list)
2:     stackSize ← high - low + 1
3:     stack ← [ ] of length stackSize
4:     top ← -1
5:     stack[++top] ← low
6:     stack[++top] ← high
7:     while top ≥ 0 do
8:         high ← stack[top--]
9:         low ← stack[top--]
10:        pivot ← partition(arr, low, high)
11:        if pivot - 1 > low then
12:            stack[++top] ← low
13:            stack[++top] ← pivot - 1
14:        end if
15:        if pivot + 1 < high then
16:            stack[++top] ← pivot + 1
17:            stack[++top] ← high
18:        end if
19:    end while
20: end procedure
21: procedure PARTITION(array : list of sortable items, low : first element of list, high : last element of list)
22:    pivot ← arr[high]
23:    i ← low - 1
24:    for j from low to high do
25:        if arr[j] ≤ pivot then
26:            i ← i + 1
27:            swap arr[i] with arr[j]
28:        end if
29:    end for
30:    swap arr[i+1] and arr[high]
31:    return i + 1
32: end procedure
```

Algorithm 3 Bucket Sort

```
1: procedure BUCKET-SORT(A: array, n)
2:     numberOfBuckets ← √len(A)
3:     buckets ← an array of empty arrays with the length numberOfBuckets
4:     max ← max(A)
5:     for each i in A do
6:         Append i to buckets[hash(i, max, numberOfBuckets)]
7:     end for
8:     Sort each bucket individually
9:     sortedArray ← [ ]
10:    for each bucket in buckets do
11:        Add all elements of the bucket to sortedArray
12:    end for
13:    return sortedArray
14: end procedure
15: procedure HASH(i, max, numberOfBuckets)
16:    return ⌊i/max × (numberOfBuckets − 1)⌋
17: end procedure
```

3.2 Search Algorithms to Implement

You are given two different search algorithms to implement in this assignment.

- Linear Search given in Alg. 4
- Binary Search given in Alg. 5

Algorithm 4 Linear Search

```
1: procedure LINEAR-SEARCH(A: array to search in, x: value to be searched)
2:     size ← len(A)
3:     for each i ← 0 to size - 1 do
4:         if A[i] == x then
5:             return i
6:         end if
7:     end for
8:     return -1
9: end procedure
```

Algorithm 5 Binary Search

```
1: procedure BINARY-SEARCH(A: sorted array, x: value to be searched)
2:     low ← 0
3:     high ← len(A - 1)
4:     while high - low > 1 do
5:         mid ← (high + low) / 2
6:         if A[mid] < x then
7:             low ← mid + 1
8:         else
9:             high ← mid
10:        end if
11:    end while
12:    if A[low] == x then
13:        return low
14:    else if A[high] == x then
15:        return high
16:    end if
17:    return -1
18: end procedure
```

3.3 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 (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 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, and 250000 starting from the beginning of the file. You will be sorting and searching in the records based on the **Flow Duration** feature given in the 7^{th} column (**Flow Duration**), which is of type *int* (see Fig. 1).

Flow ID	Source IP	Source Port	Destination IP	Destination Port	Timestamp	Flow Duration
192.168.1.101-67.212.184.66-2156-80-6	192.168.1.101	2156	67.212.184.66	80	13.06.2010 06:01	2328040
192.168.1.101-67.212.184.66-2159-80-6	192.168.1.101	2159	67.212.184.66	80	13.06.2010 06:01	2328006
192.168.2.106-192.168.2.113-3709-139-6	192.168.2.106	3709	192.168.2.113	139	13.06.2010 06:01	7917
192.168.5.122-64.12.90.98-59707-25-6	192.168.5.122	59707	64.12.90.98	25	13.06.2010 06:01	113992
192.168.5.122-64.12.90.98-59707-25-6	64.12.90.98	25	192.168.5.122	59707	13.06.2010 06:01	3120
192.168.5.122-64.12.90.66-37678-25-6	192.168.5.122	37678	64.12.90.66	25	13.06.2010 06:01	121910
192.168.5.122-64.12.90.66-37678-25-6	64.12.90.66	25	192.168.5.122	37678	13.06.2010 06:01	4073
192.168.5.122-64.12.90.97-56782-25-6	192.168.5.122	56782	64.12.90.97	25	13.06.2010 06:01	128308
192.168.5.122-64.12.90.97-56782-25-6	64.12.90.97	25	192.168.5.122	56782	13.06.2010 06:01	2449
192.168.5.122-205.188.59.193-54493-25-6	192.168.5.122	54493	205.188.59.193	25	13.06.2010 06:01	110814
192.168.5.122-205.188.59.193-54493-25-6	205.188.59.193	25	192.168.5.122	54493	13.06.2010 06:01	2391
192.168.5.122-205.188.155.110-59130-25-6	192.168.5.122	59130	205.188.155.110	25	13.06.2010 06:01	178255
192.168.5.122-205.188.155.110-59130-25-6	205.188.155.110	25	192.168.5.122	59130	13.06.2010 06:01	2955
192.168.1.101-67.212.184.66-2159-80-6	192.168.1.101	2159	67.212.184.66	80	13.06.2010 06:01	9624620
192.168.1.101-67.212.184.66-2156-80-6	192.168.1.101	2156	67.212.184.66	80	13.06.2010 06:01	9624649

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

3.4 Experiments and Analysis Tasks

Assignment Steps Summarized:

- Implement the given 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.

3.4.1 Experiments with Sorting Algorithms

In this 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 three 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									
Selection sort										
Quick sort										
Bucket sort										
	Sorted Input Data Timing Results in ms									
Selection sort										
Quick sort										
Bucket sort										
	Reversely Sorted Input Data Timing Results in ms									
Selection sort										
Quick sort										
Bucket 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 three 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 three empty rows in Table 3 with the new measured running times for the reversely sorted input data.

3.4.2 Experiments with Searching Algorithms

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

Experiments on the Given Random Data In the set of experiments, you will test the linear search algorithm 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 1000 times and taking the average of the recorded running times** for each input size. **Use nanosecond precision instead of milliseconds**.

Fill out the first row of Table 4 with the new measured running times.

Table 4: Results of the running time tests of search algorithms of varying sizes (in ns).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)										
Linear search (sorted data)										
Binary search (sorted data)										

Experiments on the Sorted Data In the final set of experiments, you will test both linear and binary search algorithms on the already sorted input data that you obtained in the previous experiments (do not sort the data again, you should already have saved the sorted data). Please make sure that **you measure the running time of the searching process only**. **The same averaging rule over 1000 tries given above should also be applied in this step**.

To perform a searching experiment (e.g., linear search on random data on an array with 500 elements), pick a random number from the array (from the 500 selected elements) and search for it. Repeat this 1000 times and calculate the average timing **in nanoseconds**.

Fill out the remaining rows of Table 4 with the new measured running times.

3.4.3 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 5 and 6, 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/searching operations. Please state which lines from the given pseudocodes you used to obtain the auxiliary space complexity answers.

Plot the results obtained from the experiments in 3.4.1 and 3.4.2. **You should obtain four separate plots for each set of experiments**, each of which should include the results for all given algorithms noted in the result tables. Table 3 should result in three plots, while the results shown in Table 4 should be plotted as a single chart. If you think that some results should be plotted separately for better illustration, you are allowed to create extra plots. For

Table 5: Computational complexity comparison of the given algorithms.

Algorithm	Best Case	Average Case	Worst Case
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Bucket Sort			
Linear Search			
Binary Search			

Table 6: Auxiliary space complexity of the given algorithms.

Algorithm	Auxiliary Space Complexity
Selection Sort	$O(1)$
Quick Sort	$O(n)$
Bucket Sort	
Linear Search	
Binary Search	

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.

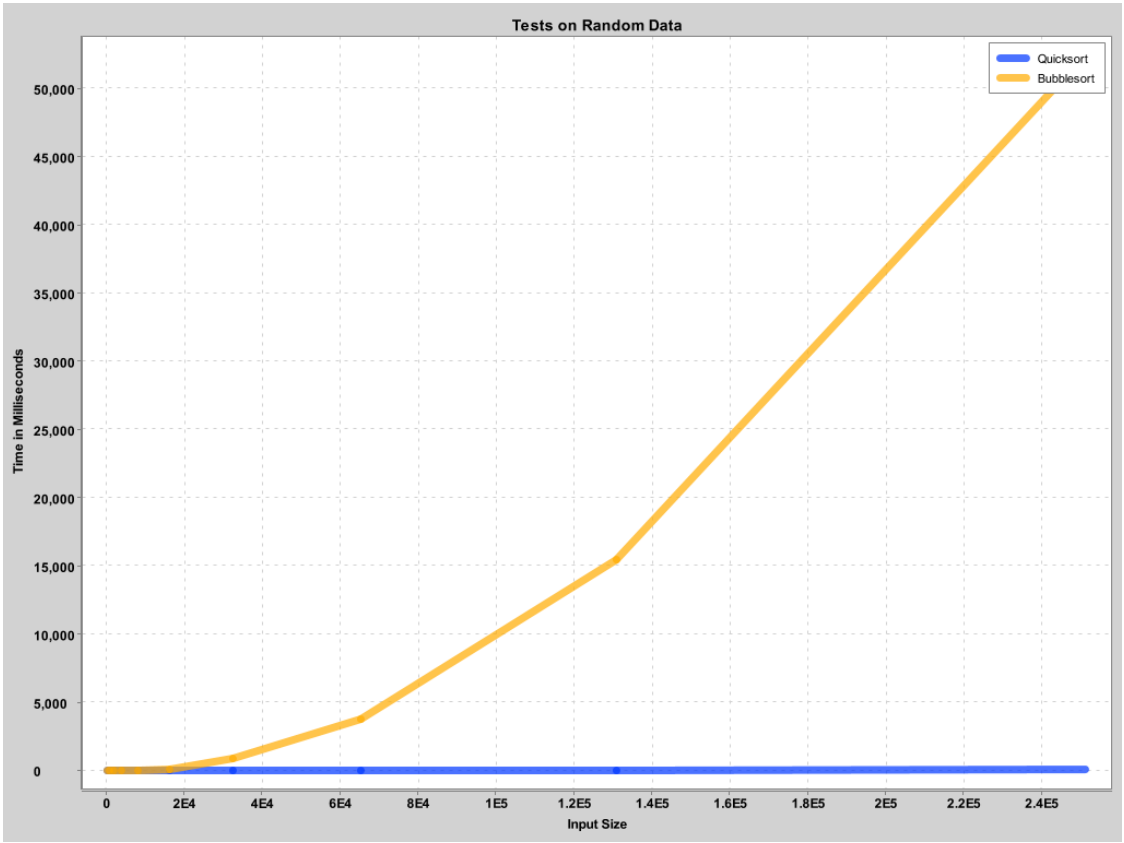


Figure 2: A sample plot showing running time results for varying input sizes on random input data for two sample sorting algorithms.

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* 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/.

3.4.4 Results Analysis and Discussion

Briefly discuss the obtained results by referring to Table 5 and the obtained plots in 3.4.3. 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/searched?
- 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 (filling out the given two results tables): 30%
- Completing the given two computational and auxiliary space complexity tables: 9%
- Plotting the results (at least four 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 (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 and search 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. 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: **Thursday, 30.03.2023 (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/spring2023/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 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: 10/03/2023.