Estimating the energy consumption of Java Programs.

Collections & Sorting algorithms

Authors: Mustafa Alsaid & Dalya AbuHemeida
Supervisor: Mauro Caporuscio
Semester: VT 2023
Subject: Computer Science

Abstract

Java applications consume energy, which has become a controversial topic since it limits the number of machines and increases the cost of data centers. This paper investigates the potential relationship between energy consumption and some quality attributes for Java Collections and Sorting algorithms in order to raise awareness about using energy-efficient programs. In addition, introduce to the developers the most and least efficient Java Collection and Sorting algorithm in terms of energy consumption, memory, and CPU usage. This was achieved by conducting a controlled experiment to measure these terms. The data obtained for the results was used to acquire Statistical and Efficiency Analysis to answer the research questions.

Keywords: Java Collections, Sorting algorithms, Energy consumption, Quality attributes.

Preface

We would like to express our sincere gratitude to our supervisor, Mauro Caporuscio, at Linnaeus University. Throughout the entire duration of this project, Mauro has consistently provided invaluable and constructive feedback, greatly contributing to its improvement. We are also indebted to Daniel Toll, at Linnaeus University, for organizing insightful workshops throughout the course of this project. His suggestions and inspiration have played a significant role in shaping the outcome.

Additionally, we extend a special thanks to Mustafa’s dear friend, Dr. Zaid Hattab, in the field of Statistics at the University of Galway in Ireland. Zaid's expertise and assistance in statistical concepts and utilizing R have been instrumental in enhancing the statistical analysis in this thesis.

We are grateful to all of them for their unwavering support, guidance, and dedication, which have been crucial to the successful completion of this research study.

Contents

1 Introduction 6
1.1 Background 6
1.2 Related work 7
1.3 Problem formulation 8
1.4 Motivation 9
1.5 Results 9
1.6 Scope/Limitation 10
1.7 Target group 10
1.8 Outline 11
2 Method 12
2.1 Research Project 12
2.2 Research Methods 12
2.3 Reliability and Validity 15
2.4 Ethical Considerations 15
3 Theoretical Background 16
3.1 Time and Space Complexity 16
3.2 Consumption Metrics 17
3.3 Java Collection Interface 17
3.4 Sortings Algorithms 19
3.5 Statistical Terms 21
4 Research project – Implementation 23
4.1 Tools and Techniques 23
4.2 Data Overview 24
4.2.1 Data Collection 24
4.2.2 Data Analysis 25
5 Results 26
6 Analysis 32
6.1 Statistical Analysis 32
6.1.1 Collections' Linear Regression Analysis 33
6.1.2 Sortings' Linear Regression Analysis 34
6.2 Efficiency Analysis 36
6.2.1 Collections Experiment Analysis 37
6.2.2 Sortings Experiments Analysis 41
7 Discussion and Conclusions 43

7.1 Future Work 45
References **47**
Appendix 1 **51**
Appendix 2 **51**

1 Introduction

Software engineering (SE) is concerned with designing, developing, testing, and maintaining software. Earlier, non-object-oriented programming (procedural) languages were used in software development, such as C, but after that, Object-Oriented Programming (OOP) was invented in the software engineering field, and several languages were created that support OOP. Java is one of the most famous languages and is widely used in OOP [1].

As software engineering gains popularity, and new areas like mobile devices and cloud computing experience significant growth, the need to minimize software energy consumption has become a crucial non-functional requirement for modern software systems [2]. Candy Pang highlights that rising energy costs and increased expenses for cooling data centers have imposed restrictions on the number of machines that can be operated [2].

This research paper is a 15 HEC bachelor thesis in Computer Science at Linnaeus University, studying the estimated energy consumption for Java Collections and Sorting algorithms, which are common and widely used by Java software developers. By conducting practical experiments to measure the energy consumption of these programs and finding relationships between the energy usage and some dynamic quality attributes. This thesis will provide a knowledge base that aims to be a unique resource for discovering the best Collections and Sortings to be selected based on energy consumption, memory, and CPU usage.

1.1 Background

Although energy consumption has received more attention in software development recently, there is still insufficient support for creating energy-efficient programs due to multiple reasons, such as the absence of abstractions and tools that allow for the examination of relevant properties affecting energy usage [3]. Moreover, programmers have limited software energy efficiency awareness and lack knowledge about reducing the energy consumption of software [2].

Java applications consume energy during execution, ranking 5th out of 27 programming languages that have been analyzed in terms of energy consumption efficiency by Pereira et al in 2017 [4]. In particular, the Java Collections framework provides a wide variety of data structures and interfaces.The Collection interface is the parent of all other interfaces within the Collections framework, and Java provides an implementation for its sub-interfaces such as List, Queue, Deque, and more, which are widely used by software engineers throughout the development phase. These Collections provide the same functionalities, such as Add, Remove, and Contains but with different time complexity depending on the specific collection [5]. Time complexity measures the amount of runtime the device needs for an algorithm. It represents the period required, which increases depending on the data input size, and uses a common expression known as big O notation [6]. In relation to Sorting algorithms, the time complexity is a significant factor in analyzing the performance and determining the efficiency of handling an extensive amount of data. Sorting algorithms are widely used in databases, data analysis, and search algorithms. It provides the best way to sort data in a certain order with a given list of elements [6].

In Java, two known built-in Sorting algorithms can be applied, a sort method of the Array class or a sort method of the Collections class; however, Sorting also can be done using Java loops [7]. Merge sort, Heap sort, Selection sort, and Quick sort are among the most commonly used Sorting algorithms in Java. They are often referred to as classic Sorting algorithms due to their popularity and historical significance in computer science [8].

Sorting algorithms, in addition to Java Collections, consume energy. This energy consumption of an application varies depending on several factors such as CPU, memory usage, and more dynamic quality attributes [9]. Dynamic attributes are the attributes that reflect the behavior of the program during its execution, such as memory usage [10]. However, finding an accurate relationship between energy consumption and quality aspects of algorithms is a challenge, and achieving the most precise performance depends on the application's requirements.

1.2 Related work

This section provides a description of the related works for this research area, which is focused on two groups of work: previous studies involving energy consumption of Collections and Sorting algorithms and other studies focusing on how to analyze energy consumption.

As recently as 2016, Hasan et al. [11] compared the energy consumption of Collections in Java. The authors analyzed various Collections and then built a profile of energy consumption for all Collections, which were analyzed earlier. The purpose of their work is to find which implementation of each collection (Lists, Sets, and Maps) consumes less energy and to use the outcome to manually improve the efficiency of a set of selected applications. Pinto et al. [12] examined the thread-safe Java Collections on two distinctive desktop machines. They realized that the energy consumption for an operation shifts broadly among different implementations of the same collection. For instance, using a newer version of hash-table can save 17% of consumed energy in real-world-benchmarks. Another study done by Saborido et al. [13] compared two Android-specific collection implementations of Maps, SparseArray, and ArrayMap. The research found out that HashMap is more efficient regarding the energy efficiency of apps than ArrayMap, while SparseArray is preferable over HashMap and ArrayMap when keys are primitive. Oliveira et al. discovered the impact of software construction on energy consumption and offered a recommendation tool that can assist developers in reducing energy usage [14].

Further work done by Alves et al. [3] discusses the lack of support and focus for creating energy-efficient software programs, and performs experiments that consist of gathering, modeling, and analyzing energy data, and presenting their findings. The experiments compare the energy consumption of Java implementations of specific Sorting algorithms "Insertion Sort, Bubble Sort, and Selection Sort" using various data sets, presenting how to combine energy measurement and analysis tools for this purpose. The implementation focused on sorting values stored in a Vector, ArrayList, and LinkedList. As a result, the total consumption using ArrayList for Bubble and Selection Sortings was at its least compared to the other data structure, nevertheless, Selection Sort consumed the least amount of energy when applying the three structures. However, this study utilizes only the classics of Sorting algorithms in Java, not considering the dynamic quality attributes in relation to energy consumption.

1.3 Problem formulation

As discussed earlier, several research studies have been conducted to explore the energy consumption in software development pertaining to Collections and Sorting algorithms [3][11][12][13][14]. However, a noticeable research gap exists in terms of investigating the relationship between energy consumption and dynamic quality attributes, specifically CPU and memory, when utilizing Java Collections with operations such as Add, Contains, and Remove, as well as Sorting algorithms. This gap indicates that Java developers often rely on a singular performance metric without considering other closely associated factors. Thus, this thesis aims to address this challenge and provide a coherent answer to the following research questions:

- **RQ1**. What is the relationship between energy consumption and quality attributes (CPU/Memory) usage in Java Collections in the case of Add, Contains, and Remove operations?
  The purpose of RQ1 is to investigate the relationship between energy consumption and CPU/Memory usage in Java Collections. Specifically, this research question seeks to determine whether there is a positive or inverse correlation between energy consumption and these dynamic quality attributes.

- **RQ2.** What is the relationship between energy consumption and the quality attributes (CPU / Memory) usage when implementing Sorting algorithms in Java?
  This question aims to investigate the relationship between energy consumption and CPU/Memory usage in Java Sorting algorithms. The relationship in question refers to various factors, such as the algorithm used to implement the Sorting and its time complexity. This research question seeks to determine whether there is a positive or

inverse correlation between energy consumption and these dynamic quality attributes.

- **RQ3**. Regarding the results of RQ1 & RQ2, how strong are these relationships?
  The purpose of this question is to determine the strength of the relationship if it exists.

- **RQ4.** What is the most efficient Java Collection/s among the three operations in terms of energy consumption considering the performance?
  This question aims to provide insight into the most efficient ways to use Java Collections in terms of the operations Add, Contains, and Remove.

- **RQ5.** What is the most efficient Java Sorting algorithm in terms of energy consumption considering the performance?
  This question aims to provide insight into the most efficient ways to use Sorting algorithms in Java. The result can be valuable for developers, as it can help optimize the application's performance, making it more efficient and effective.

Through the systematic exploration of the aforementioned research questions, complemented by the execution of multiple experiments that generate quantitative data, and subsequent analysis of the obtained results, this study endeavors to establish accurate and generalized findings pertaining to Collections and Sorting algorithms. Additionally, identifying the optimal approaches for employing Collection(s) and Sorting algorithm(s) in terms of energy consumption, while concurrently considering performance aspects.

1.4 Motivation

This thesis aims to advance the field of computer science. The results will address two areas, software developments using Java and software performance. From an industrial perspective, the results will assist Java software engineers to develop systems that consider energy consumption as a part of that system's performance. Thus, building systems that consume less energy. Since a high-performance application with low energy consumption, memory, and CPU usage is the outcome of this research, lower energy costs and less pollution will be achieved, which is a societal motivation. Moreover, providing a well-performed application attracts more users, which benefits the company economically.

1.5 Results

As the purpose of this project is to experimentally measure the overall energy consumption of Collections and Sorting algorithms used in Java applications, the specifics of what makes one algorithm perform better than another can be determined. The performance is evaluated according to different quality attributes and their consumption, following the guidelines by previous works [3][11], three metrics are identified as essential for evaluating any examined method:

1. **Memory:** The total storage of an operation during the execution in Megabytes.
2. **CPU:** The percentage of used processors during the execution.
3. **Energy:** The capacity used during the execution in Joule.

Energy consumption will be measured using the tool JoularJx. While the chosen dynamic quality attributes (Memory - CPU) will be measured using built-in classes in Java. Analyzing these data statistically using R to investigate the relationship between variables is facilitated with the help of the RStudio tool. This analysis aims to identify the most energy-efficient Collections and Sorting algorithms. Moreover, 3D scatter plots will be generated using R for energy consumption, CPU, and memory usage. The objective of these 3D scatter plots is to provide a better vision and clearer understanding of which Collection/s and Sorting algorithm/s are more efficient.

1.6 Scope/Limitation

This thesis focuses on the energy consumption efficiency of Java Collections and Sorting algorithms, hence the energy consumption efficiency for other algorithms is not included. Different limitations constrain this project. One of these limitations is the number of elements that are used in the operations of the algorithms' implementations. Each operation is limited to 400,000 elements as an input, and the number of elements is decided as a reasonable number due to the time constraints required to conduct each experiment. In addition, the number of experiments is limited to 100 experiments for each Sorting algorithm, Collections, and the operations of collection: Add, Contains, and Remove. Resulting in 3 700 experiments in total.

Generally, a broad range of Sorting algorithms in computer science exists, yet, the study is focused on ten known algorithms, and the operations used on Collections are limited to three operations. Lastly, resource constraints limit the ability to conduct the experiments on multiple PCs and across different operating systems.

1.7 Target group

This paper primarily studies the energy consumption of Java Collections and Sorting algorithms, as the main goal is to increase Java software performance regarding energy consumption, and it is assumed that the target group has at most casual knowledge of Java applications. Hoping this paper provides a solid analysis to be used by professional Java developers and architects.

1.8 Outline

This paper has been divided into several chapters. Chapter 1 provides the background, related work, scope/limitation, motivation, and results. Chapter 2 discusses the methodological framework and research methods. In Chapter 3, the relevant theoretical background concepts are described, and the knowledge gap and relevant challenges are provided. Chapter 4 discusses the implementation details of the conducted experiments and other activities needed to realize the controlled experiment. Chapter 5 provides the results of those experiments. Chapter 6 provides an analysis of the results. Chapter 7 discusses the results and analysis made in previous chapters. Additionally, It explores this study's conclusion and provides several potential future works.

2 Method

This chapter describes the research methodology approach used to answer the research questions presented in the previous section. The implemented method is discussed in detail, highlighting its strengths, limitations, and relevance to the research objectives. Moreover, the section addresses the tools required to conduct the methodology, as well as its reliability, validity, and ethical considerations.

2.1 Research Project

To effectively address the research questions RQ1 & RQ2, it is crucial to carefully consider the appropriate implementation approach, data needed to be collected, and the type of analysis to be conducted which enables a comprehensive exploration and understanding of the intended relationship. Thus, after conducting a thorough review of available methodological approaches, the controlled experiments methodology emerges as the most suitable choice for achieving the purpose of these two research questions. Compared to alternative methods, controlled experiments offer a rigorous and controlled environment for evaluating the interplay between variables and determining the potential impact of a specific set of variables on a targeted outcome. Moreover, meticulous attention is given to ensuring the reliability and validity of the obtained results, minimizing the influence of random factors, and promoting a robust and credible analysis [15].

2.2 Research Methods

The controlled experiment is a technique used in scientific research to minimize the effects of extraneous variables. It also strengthens the ability of the independent variable to change the dependent variable [15]. A controlled experiment allows us to eliminate various confounding variables or uncertainty in the project. It allows us to control the specific variables that may influence the result. In addition, it provides a standard to which the result is compared, and it allows the researcher to correct errors [16]. This methodology (controlled experiments) aims to determine the response to the formulated problem by testing the performance of Java Collections and Sorting algorithms in terms of CPU and memory, as well as their energy efficiency, in a controlled environment.

The process comprises a series of steps aimed at collecting reliable quantitative data about the performance and energy consumption efficiency of Java Collections and Sorting algorithms using a scientific approach:

1. Identify the device and work environment.
2. Identify the Collections and their methods.
3. Identify the Sorting algorithms.
4. Determine which elements are used and how they are generated.
5. Calculate the amount of energy consumed.
6. Calculate the usage of CPU and memory.

7. Determine how many experiments should be conducted.

In the first step, the device on which all experiments will be conducted and the utilized software tools are determined. The tools that have been utilized are IntelliJ IDEA Ultimate, JoularJX, and Rstudio. IntelliJ IDEA is a highly advanced and customizable integrated development environment aimed at assisting developers in writing efficient code and being able to maintain it. Code highlighting, refactoring tools, debugging, code analysis, profiling, and support for a broad range of programming languages and frameworks are among its primary features. These features help in writing efficient, clean code and easily identify and fix errors. Therefore, it has been selected to be used for implementing the experiment [17].

Investigating different tools or approaches for energy measurement is not reported, since it is not the objective to compare different tools for measuring energy or propose a new way. Rather, it is assumed that energy consumption will be collected using one of the available tools and concentrate on investigating the relationship between energy consumption and some quality attributes. Therefore, the second tool, JoularJX is an advanced Java-based agent for monitoring energy consumption at the source code level. It seamlessly integrates with the JVM during program startup. JoularJX uses a custom PowerMonitor program on Windows and Intel RAPL (via powercap) on GNU/Linux to obtain accurate energy readings. It enables real-time monitoring of energy consumption for individual methods, analysis of energy trends, and tracking of energy consumption in method-call trees and branches. JoularJX requires no source code modification and provides user-friendly and precise energy monitoring for optimizing Java program energy usage. Utilizing JoularJX can be simply done by integrating it with the Java Virtual Machine by adding the command "*java -javaagent:joularjx-$version.jar JavaProgram"* when lunching a Java program through a device terminal. Lastly, JoularJX provides a configuration option in the form of a "*filter-method-names*" variable, which can be specified in the config.properties file. This feature allows developers to selectively designate the names of specific methods for energy consumption measurement. By separating the method names with commas, JoularJX focuses on monitoring and quantifying the energy consumption of each method individually. On the other hand, in the absence of method name specifications, JoularJX considers the entire program as a unified entity when measuring energy consumption. This flexible configuration capability empowers developers to conduct fine-grained analysis of energy usage by isolating and monitoring specific methods of interest[18]. The third tool is Rstudio, which is an IDE for R, provides better functionality, and includes a source code editor, build automation tools, and a debugger. R is a programming language for statistical analysis and graphics. It is widely used for data analysis among statisticians [19].

After determining the work environment, the Java Collections and Sorting algorithms on which all experiments are conducted are determined. As a part of the Java programming language, Collections include ArrayList, LinkedList, Vector, Stack, ArrayDeque, PriorityQueue, HashSet, LinkedHashSet, and TreeSet with different time complexity for each one of them. Java Collections are used to store data -to apply specific operations on the data such as search, insertion, and deletion which is widely commonly used among developers [20]. Therefore, Java Collections has been selected to conduct the experiments. Notably, Collections are implementations of various data structures, their primary purpose is to organize, process, retrieve, and store data [21]. This study selects inserting, deleting, and searching operations as these operations are commonly and widely used. In Java Collections, these three operations are represented by the methods Add, Contain, and Remove. Sorting algorithms in Java, on the other hand, provide the best ways to sort data in a specific order with a given list of elements, and the number of available algorithms is enormous[6]. As a result, the experiment will be done using the most well-known algorithms; Insertion sort, Merge sort, Bubble sort, Selection sort, Quick sort, Counting sort, Heap sort, Bucket sort, Shell sort, and Radix sort, while taking into concern the time complexity of each algorithm. The Sorting algorithms are implemented using external resources from existing implementations; the full source code can be found in **Appendix 1**. This decision is made to have greater control over the implementations and align them with the specific requirements of this study. Implementing Sorting algorithms instead of using the Java Library for built-in Sorting has lots of advantages. It helps to have a deeper understanding of the algorithm's nature, allowing the researchers to tailor it specifically for the measurement and analysis of energy consumption. This approach also provides the opportunity to detect and address any potential errors while experimenting, which enhances the reliability and validity of the study's findings.

As a part of this process, 400,000 randomly generated integer elements are to be created using the Random class in Java, with a specific seed to ensure that all operations for all identified Collections and Sorting algorithms are performed using the same elements exactly. To clearly examine the differences between the performance of each Collection and Sorting algorithm in terms of CPU, memory usage, and energy consumption, it is necessary to select that number of random distinct integers. In addition, if the number of elements is less than 400,000, JoularJX may not be capable of measuring the energy consumption for all methods. Moreover, that number of elements is quite enough to get reasonably distinguished data on energy, CPU, and memory which are used later for statistical analysis to find such a relationship.

The energy consumption for Sorting algorithms as well as inserting, searching, and removing all elements for each Collection, is calculated using the JoularJX tool [18]. In addition, CPU and memory consumption will be measured using the OperatingSystemMXBean interface and Runtime class, respectively. This model will be repeated 100 times for each Sorting algorithm and Collection, including the operations Add, Contain, and Remove, to determine the energy consumption, CPU, and memory usage as part of the final step of the data collection process. The number of experiments is decided to be 100 to increase the accuracy of the measurements, which leads to increased reliability. Additionally, that number of experiments is feasible to be conducted. Once the data has been gathered, a statistical analysis will be conducted using R (a programming language for statistical analysis and statistical graphs) following the linear regression analysis, to determine the relationship between energy consumption and the dynamic quality attributes (CPU and memory)—if it exists—and how strong it is. Moreover, according to the gathered data, the best Collection and Sorting algorithm will be found based on energy consumption while considering performance in terms of memory and CPU usage.

As one of the most commonly used statistical models, linear regression simulates a mathematical relationship between two variables, either independent or explanatory variables, or dependent variables, providing a rational basis for identifying and predicting results that are based on scientific calculations [22]. This model offers an easy-to-understand interpretation of the result and is relatively simple [23].

2.3 Reliability and Validity

The terms reliability and validity are significant when conducting a research paper, as they contribute to the credibility of the study's results. Reliability refers to the repeatability of the study results using the same methodology approach. If the same procedure is repeated under similar conditions, the same results will be obtained. Moreover, validity encompasses the entire process of a research, it can refer to the accuracy of the method, how it is supported, and how a valid conclusion is drawn based on the collected data. In this paper, all measurements performed in the controlled experiment are done programmatically and in an automated fashion to increase reliability. A significant quantity of measurement iterations also provides further reliability. The validity issues are greatly minimized by controlling the variable (the elements) and generating identical random integer elements by quantity and order, to perform the experiment for all methods and all Collections. In addition, an improved measurement technique is used, which increases validity, and generating random elements reduces sample bias. Lastly, the related work should align with this research results, this alignment helps to strengthen the validity of the study's conclusions. The discussion regarding the alignment of the research findings with the related work will be presented in Section 7.

2.4 Ethical Considerations

The measurement of estimated energy consumption, CPU, and memory usage for Java Collections and Sorting algorithms provides no negative ethical effects on any scale abstractly. However, it is important to consider ethical considerations in the research design and execution process. For instance, it is essential to ensure that the measurement process is conducted in an objective and transparent manner without any manipulation of data or results.

3 Theoretical Background

This chapter contains information about important theoretical concepts included in this study, such as time and space complexity, energy, CPU, and memory consumption. Moreover, this chapter provides a brief explanation of the Java Collection interface, its classes, and the Sorting algorithms used in the experiments.

3.1 Time and Space Complexity

Time complexity refers to the duration or elapsed time required for the execution of an algorithm. Simply, the time complexity is a metric that quantifies the number of operations carried out by an algorithm and provides an estimation of the time needed to execute these operations [24].

Typically, when it comes to time complexity, it is referred to as 'Big O Notation', whereas Big O Notation is a notation that is used to describe the time complexity of an algorithm or a piece of code in the worst-case scenario [25]. Moreover, the "order of growth" of time complexity may vary from algorithm to algorithm based on the running time when the input size increases [25]. As shown in the table below, the ‘order of growth’ is ordered from the best to the worst, where n is the input size [25].

Time Complexity	Time	Description
O(1)	Constant	The computation time is constant. No dependence on input size
O(log n)	Logarithmic	The computation time is proportional to log(n)
O(n)	Linear	The computation time is proportional to n
O(n log n)	n*log n	The computation time is proportional to n times log n
O(n^2)	Quadratic	The computation time is proportional to n^2
O(n^3)	Cubic	The computation time is proportional to n^3
O(2^n)	Exponential	The computation time is proportional to 2^n
O(n!)	Factorial	The computation time is proportional to n!

Table 3.1 Time Complexity

On the other hand, space complexity refers to the amount of memory space required by an algorithm or program during its entire execution. It is a measure of how many variables are created to store the values, including the inputs as well as the outputs [24].

3.2 Consumption Metrics

- Energy consumption

The energy consumed by a method or a program is measured in a Joules. This is a metric that can be used to evaluate how much energy is required to run a particular method or program.

- CPU and Memory consumption

CPU consumption refers to how much load an individual processor core can handle.. It measures the percentage of time when the CPU is processing instructions. CPU consumption percentage equals TotalUsedCPUTime divided by TotalAvailCPUTime [26]. On the other hand, memory consumption in this study refers to JVM heap memory consumed by Java Collections operations (Add, Contains, and Remove) as well as Sorting algorithms [27]. Generally, high CPU and memory usage can negatively affect a device, resulting in low performance and efficiency.

3.3 Java Collection Interface

The collection interface contains three subinterfaces: List, Set, and Queue, where several classes implement each subinterface [28]. The classes HashSet and LinkedHashSet implement the Set interface, while the class TreeSet, implements the SortedSet interface, which extends the Set interface. The classes ArrayList, Vector, and LinkedList implement the List interface, while the Stack class extends the Vector class. Moreover, the class PriorityQueue implements the Queue interface, while ArrayDeque and LinkedList implement the Deque interface, which extends the Queue interface. See Figure 3.1.

These classes represent Java Collections which are widely used by software engineers throughout the development phase. Java Collections are used to store data and apply specific operations to it later such as searching and deletion [20]. Moreover, since Collections are implementations of various data structures, their primary purpose is to organize, process, retrieve, and store data [21].

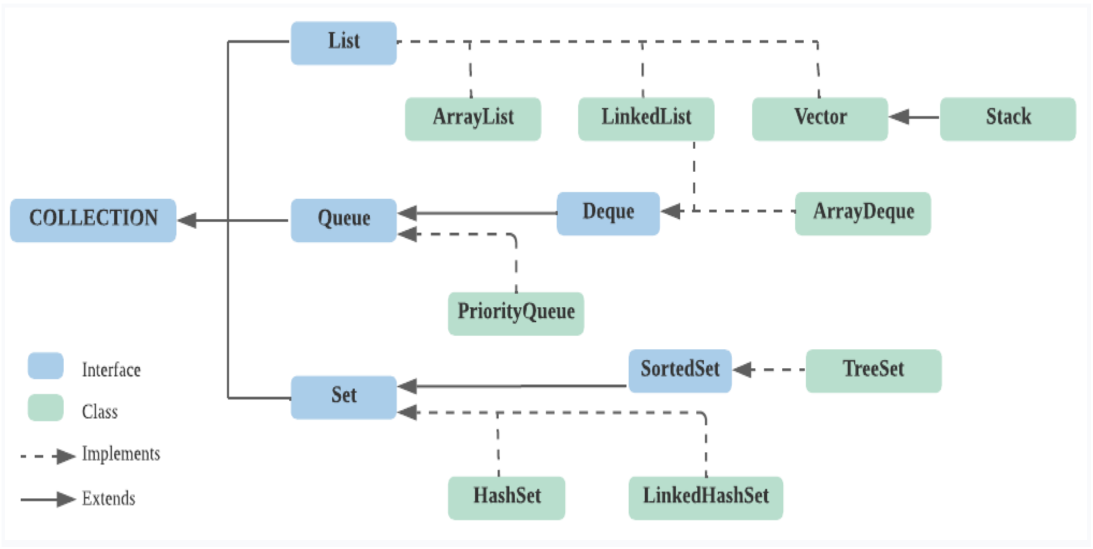

Figure 3.1 Collection interface hierarchy.

- Array List: It is a sequential collection where each stored element has a position. It is implemented using a resizable array, which grows as new elements are added and shrinks as they are removed. Each element is retrieved by its index [29].
- Linked List: It is a sequential collection representing a sequence of linked nodes where each node consists of a value and a reference to the next node. Moreover, a Linked list provides flexible size, and adding or removing elements from the beginning or end of the Linked list is efficient. Therefore, it is useful and recommended when managing dynamic collections of data. On the other hand, it is less efficient in the case of accessing an element at a specific index compared to an array [29].
- Vector: It is a sequential collection very similar to an Array list; it grows and shrinks when a new element is added or removed, and each element is retrieved by its index. The main difference between a Vector and an Array list is that each operation is synchronized, which means that if a thread is performing an operation on a Vector, no other threads can access that Vector. Therefore, Vector is recommended over the Array list in case of multithreading [30].
- Stack: Stack is a class that implements the Collection interface and represents a last-in-first-out (LIFO) data structure [29]. It is a subclass of Vector and provides all the methods of Vector in addition to its own methods for pushing and popping elements.
- ArrayDeque: It is flexible in size and represents a double-ended queue (deque). A deque is a data structure that supports element insertion and removal from both sides. ArrayDeque is implemented as a circular array and provides constant-time performance for the basic operations (Add, Remove, and Get) at both ends of the deque. Moreover, It is not recommended in the case of multithreading since it is not thread-safe [31].

- PriorityQueue: PriorityQueue is a class that implements the Collection interface and represents a first-in-first-out (FIFO) data structure (Queue). The Difference between PriorityQueue and normal Queue is that a priority queue is a data structure that stores elements in a particular order based on their priority. The elements with the highest priority are placed at the front of the queue. In contrast, elements with equal priority are ordered according to their natural order or based on other criteria. PriorityQueue implements Queue interface [29].
- HashSet: It is a class that implements the Set interface. It is an unordered collection of unique elements, meaning no element is duplicated. HashSet uses a hash table to store elements, which provides fast access in the case of Add, Remove, and Contains operations. The hash table uses a hashing function to determine the index of each element in the table. Moreover, HashSet is useful in case the user needs to store unique and ordered data and perform operations such as Add, Remove, and Contains in the data since HashSet has no guarantee on data order and data cannot be duplicated. Lastly, it is worth mentioning that HashSet is implemented using HashMap [32].
- LinkedHashSet: is a class that implements the Set interface. It is very similar to HashSet, but the difference is that LinkedHashSet implements additionally to HashSet a LinkedList as well just to keep the order of the elements. LinkedHashSet is useful when the end-user needs to maintain the order of the unique elements that were inserted into the set. However, keeping the order of the elements that were inserted comes at a cost. Therefore, LinkedHashSet is slightly slower than Hashset in Add, Contains, and Remove operations [33].
- TreeSet: It is a class that implements the SortedSet interface. It uses a balanced search tree to implement the sorted set, where the elements are stored and sorted in an ascending order based on their natural ordering [29].

3.4 Sortings Algorithms

The process of arranging a list, a sequence of provided elements, or data collection into a specific order, is called sorting. In other words, organizing a list of items in a particular order or way [8]. Consider arranging a messy room - clothes can be categorized by color, size, or type to make it simpler when searching. Similarly, Sorting algorithms help to organize and simplify complex sets of data so that they can be searched, analyzed, or displayed more effectively. There are several Sorting algorithms available, varying from simple algorithms such as Bubble sort to more complex ones like Quick sort or Heap sort. The algorithm chosen is determined by several factors, including the size of the data set, the type of data being sorted, and the desired efficiency of the sorting process [6] [8].

In the controlled experiment, ten Sorting algorithms will be considered. Starting with the simplest Sorting algorithms. *Bubble sort* compares two adjacent elements to search for the smallest; if the first index is larger than the second, it will be swapped, and the process is repeated from the beginning to the end of the list. Even though it is a popular algorithm, it is inefficient [6]. While in the *Selection sort*, finding the minimum element in a list is the first step, which is then swapped to the beginning of the list. This will be repeated until the entire list is sorted [8]. In *Insertion sort*, the element is repeatedly swapped until it is inserted in the proper index, by swapping it to the left of the unsorted list. In Insertion sort elements are moved ahead by one position only, so the average time complexity needed improvement; therefore, Shell sort existed. *Shell sort* allows the swapping and movement of far-away elements, not just the adjacent ones. This can be achieved by determining a gap number, which is called an Interval. The Interval is reduced after each process of sorting until it reaches 1 and then switches to using Insertion sort to complete the sorting process. Once the Interval number is chosen, the first index of the list will be compared to the index of the Interval number, and if the first index is larger, swapping the elements will happen. This is also done for the second index element with its Interval number, and so on [6] [8].

Efficient algorithms are also considered in this experiment. *Heap sort,* as an example, uses comparisons and is based on the Binary tree structure [6]. First, it discovers the minimum or maximum element and then places it as the root. If the maximum element is chosen as the max heap (root), it means the parent node must be greater than its children nodes, and the largest element is at the root position. The opposite is true for the minimum heap. Heapify is an important method to understand how the Heap algorithm sorts the elements. Heapify starts from the parent of the last element in the unsorted list and recursively moves down the binary tree, it compares each node with its children and swaps them if necessary. This process is repeated for each parent node until the root of the heap is reached, which results in a complete binary tree that satisfies the heap property [6]. *Merge sort* is also an efficient algorithm, it works by recursively dividing the unsorted list into two halves, sorting those halves, and then merging them back together. The algorithm divides the list into two equal halves, then sorts each half recursively. Finally, merges the two sorted halves into a single sorted list. This is repeated until the entire list is sorted. The merging process compares the first element of each sublist and selects the smaller element to insert into the output list, and this will continue until all elements have been merged into a single sorted list. On the other hand, the *Quick sort* algorithm works by selecting a pivot element from the unsorted list. Then divide the remaining elements into two sub-lists based on whether they are greater than or less than the chosen pivot. Pivot is an important factor in the performance of the algorithm, and the typical scenario for choosing a pivot element is usually to be the middle element of the list [6]. The *Bucket sort* algorithm divides the elements of the unsorted list into several groups called buckets, and each bucket holds inside it a certain range of numbers. After distributing the elements inside their corresponding bucket, each bucket is sorted individually using any Sorting algorithm, and the most commonly used is Insertion Sort. The process behind Bucket sorting can be understood as a “scatter-sort-gather” approach. Elements are first scattered into buckets then the elements in each bucket are sorted and finally the elements are gathered in order. Because of the uniform distribution of this method, the number of comparisons is reduced, and it is asymptotically fast [6]. Moreover, *Counting sort* does not perform comparisons, it instead counts the occurrences of each element in the unsorted list and then creates a list of counters, with one counter for each possible element in the input range. The counters are initialized to zero, and each input element is counted by incrementing the corresponding counter. Next, a cumulative sum of the counters is computed. Finally, the input elements are placed in their sorted positions by iterating through the original list, using the counters and cumulative sum to determine their index in the sorted list. However, when the number of elements stored in the list is very large, this algorithm might not turn out to be as efficient as it should be. To solve this issue, *Radix sort* was born. The Radix sort algorithm implements the same logic as the Counting sort but, uses only one digit of the element as the index. In this way, it can guarantee that the occurrences list will at maximum consist of 10 slots [6].

3.5 Statistical Terms

Relevant terms belonging to the statistics field as Linear Regression Model, Correlation Coefficient, Estimates, and P-Value are used.

- Linear Regression Model: One of the most used statistical models, available to simulate a mathematical relationship between two variables, either independent or explanatory variables, or dependent variables, which provides a rational basis for identifying and predicting results that are based on scientific calculations [22]. This model offers a relatively simple, easy-to-understand interpretation of the result [23]. This model is used in the statistical analysis of the raw data, to find the relationship between the quality attributes(CPU, memory) and energy consumption.

- Correlation Coefficient: parameter by which the existence and the strength of a linear relationship between two variables is tested and its value ranges from 1 to -1 [34]. In this paper, the Correlation Coefficient is used to find out the strength of the relationship between different variables.

- P-Value: Probability value which is used to decide whether the estimated quantities are statistically significant [35]. The P-Value measures the reliability of the obtained results, as it will show if the relationship is statistically significant.

- Estimates: Values are generated through the ordinary least squares method, which finds the line that fits the points in such a way that it minimizes the distance between each point and the line [36]. The ‘Estimates’ value shows how much the dependent variable’s value increases or decreases when the independent variable’s value increases by one unit.

- Standard Deviation: The dispersion or variation of set values. In other words, it is the difference between these values and the mean of the same set of values [36]. The standard deviation is not used directly but its value shows the variations in the analyzed data.

4 Research project – Implementation

This study relies on a controlled experiment methodology to base all its findings and results. This methodology assists in showing and identifying the necessary steps for implementation. This chapter covers the implementation approach and decisions taken throughout the process. Introducing and explaining the experiment and providing a step-by-step walkthrough of the technique for each experiment. This part also presents the tools used, the steps of data collection and summary, and how they are further handled.

4.1 Tools and Techniques

All experiments were conducted on an Asus laptop with a system model X411UA, 8GB of RAM, an Intel(R) coreTM i5-8250U CPU, and Windows 10 Home 64/bit. IntelliJ IDEA-2022.3 and Java JDK-18.0.2.1 were installed to implement and design the code. JoularJX 1.1 was also installed to monitor the energy consumption of each method [18]. The element chosen as an input for each Collection operation and Sorting algorithm is random integers. Using the Random class in Java, 400 000 random integers were created and stored, with a specific seed value of 200.

Measuring the CPU and memory usage in the implementation for each method was done using built-in Java classes. The CPU usage was calculated using the function 'calcCPU' which was called after each operation for each Collection and Sorting algorithm. 'calcCPU' function takes three parameters: *cpuStartTime, elapsedStartTime, and cpuCoun*t, as shown below.

```
Function calcCPU(cpuStartTime, elapsedStartTime, cpuCount)
    end = current time in nanoseconds
    totalAvailbleCPUTime = cpuCount * (end - elapsedStartTime)
    totalUsedCPUTime = currentThreadCPUTime - cpuStartTime
    Per = (totalUsedCPUTime * 100) / totalAvailbleCPUTime
        return per as integer
```

Figure 4.1 Pseudocode of CPU function

cpuStartTime was obtained using the method "*ManagementFactory.getThreadMXBean()*.getCurrentThreadCpuTime()", this method calculates and returns the overall CPU time consumed by the current thread, measured in nanoseconds [37]. On the other hand, "*elapsedStartTime*" was calculated using the method "*System.nanoTime()*" which returns the current time in nanoseconds [38]. Lastly, *cpuCount* was calculated using the method "*ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()*" which returns the number of processors available to the Java virtual machine [39]. This

function calculates the CPU usage percentage by dividing the *totalUsedCPUTime* by *totalAvailCPUTime*. The *totalAvailCPUTime* was calculated by multiplying *cpuCoun*t with the difference between end-time and *elapsedStartTime* which is the difference in time in nanoseconds before and after calling each operation of such a Collection or s
Sorting algorithm. While *totalUsedCPUTime* is the difference between the current total CPU time for the current thread in nanoseconds after calling each operation and *cpuStartTime* before calling each operation of such a Collection or Sorting algorithm. Memory consumption was calculated using the Runtime class. The Runtime class allows the application to interface with the environment in which the application is running [40]. Computing the memory usage of Collection operations and Sorting algorithms involves calculating the memory consumption before and after running the implementation. Memory consumption was computed by subtracting the free memory from the total memory after calling these methods respectively.

Monitoring and measuring the energy consumption of each operation or algorithm was easily done by modifying the filter in the Joular config file, starting with the "*package.class.method*" name. For example, to measure the energy of a specific Sorting algorithm, the filter would be modified to "*filter-method-names=insertionsort.InsertionSortAlgorithm.sort*". Finally, executing the program to get the desired results was done using a bash script with the JoularJX Java agent enabled. An example of a script used is "*$for i in {1..20}; do java -javaagent:joularjx-1.1.jar selectionsort/Main 400000; done*". This command runs the Java program 20 times, each time with a different randomly generated array of 400 000 integers, and measures the power consumption of the Java program using JoularJX.

Regarding Java Collections it is worth mentioning that each collection's object such as ArrayList, LinkedList, etc was instantiated in the Main method using the Collection interface which is implemented by Collections' classes.

4.2 Data Overview

As a result of the experiments, raw data was generated. Therefore, this section provides a comprehensive understanding of the data collection process and how it is analyzed.

4.2.1 Data Collection

Regarding memory and CPU usage, data was gathered and stored in a file using the 'writeTofile' method. This method takes two arguments, the name of the file and the result of the experiment to be written to the file, then creates three separate CSV files for each result. Moreover, the energy data was generated automatically using JoularJX. JoularJX creates a CSV file for each experiment independently. Thus, if the implementation runs 100 times, Joular will generate 100 CSV files containing the energy consumption value. Then, the energy values of the 100 files were manually gathered into one CSV file. Finally, all generated data (Energy/Memory/CPU) was stored in one Excel file, with different spreadsheets for each Collection and Sortings, to apply the analysis later.

4.2.2 Data Analysis

In order to conduct the analysis, the final Excel file was converted to a CSV format, and subsequently imported into RStudio. As mentioned earlier in the Methodology Section, RStudio is a software that reads and examines the data before applying statistical analysis techniques using R language to derive insights and draw conclusions from the data. RStudio uses commands, as an example, to read a CSV file, this command is used: "*mydata=read.csv("fileName.csv",header = TRUE)*", and to get the summary of the data, *"summary(mydata)*" command is used, which generates summary statistics for the data frame. The summary includes different statistically related values, some of these values are interesting, such as the minimum, maximum, and mean values.

To find the correlation coefficient between energy and memory, for instance, simply use the *"cor(mydata$VariableX, mydata$VariableY)*" command. The "cor()" function calculates the Pearson correlation coefficient between the two variables. The correlation coefficient ranges from -1 to 1, which indicates how weak or strong the relationship is between two variables. This process was repeated for all other variables. Finally, a linear regression model was conducted for any two desired variables using the command "*model = lm(mydata$VariableX~mydata$VariableY)*" command. VariableX is the dependent variable, while VariableY is the independent variable. This command helps to understand the relationship between these variables and make predictions about future values.

5 Results

This chapter offers an overview of the gathered data from the controlled experiments performed on the Java Collections and Sorting algorithms. It provides a summary of the values for each Sorting algorithm and Collections operation, which helps later in comparisons to be made between different variables while making analysis easier. The complete raw data can be found in **Appendix 1**.

The following tables present a summary of the data, encompassing the minimum, maximum, and mean values for each of the variables Energy, CPU, and Memory. Also the time complexity of each algorithm and operation.

Collections			Energy (Joule)	CPU Usage	Memory (MB)
ArrayList	**Add** O(1)	Min	0.000567	0.07	14.45995331
		Mean	0.001974369	0.081	14.45996635
		Max	0.006732762	0.1	14.45999908
	Contains O(n)	Min	466.6086355	0	9.086959839
		Mean	491.9261308	0.1188	9.891752014
		Max	515.679196	0.12	17.27084351
	Remove O(n)	Min	46.900643	0.12	7.399879456
		Mean	62.00662112	0.12	7.516352921
		Max	70.71309534	0.12	8.001930237
ArrayDeque	**Add** O(1)	Min	0.000687	0.06	14.45995331
		Mean	0.002953216	0.0739	14.4599704
		Max	0.014700036	0.13	14.45999908
	Contains O(n)	Min	472.8739621	0.12	7.84828949
		Mean	490.794287	0.12	8.734794006

		Max	504.286983	0.12	11.90093231
	Remove **O(n)**	Min	0.001116075	0.09	5.95791626
		Mean	0.008450762	0.134	6.094021988
		Max	0.042407282	0.24	6.35068512
LinkedList	**Add** **O(1)**	Min	0.000488	0	15.45904541
		Mean	0.001780715	0.0818	15.45996564
		Max	0.008296656	0.13	15.46000671
	Contains **O(n)**	Min	1449.880256	0.07	2.3462677
		Mean	1499.166092	0.1192	3.400434647
		Max	1537.974074	0.12	6.588676453
	Remove **O(n)**	Min	0.002567978	0.09	5.824249268
		Mean	0.013630518	0.1366	5.827473984
		Max	0.050296586	0.24	6.001930237
Stack	**Add** **O(1)**	Min	0.001580613	0.03	12.99993896
		Mean	0.004702815	0.0974	12.9999517
		Max	0.012583005	0.13	12.99998474
	Contains **O(n)**	Min	436.7424848	0.12	7.897949219
		Mean	489.6172627	0.12	8.882812805
		Max	518.8516553	0.12	14.23099518
	Remove **O(n)**	Min	34.87031117	0.11	7.399154663
		Mean	57.72997228	0.1199	7.511531067
		Max	66.05255707	0.12	8.001930237

Vector	**Add** **O(1)**	Min	0.001390911	0.03	12.99993896
		Mean	0.004055882	0.0995	12.99995056
		Max	0.013321473	0.13	12.99998474
	Contains **O(n)**	Min	418.9313185	0.12	7.729530334
		Mean	482.597081	0.12	8.513253479
		Max	507.1413135	0.12	15.41547394
	Remove **O(n)**	Min	38.18918344	0.12	7.398963928
		Mean	58.44471894	0.12	7.487701416
		Max	67.71201172	0.12	7.769287109
PriorityQueue	**Add** **O(log n)**	Min	0.002186143	0.06	13.45995331
		Mean	0.008133778	0.1072	13.45996658
		Max	0.020972939	0.12	13.45999908
	Contains **O(n)**	Min	557.3652651	0	0.125434875
		Mean	626.1236434	0.1176	2.450000763
		Max	678.308702	0.12	9.447479248
	Remove **O(n)**	Min	352.7094224	0.12	4.065910339
		Mean	403.8143728	0.12	5.802430878
		Max	479.8554691	0.12	9.34853363
	Add **O(1)**	Min	0.025050086	0.09	29.99992371
		Mean	0.049346392	0.1122	29.99994865
		Max	0.081101626	0.12	29.99998474
		Min	0.00432408	0.09	2.354904175

HashSet	**Contains** **O(1)**	Mean	0.018042638	0.1124	5.404334793
		Max	0.04702791	0.16	6.001930237
	Remove **O(1)**	Min	0.00544792	0.08	2.2056427
		Mean	0.020921409	0.1154	5.271509247
		Max	0.065847316	0.17	5.761940002
Linked HashSet	**Add** **O(1)**	Min	0.042830875	0.07	30.41864777
		Mean	0.070685211	0.0914	30.46009758
		Max	0.160779235	0.12	30.48896027
	Contains **O(1)**	Min	0.007027301	0.04	1.270233154
		Mean	0.015740149	0.1092	2.257113419
		Max	0.03793517	0.15	2.419761658
	Remove **O(1)**	Min	0.00553079	0.08	1.164207458
		Mean	0.019623584	0.1239	2.172135239
		Max	0.046522515	0.17	2.229278564
TreeSet	**Add** **O(log n)**	Min	0.089415472	0.09	18.89191437
		Mean	0.141886684	0.1022	19.13297615
		Max	0.309176967	0.11	19.19422913
	Contains **O(log n)**	Min	0.043751044	0.11	6.048225403
		Mean	0.093472037	0.114	6.07582695
		Max	0.197342535	0.13	6.243865967
	Remove **O(log n)**	Min	0.056256182	0.11	5.922714233
		Mean	0.148259318	0.1169	5.995325775

		Max	0.429610314	0.13	6.221954346

Sorting Algorithms		Energy (Joule)	CPU Usage	Memory (MB)
Bubble Sort O(n^2)	Min	1241	0.000	26.42
	Mean	1553	0.114	27.85
	Max	1621	0.120	31.40
Bucket Sort O(n+k)	Min	1557	0.0100	66.63
	Mean	1953	0.1117	80.23
	Max	2038	0.1200	89.98
Counting Sort O(n+k)	Min	0.3436	0.0700	10.00
	Mean	1.4559	0.0994	10.01
	Max	3.0048	0.1200	10.06
Heap Sort O(nlogn)	Min	0.2293	0.0900	0
	Mean	0.6555	0.1100	0
	Max	2.2523	0.1200	0
Insertion Sort O(n^2)	Min	261.1	0.0400	14.98
	Mean	291.1	0.1192	15.03
	Max	300.9	0.1200	15.92
Merge Sort O(nlogn)	Min	0.3160	0.0700	22.14
	Mean	0.8153	0.1003	22.33
	Max	3.5041	0.1200	22.62
	Min	0.2312	0.0600	0

Quick Sort O(nlogn)	Mean	0.7036	0.1043	0
	Max	1.8024	0.1200	0
Radix Sort O(nk)	Min	0.3307	0.0900	27.46
	Mean	1.0818	0.1106	27.58
	Max	2.5685	0.1200	27.98
Selection Sort O(n^2)	Min	659.3	0.020	19.08
	Mean	775.0	0.119	28.59
	Max	830.8	0.120	30.20
Shell Sort O(n(logn)^2)	Min	0.1851	0.0900	0
	Mean	0.6374	0.1121	0
	Max	1.6549	0.1200	0

6 Analysis

The following chapter analyzes the results that were obtained by the conducted controlled experiments. The goal of this analysis is to answer the research questions that were introduced earlier in the Problem formulation section. All possible methods for analyzing the results have been explored to provide accurate answers to the research questions. Therefore, two analysis sections have been included, one applying a linear regression model and the other a 3D scatter plot.

6.1 Statistical Analysis

In this section, statistical analysis using a linear regression model is conducted for Sorting algorithms and Collections in Java to investigate the relationship between variables. This analysis will answer the research questions RQ1, RQ2, and RQ3 only from Section 1.3, Problem Formulation.

In order to present a linear regression model, the Correlation Coefficient is computed to determine whether the relationship is inverse or positive, as well as if the relationship is weak or strong. Moreover, the P-Value is calculated to determine if the relationship is statistically significant. **Appendix 2** shows the values of the correlation coefficient and P-Value for each Sorting algorithm and Collection including their three operations, which will be used in the analysis.

It is pertinent to note that in order to comprehend the information presented in the below Tables 6.1 & 6.2, it is essential to know that energy consumption is the dependent variable while memory and CPU usage are the independent variables. If the correlation coefficient value is negative, it indicates an inverse relationship between the two variables, and if the value is positive, it indicates a positive relationship. Moreover, when the absolute value of the correlation coefficient is less than 0.5, the relationship is considered weak, and the opposite is true if it is equal to or greater than 0.5. The estimated value presents the approximate amount of energy consumption when the independent variable increases by 1 unit. Finally, to know if the relationship is statistically significant, the P-Value must be less than or equal to 0.05. In the below tables, a correlation written in **Bold** indicates a statistically significant relationship and is to be discussed later in the conclusion section.

Worth mentioning, the relationship is also considered strong if the linear regression model shows that the relationship is statistically significant with a P-Values less than or equal to 0.05, even though the correlation coefficient is less than 0.5 since the P-Value is more important.

6.1.1 Collections' Linear Regression Analysis

Table 6.1, provides an analysis of the raw data for Collections including Add, Contains, and Remove operations, as an outcome of applying a linear regression model using R. Taking into concern their relationship with energy-memory usage and energy - CPU usage.

		Energy - Memory usage		Energy - CPU usage	
Collections	Operation	Relationship	Estimate	Relationship	Estimate
ArrrayList	Add	Weak Positive	6.002	Weak Inverse	-0.028008
	Contains	**Strong Positive**	3.7321	Weak Inverse	-53.261
	Remove	Weak Positive	0.4489	-	-
ArrayDeque	Add	Weak Inverse	-2.240	**Strong Positive**	0.092071
	Contains	Weak Positive	0.4627	-	-
	Remove	**Strong Inverse**	-0.014974	**Strong Positive**	0.057140
LinkedList	Add	Strong Positive	0.1859	**Strong Positive**	0.0199417
	Contains	Weak Positive	2.057	Weak Inverse	-253.74
	Remove	Weak Positive	0.01710	**Strong Positive**	0.082093
Stack	Add	Weak Inverse	-6.909	Weak Positive	0.0113430
	Contains	Weak Inverse	-0.4485	-	-
	Remove	**Strong Inverse**	-10.636	**Strong Positive**	2309.06
Vector	Add	Weak Positive	4.952	Weak Positive	0.011879
	Contains	Weak Inverse	-1.764	-	-
	Remove	Weak Positive	0.7895	-	-
PriorityQueue	Add	Weak Positive	13.65	Weak Inverse	-0.032627
	Contains	Weak Inverse	-1.591	Weak Inverse	-195.43
	Remove	**Strong Positive**	17.160	-	-

		Energy - Memory usage		Energy - CPU usage	
HashSet	Add	Weak Inverse	-2.877	Weak Positive	0.02009
	Contains	**Strong Inverse**	-0.0013681	**Strong Positive**	0.083060
	Remove	**Strong Inverse**	-0.0041078	Weak Positive	0.002688
LinkedHashSet	Add	Weak Inverse	-0.2662	Weak Inverse	-0.01061
	Contains	Weak Inverse	-0.006344	**Strong Positive**	0.042383
	Remove	Weak Positive	0.006922	**Strong Positive**	0.080187
TreeSet	Add	Weak Inverse	-0.02231	Weak Positive	0.58851
	Contains	Weak Inverse	-0.08832	Weak Positive	0.28658
	Remove	Weak Inverse	-0.06375	Weak Positive	0.10425

Table 6.1 Collections' Relationships Analysis

The table above addresses RQ1 by demonstrating the relationship between energy consumption and memory usage and the relationship between energy consumption and CPU usage. Moreover, it also addresses RQ3 by demonstrating how strong these relationships are, based on the correlation coefficient and P-Value, values as explained earlier. For example, the table shows that in the case of HashSet Collection for 'Add' operations, the relationship between energy consumption and memory is weak and inverse. Therefore, when memory usage increases by one megabyte, energy decreases by 2.877 Joules, as the "Estimate" value shows.

Notably, the table above shows no significant relationship between energy consumption and CPU usage for some Collections. The reason behind this result is that the standard deviation is zero since the CPU usage is constant and fixed. In other words, based on the collected data, it is noticed that CPU usage is fixed regardless of whether energy consumption increases or decreases.

6.1.2 Sortings' Linear Regression Analysis

Table 6.2, provides an analysis of the raw data for Sorting algorithms, as an outcome of applying a linear regression model using R. Taking into concern their relationship with energy-memory usage and energy - CPU usage.

	Energy - Memory usage		Energy - CPU usage	
Sorting Algorithms	Relationship	Estimate	Relationship	Estimate

Bubble Sort	Weak inverse	-13.635	Weak inverse	-75.26
Bucket Sort	Weak positive	1.1014	Weak positive	193.80
Counting Sort	Weak inverse	-1.146	Weak inverse	-4.3836
Heap Sort	-	-	**Strong positive**	7.8139
Insertion Sort	**Strong inverse**	-26.386	Weak positive	137.313
Merge Sort	Weak inverse	-0.4435	Weak positive	0.9381
Quick Sort	-	-	Weak inverse	-0.5146
Radix Sort	Weak inverse	-0.2801	Weak positive	2.8106
Selection Sort	**Strong positive**	11.7746	Weak inverse	-117.32
Shell Sort	-	-	**Strong inverse**	-5.0116

Table 6.2 Sortings' Relationships Analysis

The table analyzes the data and addresses RQ2 by first answering if a relationship exists. The ones with defined relationships have been categorized into positive or inverse relationships. Taking Selection sort as an example, there is a positive relationship between energy consumption and memory usage, and this means when memory increases by one megabyte, energy consumption increases by 11.7746 joules (estimated value). However, the relationship between energy consumption and CPU is inverse for the same Sorting algorithm; this means when CPU increases by one unit, energy consumption decreases by 117.32 joules (estimated value). On the other hand, RQ3 is answered by showing if the relationship between two variables is weak or strong. For example, in the Insertion sort, the two variables, memory, and energy, indicate a strong relationship, meaning if memory (the independent variable) increases, energy consumption (the dependent variable) decreases significantly. In the same Sorting algorithm, the two variables, CPU and energy, indicate a weak relationship, meaning if CPU (the independent variable) increases, energy consumption (the dependent variable) increases slightly.

Table 6.2 exhibits no significant correlation between energy consumption and memory usage for some Sorting algorithms. This lack of correlation is attributed to differences in the implementation of these algorithms. For instance, the Heap sort method operates directly on the input array and rearranges the elements in place, without creating any additional arrays or data structures to store intermediate results. The only memory usage in the Heap sort method is the storage of a temporary variable 'temp' that is used to swap the values of two elements. However, this temporary variable is a constant-size integer and does not depend on the size of the input array, so it does not contribute significantly to the memory usage of the algorithm. Similarly, Quick sort implementation does not use any additional memory also, since it sorts the input array in place, meaning that it modifies the input array rather than creating a new array to hold the sorted elements. Furthermore, Shell sort implementation performs in-place comparison sorting, which means it does not require any additional memory to be allocated for temporary storage during the sorting process. In summary, the lack of significant correlation between energy consumption and memory usage in these Sorting algorithms is attributed to their respective in-place sorting implementations, where they do not require additional memory allocation or data structures for temporary storage during the sorting process.

6.2 Efficiency Analysis

This section provides an efficiency analysis for Java Collections and Sorting algorithms. It presents a rigorous evaluation of their performance, in terms of energy consumption considering quality attributes (Memory, CPU) based on appropriate measurements and analyses.
The focus of this section is to answer the final two research questions:

- ☐ **RQ4.** What is the most efficient Java Collection/s among the three operations in terms of energy consumption considering the performance?
- ☐ **RQ5.** What is the most efficient Java Sorting algorithm in terms of energy consumption considering the performance?

A proper analysis of the results is necessary to address the research questions. Each Collection and Sorting algorithm was evaluated based on three principal objectives, energy consumption, memory, and CPU usage. The goal is to present these objectives in an understandable graphical form, to show, and determine the most efficient method with the best performance in terms of energy consumption considering CPU and memory usage.

Therefore, a 3D scatter plot is generated using R Studio, which is a suitable technique to assess the three objectives and gives the reader a visualized model to compare the data points. The data points represent the mean value of memory, CPU, and energy consumption respectively in a three-dimensional coordinate system, with one variable plotted on the x-axis and the other variable plotted on the y-axis and z-axis.

6.2.1 Collections Experiment Analysis

The efficiency analysis of Java Collections involves evaluating each Collection individually to determine the most efficient Collection(s) in terms of energy consumption, CPU, and memory usage. If it is not possible to find a Collection that excels in all three aspects, alternative approaches prioritize two aspects to identify the most efficient Collection(s).

The efficiency analysis of Java Collections does not rely on determining the most efficient Collection(s) as List, Queue, and Set. This approach is not aligned with the research objectives and methodology of the study. A Collection classified as a List, for example, might be the most efficient among other collections classified as a List. However, this does not necessarily imply that it is the most efficient of all Collections classified as Queues or Sets. Attempting to find the most efficient Collection(s) based on their classification would not yield the desired results. It would undermine the study's strength and coherence, as it deviates from the core objective of this study and does not align with other related works.

Instead, the efficiency analysis should focus on evaluating the performance of individual Collections based on specific criteria, such as energy consumption, CPU utilization, and memory usage. This approach ensures a comprehensive and accurate assessment of each Collection's efficiency without being restricted by its classification.

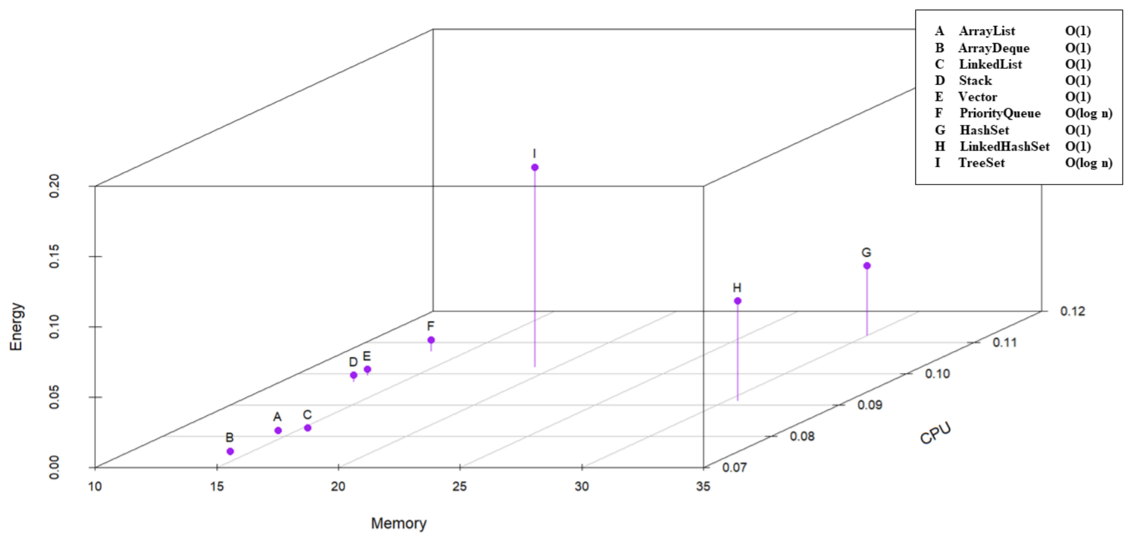

Figure 6.1: Collections 3D scatter plot for Add

Figure 6.1 shows a 3D scatter plot of all Collections in the case of the Add operation. The graph shows the mean value of memory, CPU, and energy consumption for each Collection on the x-axis, y-axis, and z-axis respectively. From the plot, the most efficient Collections in terms of energy consumption can be derived, taking into account the quality attributes. For example, it is clearly shown that ArrayList (A), ArrayDeque

(B), and LinkedList (C) are the best choices in terms of the three values of memory, CPU, and energy consumption combined. LinkedList is the best choice when only energy consumption is considered because it has the lowest energy consumption (See **Appendix 1**). On the other hand, ArrayDeque is the best choice when considering CPU consumption, as it has the lowest CPU consumption. Vector, on the other hand, is the best solution in terms of memory consumption. When it comes to the least efficient Collections in terms of memory, CPU, and energy consumption, HashSet (G), LinkedHashSet (H), and TreeSet (I) are collectively the most problematic. TreeSet is the worst in terms of energy consumption. On the other hand, HashSet is the worst in terms of CPU consumption and LinkedHashSet is the worst in terms of memory consumption.

Notably, Based on the gathered data and the conducted efficiency analysis, the time complexity of Collections does not reflect their efficiency in terms of energy consumption, CPU, and memory usage in Add operation. TreeSet(I) is less efficient in terms of energy consumption than PriorityQueue(F), even though both have time complexity O(log n). Additionally, some Collections have time complexity O(1) but they are less efficient than PriorityQueue in terms of one variable or two variables or all of them. For instance, LinkedHashSet is less efficient in terms of energy consumption and memory usage than PriorityQueue. Furthermore, HashSet(G) is less efficient than priorityQueue in terms of all aspects of energy consumption, CPU, and memory usage.

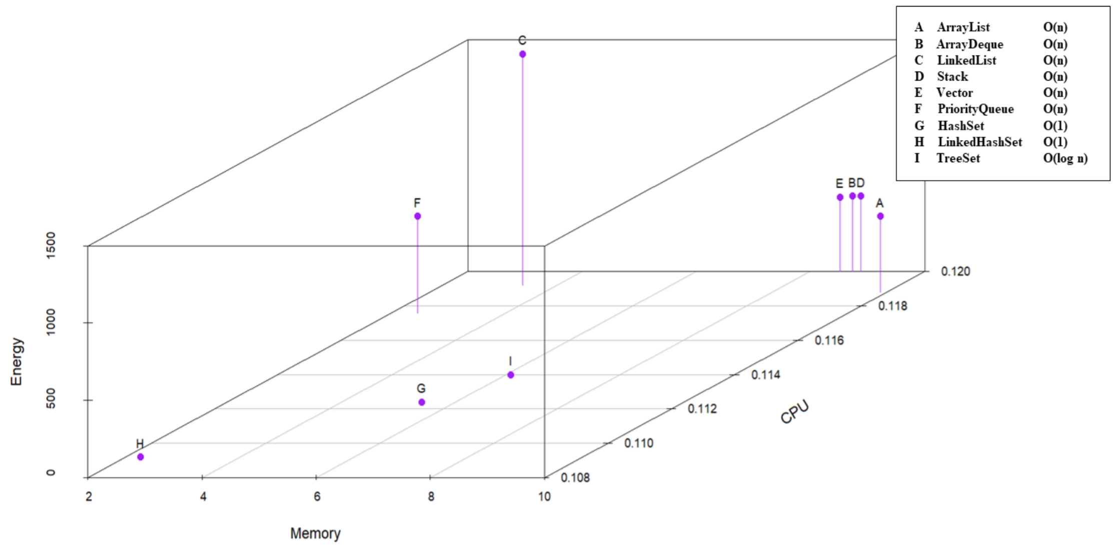

Figure 6.2: Collections 3D scatter plot for Contains

Figure 6.2 shows the memory, CPU, and energy consumptions of Collections for Contains operations on the x, y, and z axes respectively. Based on the obtained result, the scatter plot shows that LinkedHashSet(H) is the absolute most efficient Collection among the others in terms of memory, CPU, and energy consumption combined, Therefore, it can be recommended as the best choice for Java developers who are interested in the efficiency in terms of these three objectives in case of Contains operation.

Collection HashSet(G) has the second lowest CPU usage and energy consumption. Additionally, it has the lowest memory usage after LinkedHashSet (H) and PriorityQueue (F) respectively. Collection PriorityQueue (F) has roughly high CPU usage and energy consumption. On the other hand, the collections Stack (D), ArrayDeque(B), and Vectorn (E) are the least efficient Collections in terms of memory, CPU, and energy consumption since they have roughly high memory, CPU and energy consumption even though LinkedList (C) has the highest energy consumption among other Collections but it has less memory and CPU usage than the Collections D, B, and E.

According to an efficiency analysis of Java Collections, the time complexity of a collection indicates its efficiency in terms of energy consumption and CPU usage. It has been found that LinkedHashSet (H), HashSet (G), and TreeSet (I) are the most efficient in terms of energy consumption, CPU usage, and memory usage, respectively. The efficiencies of these collections align with their respective time complexity. These are the fastest among all collections analyzed for the 'contains' operation. For this operation, LinkedHashSet, HashSet, and TreeSet have time complexity of O(1), O(1), and O(log n), respectively. Conversely, the "Contains" operation for other collections has a time complexity of O(n), making it less efficient. It is important to note that when selecting a collection for a given task, numerous factors should be taken into consideration. These factors include trade-offs between time complexity, energy consumption, and other relevant considerations such as CPU and memory usage.

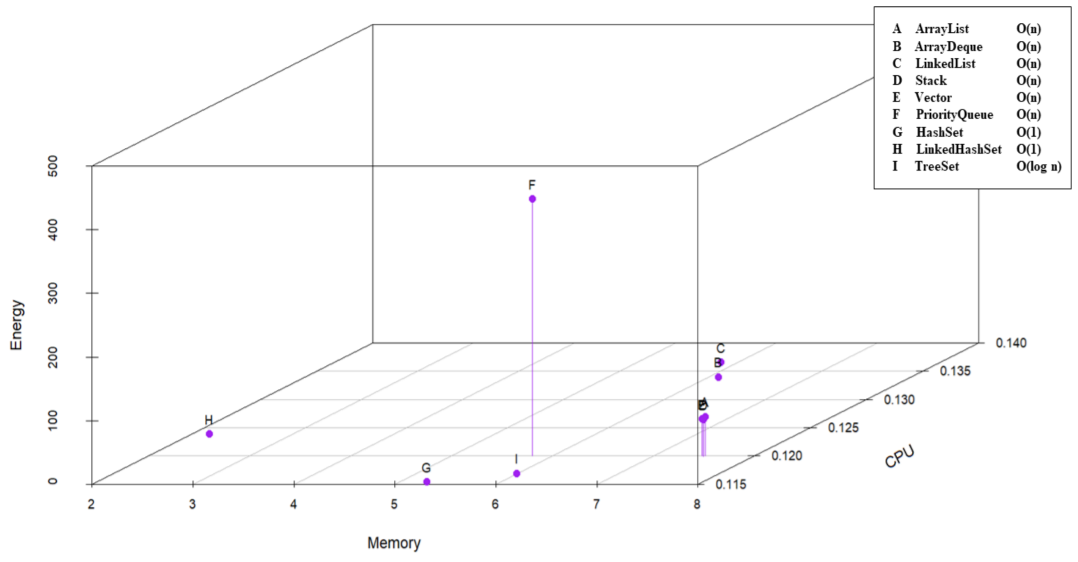

Figure 6.3: Collections 3D scatter plot for Remove

Based on the obtained results for Remove operation, LinkedHashset (H) can be recommended to be the most efficient Collection in terms of three variables, memory, CPU, and energy consumptions combined, even though HashSet (G) has less CPU consumption. LinkedHashSet consumes 6% less energy than HashSet, at the same time, HashSet consumes 7% less CPU than LinkedHashSet but HashSet consumes 243%

more memory than LinkedHashSet, therefore LinkedHashSet is recommended to Java developers if these three factors (memory, CPU, and energy) are considered at once.See **Appendix 1**.

Moreover, if a developer is interested in Collections' efficiency in terms of energy consumption only, ArrayDeque (B) is the best choice, because it is the most efficient Collection among others in terms of energy consumption. In addition, if a developer is interested in CPU consumption efficiency, HashSet is the best choice since it has the lowest CPU consumption among other Collections. On the other hand, PriorityQueue (F) is the least efficient Collection among others in terms of memory, CPU, and energy consumption combined, since it has a very high energy consumption compared with other Collections in the case of Remove operation. Moreover, it has roughly high CPU and memory consumption.

According to an efficiency analysis of Java Collections, the ArrayDeque collection exhibits the highest level of efficiency in terms of energy consumption. Despite its time complexity being O(n), which is less efficient than that of HashSet and LinkedHashSet, both of which have a time complexity of O(1). Among other Collections, LinkedHashSet and HashSet demonstrate the greatest efficiency in terms of memory usage and time complexity. In contrast, TreeSet exhibits inferior memory efficiency compared to both PriorityQueue and LinkedList, despite having a time complexity of O(log n), which is better than PriorityQueue and LinkedList, both of which have a time complexity of O(n). Furthermore, the HashSet collection boasts the most efficient CPU usage with a time complexity of O(1). However, LinkedHashSet's time complexity is O(1), which is generally regarded as superior to O(n). However, there are several Collections that exhibit better CPU efficiency than LinkedHashSet in terms of CPU usage, including the ArrayList, Stack, and PriorityQueue, which have a time complexity of O(n).

6.2.2 Sortings Experiments Analysis

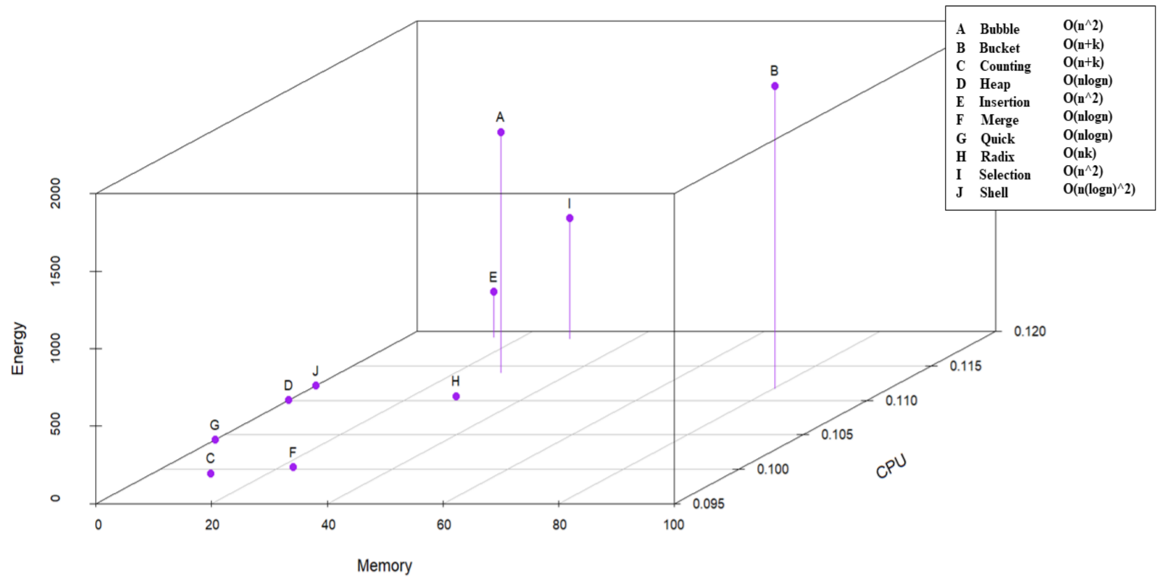

Figure 6.4 Sortings 3D scatter plot

In Figure 6.4, each axis displays one of the measured objectives in each Sorting algorithm. Each data point represents one of the Sorting algorithms based on the gathered data from the experiment. From the scatter plot, it can be inferred that Quick sort (G), Heap sort (D), and Shell sort (J) are the most efficient choices for developers who prioritize high performance in both memory and CPU usage while maintaining low energy consumption; since they consume no additional memory, very low energy, and a low CPU. However, Figure 6.4 provides a variety of options for the developer to select and consider. For instance, if the aim is to have a Sorting algorithm with the lowest energy consumption and the highest performance in CPU usage, regardless of memory usage, Counting sort (C) and Merge sort (F) respectively are the optimal choices. On the other hand, if the aim is to achieve the lowest energy consumption and memory usage, regardless of CPU usage, then Quick sort (G), Heap sort (D), and Shell sort (J) are all the best options and most efficient. These three algorithms consume no additional memory and the lowest energy among the others. Finally, the Bucket sort (B) may not be the most efficient choice due to its high energy consumption and memory usage, as well as its relatively high CPU usage when compared to other Sorting algorithms. Based on the efficiency analyses of Sorting algorithms, it can be inferred whether time complexity is related to the algorithm's efficiency and performance or not.

The Sorting algorithms winners of consuming the least energy, CPU, and memory, have the time complexity, as follows: Quick sort O(nlogn), Heap sort O(nlogn), and

Shell sort O(n(logn)^2), all of which exhibit faster performance compared to the least efficient Sorting algorithm, Bucket sort, which has a time complexity of O(n+k), particularly when processing large datasets.

7 Discussion and Conclusions

The goal of this study is to determine the potential relationship, if any, between energy consumption and commonly performed operations on Java Collections - including addition, search, and removal of elements, as well as widely used Sorting algorithms in Java programming. Additionally, the study aims to identify the most efficient Java Collections and Sorting algorithms based on their energy consumption, CPU, and memory usage. The study employs controlled experiments that involve measuring the energy consumption, CPU, and memory usage of each Java Collection during the specified operations and Sorting algorithms. The conducted analysis for the outcomes of the controlled experiments provides direct answers to the research questions. These answers are anticipated to offer significant insights into the energy efficiency of Java Collections and Sorting algorithms, which could inform future software development strategies and contribute to the reduction of energy consumption and environmental impact in software development.

The present research project is similar to the studies conducted by Hasan et al [11] and Pinto et al [12] in that both studies aim at scrutinizing and quantifying the energy consumption of Java Collections. According to Hasan et al, energy consumption for Java Lists, Maps, and Sets has been examined with regard to various operations, such as insertion, iteration, and random access, highlighting significant variations based on the type of operation. This research project shares similarities and differences with Hasan et al. Both studies involve examining and quantifying the energy consumption of Java Collections and investigating the energy consumption of varied operations. However, the differences between the two studies lie in the fact that Hasan et al explore the energy consumption of diverse operations with different scenarios, such as insertion at the beginning, middle, and end, whereas this project focuses on analyzing the energy consumption, CPU usage, and memory usage of various operations, specifically, insertion, searching, and deletion. Additionally, this project attempts to identify a statistical correlation between each Collection and quality attributes, such as CPU and memory usage. It is noteworthy that the findings of Hasan et al demonstrate that LinkedList has lower energy consumption than ArrayList when inserting elements at the start of a list, which aligns with the efficiency analysis results of the current research project in terms of energy consumption for the insertion operation. This concurrence suggests that LinkedList is a preferable option over ArrayList in terms of energy efficiency for inserting elements at the beginning of a list.

The study conducted by Pinto et al investigates the energy efficiency of 16 commonly used Java Collections, categorized into lists, sets, and mappings, in different operations such as insertion, removal, and traversal on the Tomcat and Xalan systems. The study reveals that newer hash-table implementations can yield significant energy savings. It also finds that non-thread-safe implementations of data collections consume less energy than thread-safe ones and that energy consumption is influenced by different operations. Additionally, the study examines the impact of thread counts, initial capacities, and load factors on the energy consumption of map implementations in Java.

The current research project shares similarities with Pinto et al's study in terms of examining energy consumption in Java collections. In contrast to the investigation by Pinto et al, this study scrutinizes the energy consumption of Java Collections on a singular device and concentrates on Collections that are embedded in the Java Collection framework, intended for performing insertion, search, and removal operations. Moreover, this study assesses the energy consumption associated with Collections in terms of their quality attributes and explores the statistical relationship between these attributes, and does not confine its focus solely to Java thread-safe Collections and does not extend its experiments to multiple systems, which distinguishes it from the study conducted by Pinto et al.

Additionally, this research project is also similar to the study conducted by Alves et al. [3], which examined and analyzed the energy consumption of some Sorting algorithms. Alves's main investigation revealed that the Selection sort algorithm consumes the least energy, while the Bubble sort algorithm consumes the most energy. It is noteworthy that the findings of this research, as presented in Section 6.2.2, align with the observation that the Selection sort algorithm consumes a significant amount of energy when compared to the Bubble sort. However, Alves's area of investigation focuses on three classic Sorting algorithms only and does not take into account any quality attributes while measuring energy consumption. In contrast, this study analyzed additional Sorting algorithms and their energy consumption in relation to some quality attributes, such as memory and CPU usage. Consequently, the findings suggest that the Bucket sort algorithm is the least efficient and worst Sorting algorithm.

In conclusion, this paper has provided answers to the research questions presented in Section 1.3 and focuses on estimating the energy consumption, memory, and CPU usage of several Java Collections and Sorting algorithms. The study conducted controlled experiments and employed tools such as JoularJX to measure energy consumption, and RStudio to implement a statistical analysis using R to find out the relationship between energy consumption and quality attributes. Moreover, an efficiency analysis is conducted, and its findings are then represented using a 3D scatter plot to compare the mean values of energy consumption, CPU, and memory usage for each operation and algorithm.

Regarding Java Collections, the examination reveals a complex landscape in terms of efficiency across various operations, such as Add, Contains, and Remove. It becomes evident that singling out a universally superior Collection(s) is an unattainable goal. Notably, LinkedList stands out for its exceptional energy efficiency during Add operations, but its performance falters significantly when it comes to Contains operations. Therefore, a comprehensive evaluation considering the specific requirements and trade-offs of each operation is essential to determine the most suitable Collection(s) for a given context.

Moreover, in addition operation, ArrayLists, ArrayDeque, and LinkedLists are recommended when memory, CPU, and energy consumption are all considered together. Notably, LinkedList is the optimal choice when only energy consumption is considered during addition. In terms of memory, CPU, and energy consumption, LinkedHashSet has the highest efficiency level when used for searching operations. If memory, CPU, and energy consumption are considered as a single unit, LinkedHashSet is the most efficient collection when used for removing. ArrayDeque, however, is the most energy-efficient choice for Java developers when it comes to the removal of data. Concerning Sorting algorithms, Quick sort, Heap sort, and Shell sort were identified as the most efficient in terms of energy consumption, CPU, and memory usage. Bucket sort was found to be less efficient due to its higher consumption and usage in all terms.

This study has been conducted as the outcome of all those experiments' results and analyses covering effectively the research gap on which it is based. Additionally, the study highlights the need for developers to consider multiple factors when selecting a Collection or Sorting algorithm, emphasizing that there may be a lack of correlation between energy consumption and quality attributes (CPU and memory usage) in certain algorithms and Collections. The study also points out that time complexity does not always reflect efficiency in terms of energy consumption, CPU, and memory usage. Thus, the study underscores the importance of optimizing software for energy consumption, particularly in resource-constrained environments, and recommends using controlled experiments and appropriate methodologies to guide programmers in developing energy-efficient software programs which leads the software engineering field to be improved.

Lastly, based on the findings of the conducted statistical analysis, it appears that there is a limited possibility of generalizing the correlation and relationship between energy consumption and CPU and memory usage across various Collection and Sorting algorithms in most cases. Specifically, the observed P-value is greater than 0.05, indicating that the results are not statistically significant. Therefore, further investigation is warranted as a part of future research endeavors to better understand this relationship.

7.1 Future Work

This section provides suggestions for future research to advance the current study. Time and resources are factors that limit this bachelor's project. Hence, some aspects of this study may benefit from additional time and resources to be executed optimally. This section outlines what was discovered during the project’s work, areas that can be improved, and recommendations for future work.

As mentioned earlier, when statistical analysis has been conducted, it has been determined that the ability to generalize the correlation and relationship between energy consumption, CPU, and memory usage for Collection and Sorting algorithms is limited in most cases. It is assumed and suggested that further examination of the P-value is necessary. It could be investigated by increasing the input size and the number of experiments significantly, for each operation and algorithm.

Moreover, future work considering different tools for measuring energy consumption and conducting experiments on different operating systems can help increase the generalizability of the statistical analysis results. Different operating systems have different features and characteristics; therefore, experimenting on multiple operating systems can help evaluate how generalizable the findings are while ensuring that the findings are robust and not specific to a particular operating system. Worth mentioning is that this study raised the hypothesis that there might not be significant differences in the results when comparing the implementation of Sorting algorithms to Java library built-in algorithms in terms of the most efficient algorithm regarding energy consumption; however, to validate this hypothesis, further investigation comparing the implementation results with the Java library results is recommended.

A possible approach to advance this research could be done using machine learning. Using machine learning can help in developing predictive models that can forecast energy consumption based on different quality attribute metrics. Additionally, assist in discovering patterns and trends in the data that may be difficult to detect using typical statistical methods. Machine learning can provide powerful tools for understanding and analyzing the relationships between energy consumption and quality attributes for Java Collections and Sorting algorithms. Applying these tools makes it possible to gain deeper insights into the performance characteristics of different operations and algorithms to discover the most optimal energy consumption in software development.

References

[1] Mykhailo Spirich, “Is Java still relevant in 2023,” *Axon*, 15-Jan-2023. [Online]. Available: https://www.axon.dev/blog/is-java-still-relevant-in-2022.

[2] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, “What do programmers know about software energy consumption?” IEEE Software, vol. 33, no. 3, pp. 83–89, 2016.

[3] D. S. Alves, O. A. Ferreira, L. M. Duarte, D. Silva, and P. H. Maia, “Experiments on model-based software energy consumption analysis involving Sorting algorithms,” *Revista de Informática Teórica e Aplicada*, vol. 27, no. 3, pp. 72–83, 2020.

[4] R. Pereira *et al.*, “Energy efficiency across programming languages: How do energy, time, and memory relate?,” *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 2017. doi:10.1145/3136014.3136031

[5] “Java Collections framework,” Programiz. [Online]. Available: https://www.programiz.com/java-programming/Collections.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms. Cambridge (Inglaterra): Mit Press, 2009.

[7] “Java development kit version 17 API specification,” *Arrays (Java SE 17 & JDK 17)*, 12-Dec-2022. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html.

[8] R. Sedgewick and K. Wayne, “Algorithms, 4th Edition,” *Princeton University*. [Online]. Available: https://algs4.cs.princeton.edu/home/.

[9] S. Ergasheva, Z. Kholmatova, A. Kruglov, G. Succi, X. Vasquez, and E. Zuev, “Analysis of energy consumption of software development process entities,” Electronics, vol. 9, no. 10, p. 1678, 2020.

[10] H. Makabee, “Conference talk – Hayim Makabee on software quality attributes,” *Effective Software Design*, 24-Nov-2014. [Online]. Available: https://effectivesoftwaredesign.com/2014/11/20/conference-talk-hayim-makabee-on-software-quality-attributes/.

[11] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, “Energy Profiles of java Collections classes,” Proceedings of the 38th International Conference on Software Engineering, 2016.

[12] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, “A comprehensive study on the energy efficiency of Java’s thread-safe Collections,” 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016.

[13] Saborido, R., Morales, R., Khomh, F., Guéhéneuc, Y.-G., &amp; Antoniol, G. (2018). Getting the most from map data structures in Android. Empirical Software Engineering, 23(5), 2829–2864. https://doi.org/10.1007/s10664-018-9607-8.

[14] Oliveira, W., Oliveira, R., Castor, F., Fernandes, B., &amp; Pinto, G. (2019). Recommending energy-efficient Java Collections. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). https://doi.org/10.1109/msr.2019.00033.

[15] P. D. Anne Marie Helmenstine, “What is a controlled experiment?,” ThoughtCo, 11-Dec-2019. [Online]. Available: https://www.thoughtco.com/controlled-experiment-609091.

[16] Wohlin, C., Runeson, P., Höst Martin, Ohlsson, M. C., Regnell Björn, &amp; Wesslén Anders. (2012). Experimentation in software engineering: An introduction. Springer-Verlag New York.

[17] “IntelliJ IDEA – the leading Java and Kotlin Ide,” JetBrains. [Online]. Available: https://www.jetbrains.com/idea/.

[18] Joularjx. (n.d.). Retrieved October 27, 2022, from https://www.noureddine.org/research/joular/joularjx.

[19] “R & R studio,” *Statistics*. [Online]. Available: https://statistics.byu.edu/r-r-studio.

[20] S, R. A. (2023, February 13). *Collections in Java and how to implement them? [updated]*. Simplilearn.com. Retrieved March 22, 2023, from https://www.simplilearn.com/tutorials/java-tutorial/java-collection.

[21] D. Loshin and S. Lewis, “What are data structures? - definition from whatis.com,” SearchDataManagement, 09-Mar-2021. [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/data-structure.

[22] “What is linear regression?” Statistics Solutions, 10-Aug-2021. [Online]. Available: https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/what-is-linear-regression/

[23] About linear regression. IBM. (n.d.). Retrieved November 13, 2022, from https://www.ibm.com/se-en/topics/linear-regression

[24] Hexanovate, “What is Time & Space Complexity in data structure? Tap Academy,” *Tap Academy*, 18-Feb-2023. [Online]. Available: https://thetapacademy.com/time-and-space-complexity-in-data-structure/

[25]“Big O notation cheat sheet: Data structures and algorithms,” Flexiple, https://flexiple.com/algorithms/big-o-notation-cheat-sheet/

[26] *What is CPU usage? - it glossary*. SolarWinds. (n.d.). Retrieved March 22, 2023, from https://www.solarwinds.com/resources/it-glossary/what-is-cpu

[27] baeldung, W. by: (2020, June 27). *Java heap space memory with the runtime API*. Baeldung. Retrieved March 22, 2023, from https://www.baeldung.com/java-heap-memory-api

[28] *Java collection interface*. Programiz. (n.d.). Retrieved March 22, 2023, from https://www.programiz.com/java-programming/collection-interface

[29] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and algorithms in Java*. Hoboken, NJ: Wiley, 2014.

[30] “What is a vector in Java?,” *Educative*. [Online]. Available: https://www.educative.io/answers/what-is-a-vector-in-java.

[31] S. Woltmann, “Java arraydeque (with example),” HappyCoders.eu, 07-Jun-2022. [Online]. Available: https://www.happycoders.eu/algorithms/arraydeque-java/.

[32] Mansi, “What is Hashset in Java?,” *Scaler Topics*, 13-Sep-2022. [Online]. Available: https://www.scaler.com/topics/hashset-in-java/.

[33] W. by: baeldung, “A guide to linkedhashset in Java,” *Baeldung*, 13-Dec-2022. [Online]. Available: https://www.baeldung.com/java-linkedhashset.

[34] “PH717 module 9 - correlation and regression,” *The Correlation Coefficient (r)*. [Online]. Available: https://sphweb.bumc.bu.edu/otlt/MPH-Modules/PH717-QuantCore/PH717-Module9-Correlation-Regression/PH717-Module9-Correlation-Regression4.html.

[35] C. Thieme, “Understanding linear regression output in R,” *Medium*, 16-Jun-2021. [Online]. Available: https://towardsdatascience.com/understanding-linear-regression-output-in-r-7a9cbda948b3.

[36] “Standard deviation in R: Methods to calculate standard deviation in R,” *EDUCBA*, 27-Mar-2023. [Online]. Available: https://www.educba.com/standard-deviation-in-r/.

[37] *ThreadMXBean (Java Platform SE 8)*, 05-Apr-2023. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/management/ThreadMXBean.html#getCurrentThreadCpuTime--.

[38] *System (java platform SE 8)*, 05-Apr-2023. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--.

[39] *OperatingSystemMXBean (java platform SE 8)*, 05-Apr-2023. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/management/OperatingSystemMXBean.html#getAvailableProcessors--.

[40] Runtime (Java Platform SE 7). (2020, June 24). Retrieved November 24, 2022, from https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html.

Appendix 1

https://github.com/MustafaAlsaid2022/EstimateEnergyConsumptionOfJavaPrograms

Appendix 2

	Energy - Memory		Energy - CPU	
Sorting Algorithm	**Correlation - coefficient**	**P-Value**	**Correlation - coefficient**	**P-Value**
Bubble Sort	-0.1521349	0.131	-0.03776931	0.709
Bucket Sort	0.1879717	0.0611	0.09919184	0.326
Counting Sort	-0.06486479	0.521	-0.09513352	0.346
Heap Sort	-	-	0.3741691	0.000126
Insertion Sort	-0.4640469	1.157e-06	0.1799327	0.0732
Merge Sort	-0.1041293	0.303	0.0201857	0.842
Quick Sort	-	-	-0.02678887	0.7913
Radix Sort	-0.1803464	0.0726	0.05118075	0.613
Selection Sort	0.9122504	2.2e-16	-0.02620743	0.7958
Shell Sort	-	-	-0.2078286	0.038

		Energy - Memory		Energy - CPU	
Collections		**Correlation - coefficient**	**P-Value**	**Correlation - coefficient**	**P-Value**
ArrayList	Add	0.1055173	0.296	-0.1214094	0.229
	Contains	0.4525864	2.27e-06	-0.08360846	0.408
	Remove	0.01856971	0.85450	-	-
	Add	-0.02557409	0.801	0.4576277	1.69e-06

ArrayDeque	Contains	0.06172995	0.542	-	-
	Remove	-0.2785587	0.00501	0.3767748	0.000112
LinkedList	Add	0.01275436	0.9	0.2144147	0.0322
	Contains	0.08315959	0.411	-0.07822042	0.439
	Remove	0.03188009	0.753	0.4313321	7.46e-06
Stack	Add	-0.07347836	0.468	0.1342463	0.183
	Contains	-0.04464102	0.659	-	-
	Remove	-0.3809132	9.24e-05	0.5528358	2.46e-09
Vector	Add	0.04854247	0.632	0.120845	0.23105
	Contains	-0.1306377	0.195	-	-
	Remove	0.0253143	0.8026	-	-
PriorityQueue	Add	0.06130691	0.545	-0.09562125	0.34396
	Contains	-0.1024608	0.31	-0.1601259	0.112
	Remove	0.7191587	<2e-16	-	-
HashSet	Add	-0.004342602	0.966	0.01278713	0.89952
	Contains	-0.24648	0.0134	0.2793574	0.00488
	Remove	-0.5002826	1.16e-07	0.007478494	0.941
LinkedHashset	Add	-0.1484179	0.141	-0.005122781	0.959657
	Contains	-0.1693141	0.09217	0.241253	0.0156
	Remove	0.1476215	0.143	0.2810776	0.00461
TreeSet	Add	-0.05807084	0.566	0.09112543	0.367
	Contains	-0.1108526	0.272	0.06880524	0.496

	Remove	-0.1657397	0.0994	0.0134883	0.894