On the Energy Efficiency of Sorting Algorithms

Simão Cunha
Universidade do Minho
Braga, Portugal
a93262@alunos.uminho.pt

Luís Silva
Universidade do Minho
Braga, Portugal
pg50564@alunos.uminho.pt

Gonçalo Pereira
Universidade do Minho
Braga, Portugal
a93168@alunos.uminho.pt

Abstract

The purpose of this report is to document all the work done in the context of the project of the course *Experimentação em Engenharia de Software* (Empirical Methods in Software Engineering) that belongs to the Masters profile Software Development, Validation and Maintenance from University of Minho.

This project occurs due to the different performances of several languages that implements the same sorting algorithms and, in order to create an language ranking, we will monitor and measure 3 implementations of sorting algorithms expressed in 13 languages - in some cases, we will test the difference with the compilation and the interpretation metrics of the same language.

We performed a statistic analysis thanks to some Python libraries and multi-criteria optimization, which was a workshop given by Luís Paquete from University of Coimbra [Paquete 2023]. With this study we where observed that among the Sorthing Algorithms we chose, QuickSort proved to be the best. Altough C was the fastest language, our multi-criteria research showed us that there are also other powerfull languages that are as good as C.

Keywords Green Software, Programming Languages, Sorting Algorithms, RAPL

1 Introduction

At the moment, more than ever, there is an important concern about energy consumption among computer manufacturers, programmers and regular computer users - everybody aims to have the maximum performance while minimizing energy consumption, either through the use of low-power hardware components or software that accomplishes tasks efficiently with varying energy requirements [Pereira, R. *et al.* 2020]. This has led to the emergence of the concept of Green Software. According to [Bener, A. *et al.* 2014], *Greening in software aims to reduce the environmental impact caused by the software itself. [...] Green specifications provide a way to indicate a service's carbon footprint and eventually specify operational constraints to allow more flexibility during service provisioning.*

Therefore, we created a programming language ranking according to several factors such as energy consumption, performance and memory consumption, while they are executing sorting algorithms. In this study, we use 13 programming languages, 3 sorting algorithms, the Running Average Power Limiting (RAPL) interface from Intel [Hahnel, M. *et al.* 2012], multicriteria optimization techniques [Paquete 2023] and data analysis with libraries from Python.

Perfil ES'23, 2023
2023.

2 Methodology

2.1 Implementations of Sorting Algorithms

As already said before, we used 13 programming languages: C, C++, C#, Python, Java, Rust, Ruby, Kotlin, Haskell, Prolog, PHP, JavaScript and Go - we also tested a Python compiller called Codon [Python compiller Codon], so in fact we will have 14 different languages in our study. We chose this programming language because these are the languages that we were confronted in our university degree:

- Haskell: presented in Functional Programming (1st year)
- C: presented in Imperative Programming (1st year)
- Java: presented in Object Oriented Programming (2nd year)
- Prolog: presented in Artificial Intelligence (3rd year)
- PHP, JavaScript and C#: presented in Informatic Labs IV (3rd year)
- Python: presented in Language Processing (3rd year)
- C++: presented in Computer Graphics (3rd year)
- Ruby: presented in Cloud Computing Applications and Services(4th year)
- Kotlin: presented in Topics of Software Development (4th year)

In adition, we add Go to our measurements because it was a language that we consider that has a clean syntax and seems easy to learn. We also added a Python compiller called Codon to compare directly with the Python interpreted program and see some differences between the compilation and the interpretation of the same programming language.

We choose 3 sorting algorithms: Bubble sort, Quick sort and Selection Sort. As seen in [Sorting algorithms complexity], we choose 3 algorithms - whose implemetations were obtained from ChatGPT and [GeeksForGeeks website 2023] - with different time complexities in the best case:

- Bubble sort: $\omega(n)$
- Quick sort: $\omega((nlog(n))$
- Selection sort: $\omega(n^2)$

2.2 Energy Consumption Monitorization

In order to monitorize energy consumption of the sorting algorithms implemented in all 13 programming languages, we used the Running Average Power Limiting (RAPL) interface from Intel [Hahnel, M. *et al.* 2012].

As said in [Zhang, Z. *et al.* 2021], RAPL was introduced since Sandy Bridge microarchitecture. It enables accurate energy consumption measurement and provides fine-grained power capping capability. Thus, by monitoring and reacting to the power consumption of computing, RAPL has been widely used in data centers to improve energy efficiency and enforce power budget compliance. In the beginning of the semester, it was presented by the teachers the RAPL interface in C [Vince Weaver: RAPL interface in C]. However, the last update of this interface was in 2015, so, in the University of Minho, this was updated to support the latest Intel's new architectures and refactored its source code in order to provide a modular tool in C for energy consumption measurement - it uses a system call to execute the binary.

Therefore, this interface creates a .J file with the results obtained from the execution of the RAPL:

- Program: Name of the program executed;
- Package: It measures the energy consumption of the entire socket. It includes the consumption of all the cores, integrated graphics and also the uncore components (last level caches, memory controller);
- Core(s): the energy consumed by all cores and caches;
- GPU: the energy consumed by the GPU;
- DRAM?: this domain estimates the energy consumption of the random access memory (RAM);
- Time (sec): the execution time of the program

In order to know the cores temperature, we will use the lm-sensors library [lm-sensors 2021]. Since every computer might have different number of cores, we will take in consideration the average temperature in all cores and we measure this value while our RAPL adaption is running the sorting algorithm implemented in a certain programming language.

2.3 Limiting the Energy Consumption with PowerCap

In order to effectively limit the energy consumption of the processor during script execution, we employed the PowerCap library [PowerCap 2023]. This powerful tool allows us to exert fine-grained control over the power usage of the processor. By specifying a range of limits, from as low as 5 to as high as 1000 Watts, we can precisely manage and optimize the energy consumption.

PowerCap operates by constantly monitoring and managing the power consumption of the processor in real-time. It ensures that the processor's energy usage remains within the specified limits, preventing excessive power draw and potential wastage. By actively controlling the power cap, we can strike a balance between energy efficiency and script execution performance.

With the help of the PowerCap library, we can achieve significant energy savings while maintaining the desired functionality of the scripts. This not only contributes to a more sustainable computing environment but also enables us to optimize power usage in scenarios where energy conservation is crucial, such as in portable devices or data centers.

2.4 Getting the development cost

Since there is a concern about Green Software, we need to strike a balance between energy consumption and software development cost. While it's important to have software that consumes fewer Joules, nobody wants it to be excessively expensive to develop. So, in order to compare its development cost (or an estimation), we will use the scc [sloc cloc and code 2023], since it has support to all the programming languages we used in this study.

2.5 Obtaining the memory usage by each algorithm

Considering the memory usage of a particular program is crucial for Green Software. Higher memory usage leads to increased hardware and software energy consumption, which has a negative impact on the environment. Therefore, to determine the memory usage of each sorting algorithm, we will use the "maximum resident" provided by the Linux time command. It refers to the maximum amount of resident memory (in KBytes) used by a process during its execution. It represents the peak memory usage throughout the execution of the command.

2.6 Dataset Creation

After deciding on the metrics to be used in this study, we created a bash script to consolidate all the RAPL measures into a single CSV file (comprising approximately 31,500 measures!!!). In addition to the existing metrics, we will also include new columns such as:

- Language: Name of the programming language;
- Program: Name of the sorting algorithm;
- PowerLimit: Value of the PowerCap applied (in Watts);
- Size: Size of the array used in the sorting algorithm;
- Cost: Value of the development cost (in $);
- Time: Execution time (now measured in miliseconds);
- Temperature: Mean temperature in all cores (in ºC);
- Memory: Peak memory usage throughout the execution of the command (in KBytes)

So, the script works as follows:

- Compile the script that calcules the mean temperature in all cores (`sensors.c`);
- Throw a 5 minutes sleep of the main thread in order to cooldown the computer and register the value of the temperature with the help of sensors (`temperatureUpdate.py`)

- Update the number of tests to be done with each algorithm (`ntimesUpdate.py`)
- In a nested loop, we run the algorithm by applying a powercap value - we used 5, 10, 20, 50 and 1000 and giving it a size value to the array - we will use 1000, 2500 and 5000.

At the end, we will have the following dataset - we'll only represent de first 10 lines:

```
Language,Program,PowerLimit,Size,Cost,Package,Core,GPU,DRAM,Time,Temperature,Memory
C ,BubbleSort ,5, 1000,1016, 0.030090 , 0.016174 , 0.002991 , 0.000000 , 9 , 45.3 , 904
C ,BubbleSort ,5, 1000,1016, 0.032410 , 0.017456 , 0.005005 , 0.000000 , 10 , 45.3 , 904
C ,BubbleSort ,5, 1000,1016, 0.034790 , 0.016541 , 0.004944 , 0.000000 , 9 , 45.0 , 900
C ,BubbleSort ,5, 1000,1016, 0.039490 , 0.020203 , 0.007812 , 0.000000 , 10 , 45.0 , 904
C ,BubbleSort ,5, 1000,1016, 0.031921 , 0.015259 , 0.002319 , 0.000000 , 9 , 45.0 , 904
C ,BubbleSort ,5, 1000,1016, 0.038147 , 0.017029 , 0.005249 , 0.000000 , 9 , 45.0 , 904
C ,BubbleSort ,5, 1000,1016, 0.027954 , 0.015869 , 0.001770 , 0.000000 , 9 , 45.0 , 904
C ,BubbleSort ,5, 1000,1016, 0.024109 , 0.020081 , 0.000000 , 0.000000 , 8 , 45.0 , 904
C ,BubbleSort ,5, 1000,1016, 0.032532 , 0.015869 , 0.002380 , 0.000000 , 9 , 45.0 , 904
```

Figure 1. Dataset created

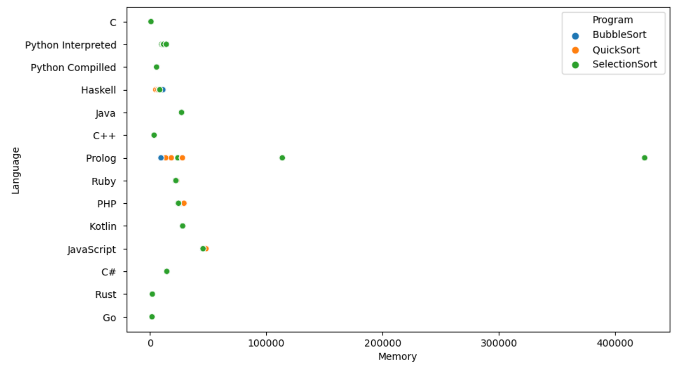


Figure 3. Memory per language and per algorithm

3 Benchmarking the Performance of the Implementation of Sorting Algorithms

In order to benchmark the performance of all sorting algorithms of all programming languages, we will run the bash script in the following computer:

Operating System	Linux Ubuntu 22.04
Processor	Intel(R) Core(TM) i7-8750H
Clockspeed	2.2 GHz
Turbo Speed	4.1 GHz
Cores	6
Threads	12
RAM	16GB
Cache Size	L1: 384K, L2: 1.5MB, L3: 9MB

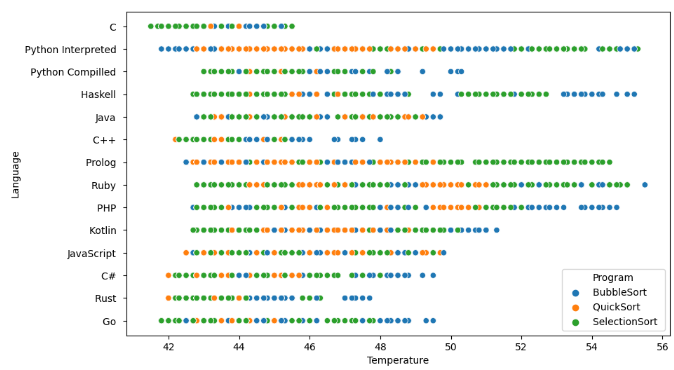


Figure 4. Temperature per language and per algorithm

3.1 Data Visualization - All Features

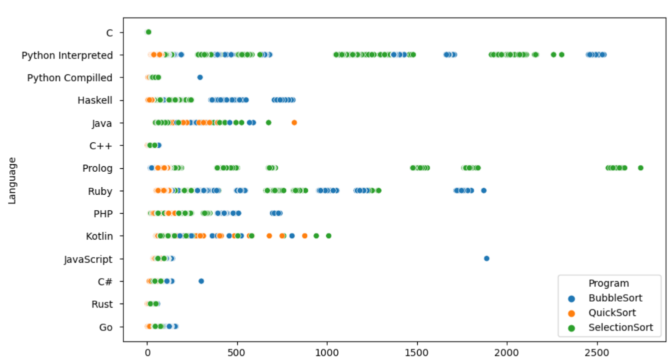


Figure 2. Time per language and per algorithm

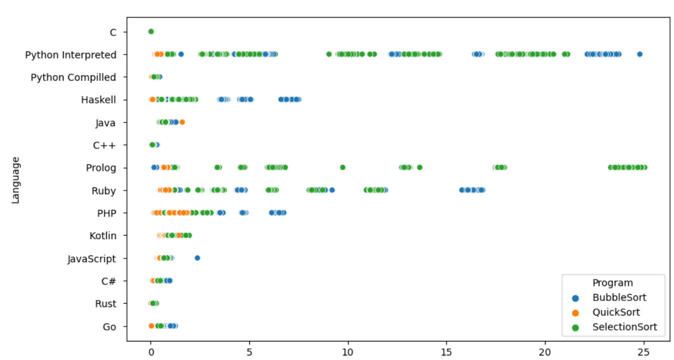


Figure 5. Package per language and per algorithm

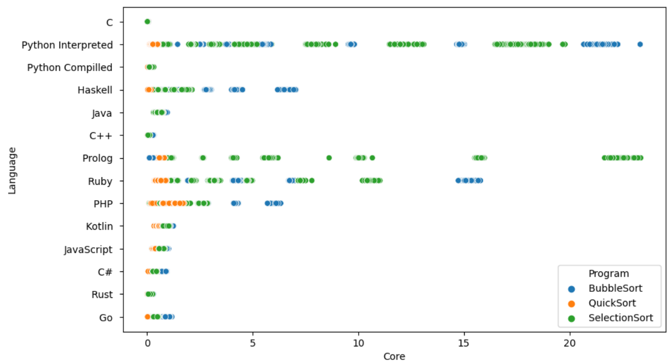


Figure 6. Core per language and per algorithm

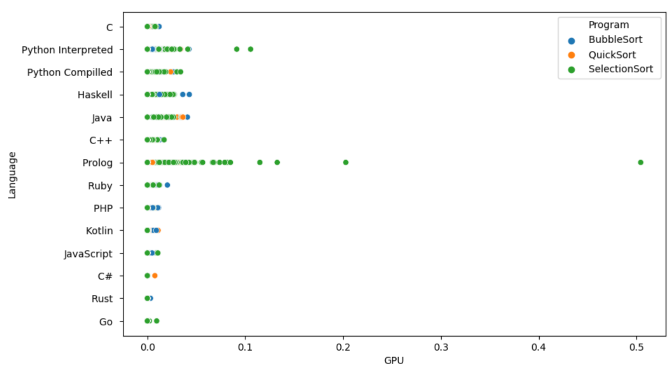


Figure 7. GPU per language and per algorithm

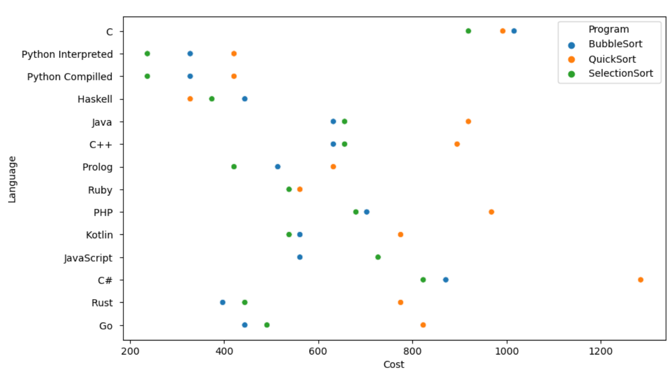


Figure 8. Development cost per language and per algorithm

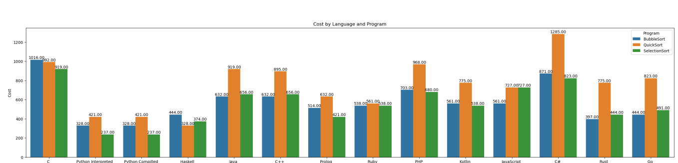


Figure 9. Development cost for each algorithm and for each language

3.2 Powercap Influence For Each Sorting algorithm

3.2.1 By Time (Size = 5000)

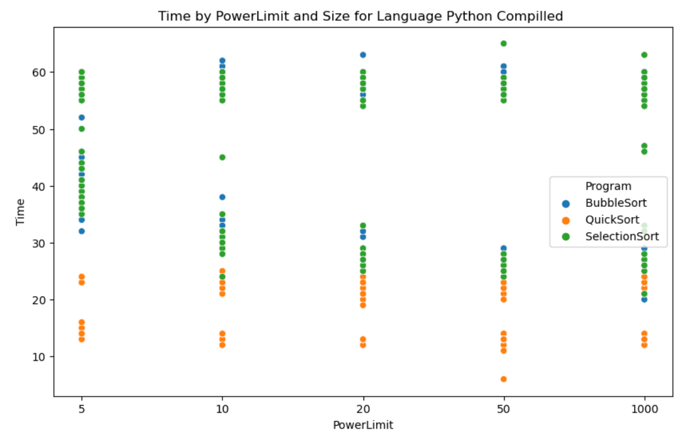


Figure 10. Time by PowerLimit and Size for Python Compilled

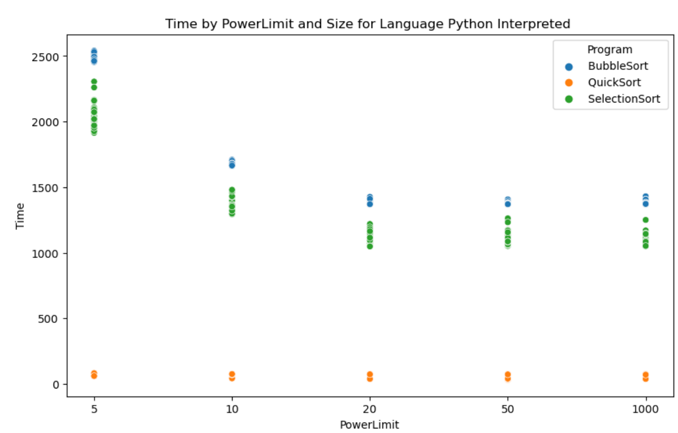


Figure 11. Time by PowerLimit and Size for Python Interpreted

3.2.2 By package (Size = 5000)

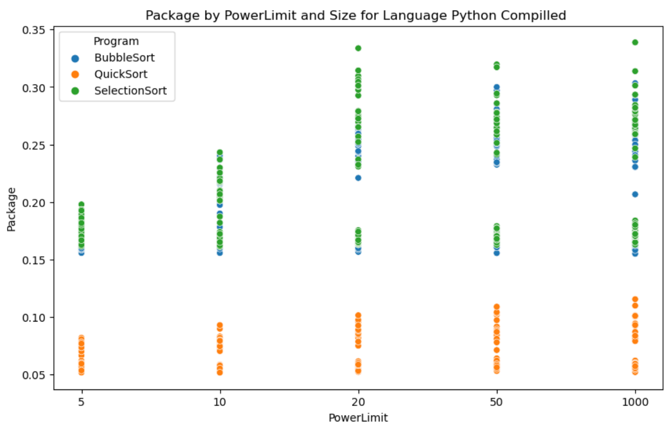


Figure 12. Package by PowerLimit and Size for Python Compilled

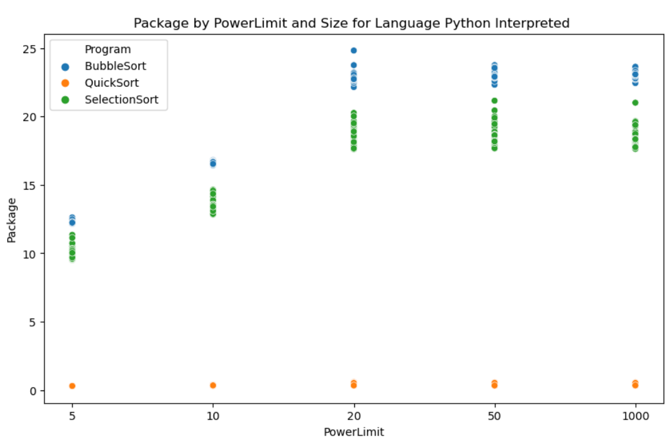


Figure 13. Package by PowerLimit and Size for Python Interpreted

3.3 By memory (Size = 5000)

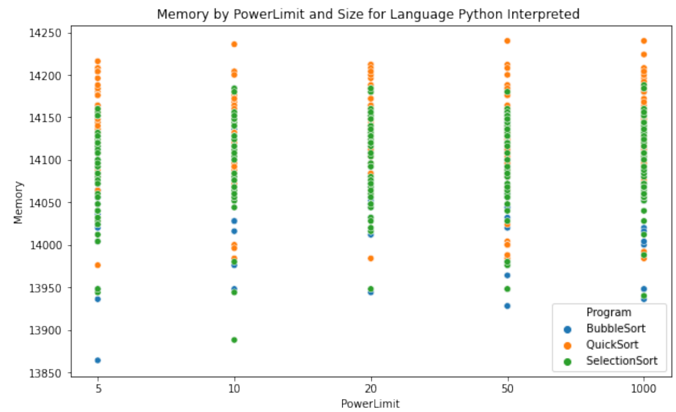


Figure 14. Memory by PowerLimit and Size for Python Interpreted

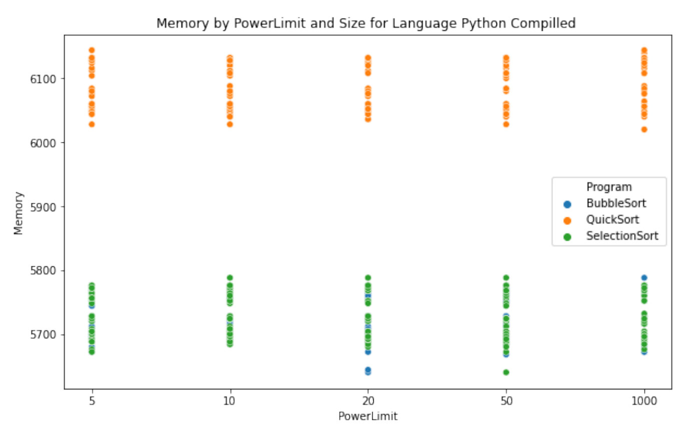


Figure 15. Memory by PowerLimit and Size for Python Compilled

3.4 Powercap Influence For Each Programming Language (Algorithm = Quick Sort, Size = 5000)

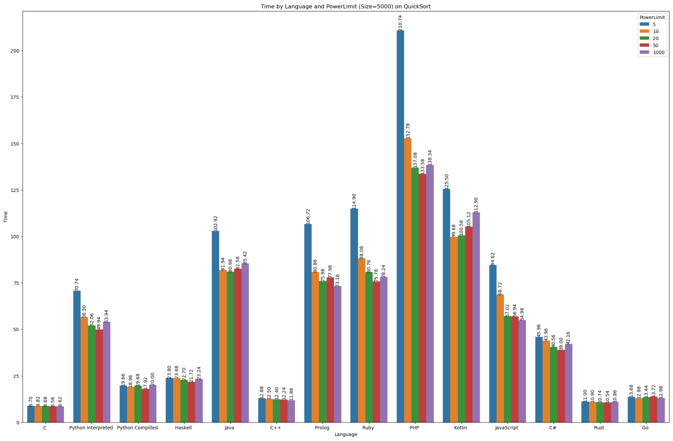


Figure 16. Time by Language and PowerLimit (Size=5000) on QuickSort

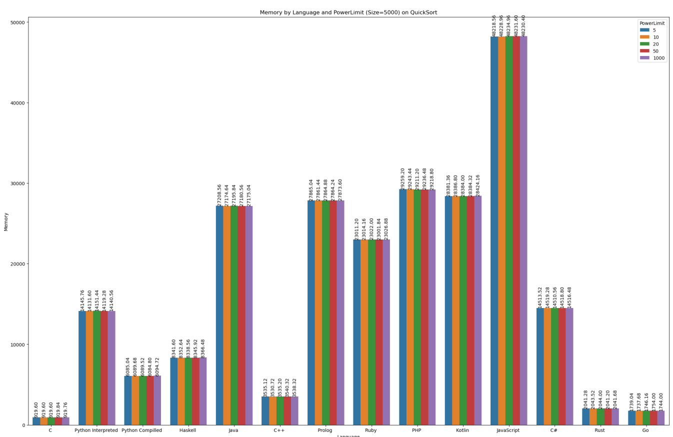


Figure 17. Memory by Language and PowerLimit (Size=5000) on QuickSort

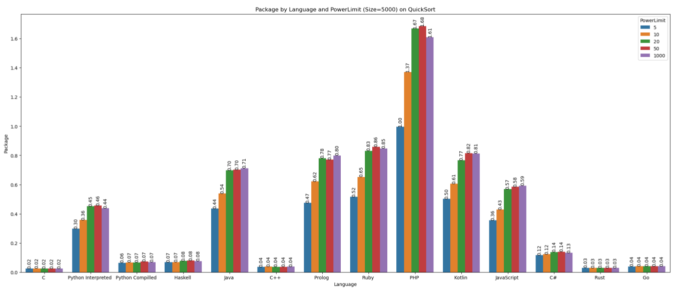


Figure 18. Package by Language and PowerLimit (Size=5000) on QuickSort

3.5 Size Influence For Each Algorithm And For Each Language

3.5.1 By time

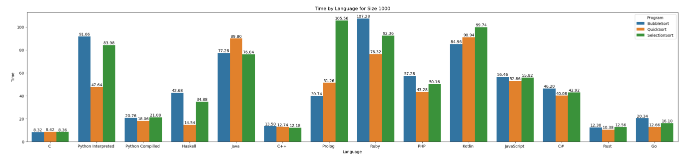


Figure 19. Time by Language for Size 1000

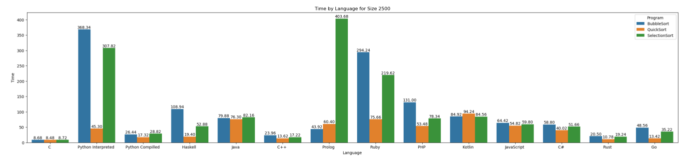


Figure 20. Time by Language for Size 2500

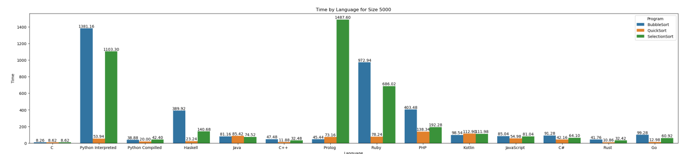


Figure 21. Time by Language for Size 5000

3.5.2 By Memory

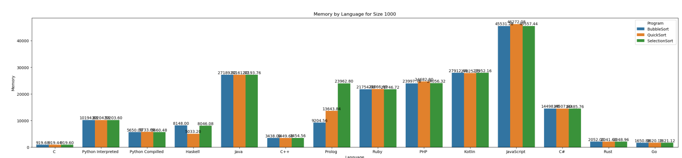


Figure 22. Memory by Language for Size 1000

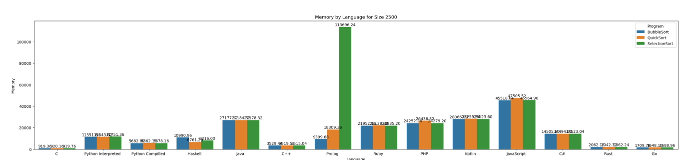


Figure 23. Memory by Language for Size 2500

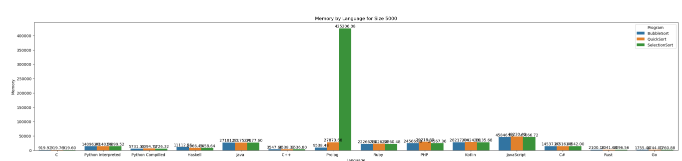


Figure 24. Memory by Language for Size 5000

3.5.3 By Package

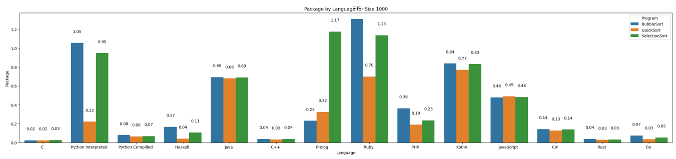


Figure 25. Package by Language for Size 1000

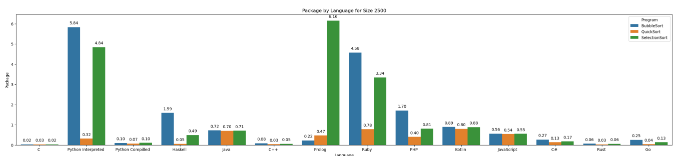


Figure 26. Package by Language for Size 2500

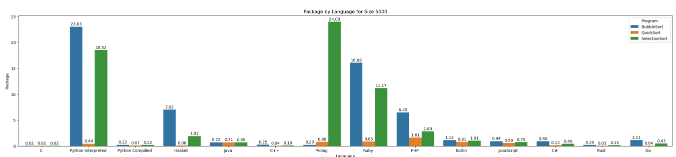


Figure 27. Package by Language for Size 5000

3.5.4 By Temperature

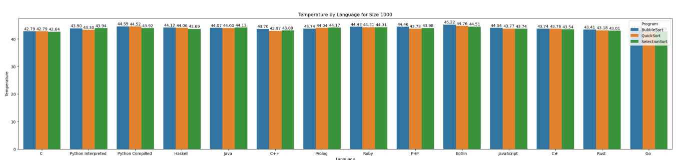


Figure 28. Temperature by Language for Size 1000

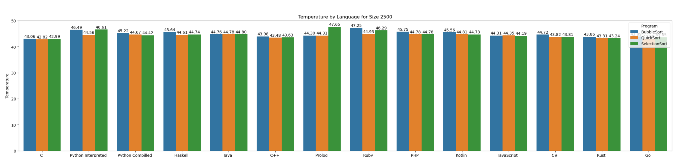


Figure 29. Temperature by Language for Size 2500

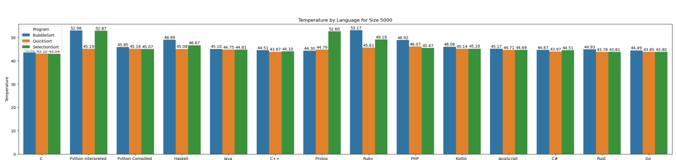


Figure 30. Temperature by Language for Size 5000

3.6 Clustering

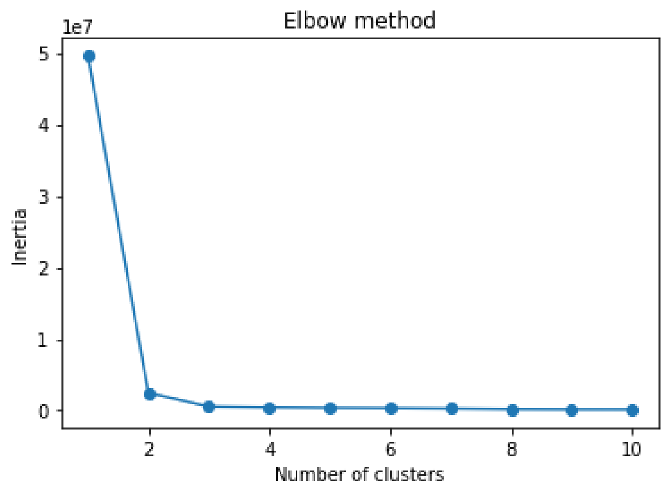


Figure 31. Elbow method

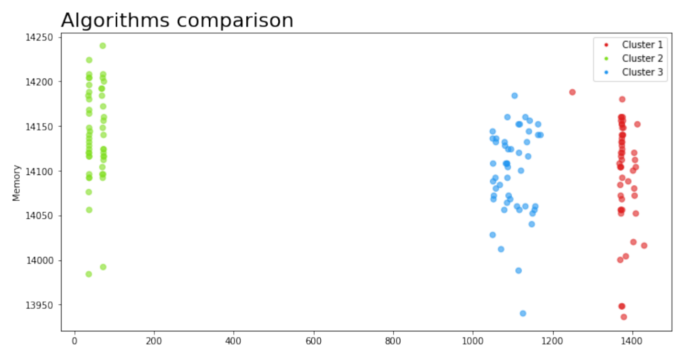


Figure 32. K-means clustering algorithm

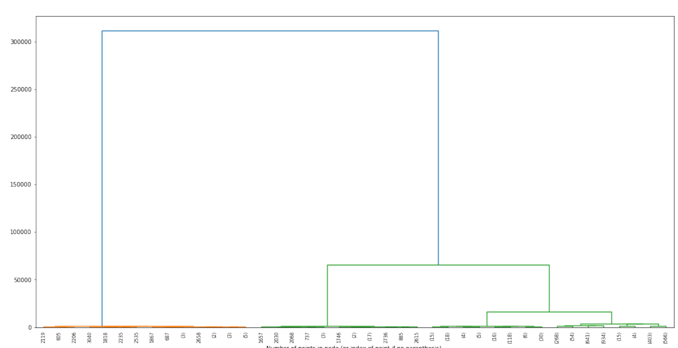


Figure 33. Hierarchical clustering

3.7 Decision Tree - Machine Learning Model

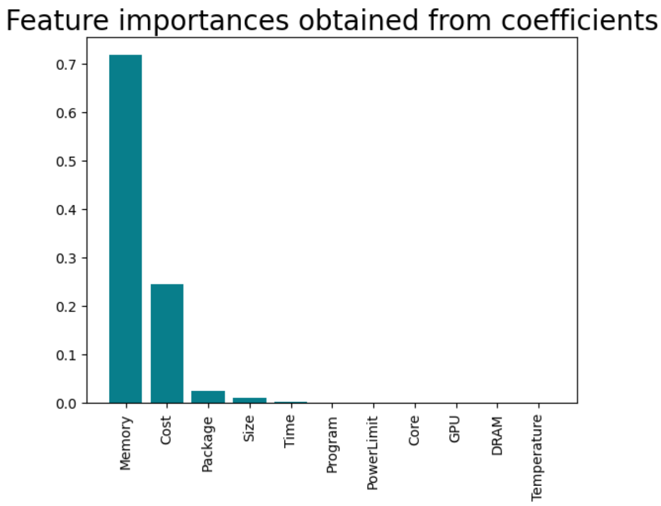


Figure 34. Feature importance

```
Classification report for training data
                    precision    recall  f1-score   support

                 C       1.00      1.00      1.00      1714
                C#       1.00      1.00      1.00      1695
               C++       1.00      1.00      1.00      1689
                Go       1.00      1.00      1.00      1691
           Haskell       1.00      1.00      1.00      1698
              Java       1.00      1.00      1.00      1648
        JavaScript       1.00      1.00      1.00      1687
            Kotlin       1.00      1.00      1.00      1693
               PHP       1.00      1.00      1.00      1688
            Prolog       1.00      1.00      1.00      1699
   Python Compilled       1.00      1.00      1.00      1712
Python Interpreted       1.00      1.00      1.00      1707
              Ruby       1.00      1.00      1.00      1642
              Rust       1.00      1.00      1.00      1662

          accuracy                           1.00     23625
         macro avg       1.00      1.00      1.00     23625
      weighted avg       1.00      1.00      1.00     23625


Classification report for test data
                    precision    recall  f1-score   support
...
          accuracy                           1.00      7875
         macro avg       1.00      1.00      1.00      7875
      weighted avg       1.00      1.00      1.00      7875
```

Figure 35. First decision tree model (overfitted)

```
Best parameters: {'max_depth': 9, 'min_samples_leaf': 0.05}
Best score: 0.9144119958304412
Classification report for training data
                    precision    recall  f1-score   support

                 C       1.00      1.00      1.00      1714
                C#       1.00      1.00      1.00      1695
               C++       1.00      1.00      1.00      1689
                Go       1.00      1.00      1.00      1691
           Haskell       0.94      0.66      0.78      1698
              Java       1.00      1.00      1.00      1648
        JavaScript       1.00      1.00      1.00      1687
            Kotlin       0.82      1.00      0.90      1693
               PHP       1.00      0.89      0.94      1688
            Prolog       1.00      0.00      0.00      1699
   Python Compilled       0.90      1.00      0.95      1712
Python Interpreted       0.54      1.00      0.70      1707
              Ruby       0.81      1.00      0.90      1642
              Rust       1.00      1.00      1.00      1662

          accuracy                           0.90     23625
         macro avg       0.93      0.90      0.87     23625
      weighted avg       0.93      0.90      0.87     23625

...
          accuracy                           0.90      7875
         macro avg       0.93      0.90      0.87      7875
      weighted avg       0.93      0.90      0.87      7875
```

Figure 36. Second decision tree model (tunned and no overfitted)

```
# Assuming you have already defined the variables:
Program = "QuickSort "
Size = 0.25
Cost = 1000
Package = 1.05
Time = 10
Memory = 2000

# Create a dictionary with the input data
input_data = {
    'Program': [replace_map['Program'][Program]],
    'Size': [Size],
    'Cost': [Cost],
    'Package': [Package],
    'Time': [Time],
    'Memory': [Memory]
}

# Create a DataFrame from the input data
input_df = pd.DataFrame(input_data)

# Extract the feature columns (x) from the input DataFrame
x = input_df[["Program", "Size", "Cost", "Package", "Time", "Memory"]]

# Make predictions for the input data
prediction = grid_search.predict(x)

# Retrieve the inferred value of "Language"
inferred_language = prediction[0]

print("Inferred Language:", inferred_language)

Inferred Language: Rust
```

Figure 37. Language prevision with decision tree

3.8 Multi-Criteria Optimization

	Cost	Package	Time	Temperature	Memory
Indifference	100,00	0,30	20,00	1,00	1000,00
Veto	500,00	0,70	80,00	5,00	10000,00
Weight	0,10	0,30	0,30	0,10	0,20
Lambda	0,80				
Languages	C				
	C#				
	C++				
	Go				
	Haskell				
	Java				
	JavaScript				
	Kotlin				
	PHP				
	Prolog				
	Python Compiled				
	Python Interpreted				
	Ruby				
	Rust				

Table 1. Parameters table

Credibility Index with lambda														
	C	C#	C++	Go	Haskell	Java	JavaScript	Kotlin	PHP	Prolog	Python Compiled	Python Interpreted	Ruby	Rust
C		1,00	1,00	1,00		1,00	1,00	1,00	1,00	1,00			1,00	1,00
C#						1,00			1,00					
C++	1,00	1,00		1,00		1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	
Go	1,00	1,00	1,00		1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
Haskell		1,00				1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	
Java								1,00	1,00	1,00				
JavaScript														
Kotlin									1,00					
PHP														
Prolog						1,00	1,00	1,00	1,00				1,00	
Python Compiled		1,00			1,00	1,00	1,00	1,00	1,00	1,00		1,00	1,00	
Python Interpreted						1,00	1,00	1,00	1,00	1,00			1,00	
Ruby						1,00		1,00	1,00	1,00				
Rust	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	

Table 2. Credibility index with lambda

4 Towards a Ranking of Sorting Algorithms and Programming Languages

4.1 Data Visualization - All Features

In accordance with section 3.1, we generated a scatter plot to analyze the language dispersion per programming language, considering various features including time, memory, temperature, package, core, GPU, and cost. Based on our analysis, the following observations can be made:

- The feature *package* takes into consideration the values from *core* and *GPU*, so we will disregard those features.
- In a general sense, Quick Sort emerges as the fastest sorting algorithm, exhibiting the lowest values for *package* and *core*. Additionally, it tends to be the most expensive algorithm across the majority of programming languages. The temperature of the cores during the execution of Quick Sort remains relatively consistent across different languages, averaging around 46ºC.
- Python demonstrates the lowest development cost among all the considered languages.

4.2 Powercap Influence For Each Sorting Algorithm

As seen in section 3.2, we will analyze the influence of power cap on each sorting algorithm. To conduct this analysis, we will fix the size value (i.e., the size of the array used in the algorithm) to 5000 and explore various values of the *PowerLimit* parameter. We will evaluate the impact in terms of execution time, power consumption (package) and memory consumption (memory). Due to the extensive range of programming languages considered in our study, we will focus on Python (both the compiled and interpreted versions) for this paper - including graphs for all the languages would not be practical for our study. Additionally, we will directly compare both versions of Python.

While examining the scatter plots, we can conclude that Quick Sort consistently performs as the fastest sorting algorithm across all the *PowerLimit* values in both versions of Python. In general, Bubble Sort is slower than Selection Sort, and consequently, slower than Quick Sort.

The time interval for all the compiled Python versions is [0, 70] ms, while the time interval for the interpreted version is [0, 3000] ms. Therefore, the compiled version of Python is faster than the interpreted version.

Now turning our attention to the energy consumed, we can see that Quick Sort is the most energy-efficient sorting algorithm, while Bubble Sort consumes more energy than Selection Sort.

The package interval for all the compiled Python versions is [0, 0.40] J, while the package interval for the interpreted version is [0, 30] J. Therefore, the compiled version of Python consumes less energy than the interpreted version.

Lastly, we will analyze the memory consumption of each sorting algorithm. In both versions, Quick Sort is the algorithm that consumes the most memory, while Bubble Sort consumes the least. However, it's important to note that they operate on different scales: the compiled version has a memory interval of [0, 7000] Kb, while the interpreted version has a memory interval of [13500, 14500] Kb. Therefore, the compiled version consumes less memory than the interpreted version.

4.3 Powercap Influence For Each Programming Language

As described in section , we will compare the influence of power cap on each programming language. Similarly to the previous subsection, we will fix certain values: we will only consider the best algorithm, Quick Sort (the fastest and most energy-efficient), and set the size value to 5000. Furthermore, we will analyze the impact on time, memory, and energy consumption.

After analyzing all three graphs, the following observations can be made:

- The lower the value of PowerLimit, the slower the language performs.

- In general, as the PowerLimit value increases, a language tends to perform faster and consume more energy.
- PHP is the slowest programming language (and the most energy-consuming), while C is the fastest programming language (and the least energy-consuming).
- Limiting the power to the algorithms does not affect the memory consumed by the language.
- JavaScript consumes the most memory, while C consumes the least.

4.4 Size Influence For Each Algorithm And For Each Language

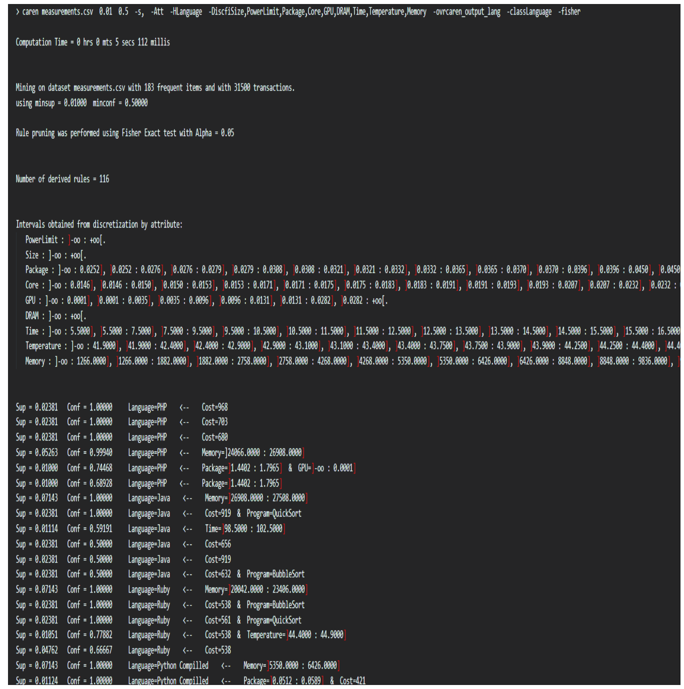
As seen in section 3.5, we will analyze the influence of size on each sorting algorithm and programming language. To do so, we will fix the PowerLimit value to 1000 and compare the impact on time, memory, package, and temperature.

After analyzing all the graphs, we can observe that as the size value increases, the sorting algorithms and programming languages tend to become slower, consume more memory, and consume more energy. However, the temperature of the cores shows only a minor increase and remains consistent across different programming languages.

4.5 Association Rules

To identify the association rules within our dataset, we used the program Caren [Paulo Azevedo 2017] in two scenarios: one with Language as the target feature and another with Program as the target feature. We also developed a script that consolidates all the association rules into a single text file, according with a certain confidence value (in this case, we will use conf = 1). After analyzing the results, we derived some interesting association rules, such as:

- If the cost of the algorithm is 919 and the program used is QuickSort, then the associated language is Java. This association has a support of 0.02381 and a confidence of 1.00000;
- If the package value falls within the range of 0.0279 to 0.0308, the time value falls within the range of 10.5000 to 11.5000, and the program used is QuickSort, then the associated language is Rust with a support of 0.01765 and a confidence of 1.00000;
- If the core value falls within the range of 0.0152 to 0.0169, the time value falls within the range of 10.5000 to 11.5000, and the language is Rust, then the associated program is QuickSort with a support of 0.01698 and a confidence of 1.00000



```
> caren measurements.csv  0.01  0.5  -s,  -Att  -HLanguage  -DiscfiSize,PowerLimit,Package,Core,GPU,DRAM,Time,Temperature,Memory  -ovrcaren_output_lang  -classLanguage  -fisher

Computation Time = 0 hrs 0 mts 5 secs 112 millis

Mining on dataset measurements.csv with 183 frequent items and with 31500 transactions.
using minsup = 0.01000  minconf = 0.50000

Rule pruning was performed using Fisher Exact test with Alpha = 0.05

Number of derived rules = 116

Intervals obtained from discretization by attribute:
  PowerLimit : ]-oo : +oo[.
  Size : ]-oo : +oo[.
  Package : ]-oo : 0.0252], ]0.0252 : 0.0276], ]0.0276 : 0.0279], ]0.0279 : 0.0308], ]0.0308 : 0.0321], ]0.0321 : 0.0332], ]0.0332 : 0.0365], ]0.0365 : 0.0370], ]0.0370 : 0.0396], ]0.0396 : 0.0450], ]0.0450
  Core : ]-oo : 0.0146], ]0.0146 : 0.0150], ]0.0150 : 0.0153], ]0.0153 : 0.0171], ]0.0171 : 0.0175], ]0.0175 : 0.0183], ]0.0183 : 0.0191], ]0.0191 : 0.0193], ]0.0193 : 0.0207], ]0.0207 : 0.0232], ]0.0232 :
  GPU : ]-oo : 0.0001], ]0.0001 : 0.0035], ]0.0035 : 0.0096], ]0.0096 : 0.0131], ]0.0131 : 0.0282], ]0.0282 : +oo[.
  DRAM : ]-oo : +oo[.
  Time : ]-oo : 5.5000], ]5.5000 : 7.5000], ]7.5000 : 9.5000], ]9.5000 : 10.5000], ]10.5000 : 11.5000], ]11.5000 : 12.5000], ]12.5000 : 13.5000], ]13.5000 : 14.5000], ]14.5000 : 15.5000], ]15.5000 : 16.5000
  Temperature : ]-oo : 41.9000], ]41.9000 : 42.4000], ]42.4000 : 42.9000], ]42.9000 : 43.1000], ]43.1000 : 43.4000], ]43.4000 : 43.7500], ]43.7500 : 43.9000], ]43.9000 : 44.2500], ]44.2500 : 44.4000], ]44.4
  Memory : ]-oo : 1266.0000], ]1266.0000 : 1882.0000], ]1882.0000 : 2758.0000], ]2758.0000 : 4268.0000], ]4268.0000 : 5350.0000], ]5350.0000 : 6426.0000], ]6426.0000 : 8848.0000], ]8848.0000 : 9836.0000], ]

Sup = 0.02381   Conf = 1.00000    Language=PHP   <--   Cost=968
Sup = 0.02381   Conf = 1.00000    Language=PHP   <--   Cost=703
Sup = 0.02381   Conf = 1.00000    Language=PHP   <--   Cost=680
Sup = 0.05263   Conf = 0.99940    Language=PHP   <--   Memory=]24066.0000 : 26908.0000]
Sup = 0.01000   Conf = 0.74468    Language=PHP   <--   Package=]1.4402 : 1.7965]  &  GPU=]-oo : 0.0001]
Sup = 0.01000   Conf = 0.68928    Language=PHP   <--   Package=]1.4402 : 1.7965]
Sup = 0.07143   Conf = 1.00000    Language=Java   <--   Memory=]26908.0000 : 27508.0000]
Sup = 0.02381   Conf = 1.00000    Language=Java   <--   Cost=919  &  Program=QuickSort
Sup = 0.01114   Conf = 0.59191    Language=Java   <--   Time=]98.5000 : 102.5000]
Sup = 0.02381   Conf = 0.50000    Language=Java   <--   Cost=656
Sup = 0.02381   Conf = 0.50000    Language=Java   <--   Cost=919
Sup = 0.02381   Conf = 0.50000    Language=Java   <--   Cost=632  &  Program=BubbleSort
Sup = 0.07143   Conf = 1.00000    Language=Ruby   <--   Memory=]20042.0000 : 23406.0000]
Sup = 0.02381   Conf = 1.00000    Language=Ruby   <--   Cost=538  &  Program=BubbleSort
Sup = 0.02381   Conf = 1.00000    Language=Ruby   <--   Cost=561  &  Program=QuickSort
Sup = 0.01051   Conf = 0.77882    Language=Ruby   <--   Cost=538  &  Temperature=]44.4000 : 44.9000]
Sup = 0.04762   Conf = 0.66667    Language=Ruby   <--   Cost=538
Sup = 0.07143   Conf = 1.00000    Language=Python Compilled   <--   Memory=]5350.0000 : 6426.0000]
Sup = 0.01124   Conf = 1.00000    Language=Python Compilled   <--   Package=]0.0512 : 0.0589]  &  Cost=421
```

Figure 38. Partial output got from CAREN

4.6 Clustering

After completing the statistical analysis, we proceeded to employ various clustering techniques, specifically the K-means algorithm and hierarchical clustering (refer to section 3.6). The dataset utilized in this analysis consisted of results obtained from Python Interpreted, with Size = 5000 and PowerLimit = 1000. Our objective was to assess whether the clustering algorithm could identify any discernible patterns within the dataset, with a particular focus on the sorting algorithms.

To determine the optimal number of clusters, we employed the elbow method. This technique involves evaluating the within-cluster sum of squares for different numbers of clusters and selecting the number of clusters where the improvement in clustering quality diminishes significantly. In our case, the analysis indicated that the optimal number of clusters is 3, which aligns with the number of sorting algorithms under consideration.

Upon applying the K-means algorithm, we observed that the data points formed three distinct groups. These groups potentially corresponded to the three sorting algorithms that were included in our analysis. This finding suggests that the clustering algorithm was able to successfully segregate the data points based on their similarities or dissimilarities, highlighting potential patterns or differences among the sorting algorithms.

In addition, we conducted hierarchical clustering on the entire dataset; however, we did not observe any significant or meaningful results.

4.7 Decision Tree - Machine Learning Model

The next method we will utilize in our study is a machine learning model called a Decision Tree. The first step we took is to identify the relevant features for the model, which are Memory, Core, Package, Size, and Time. Additionally, we have decided to include the Program feature due to its relevance in the language prediction script we have developed.

Next, we created the decision tree model; however, our initial version was overfitting the data. To resolve this problem, we performed model tuning and found that the optimal values for max_depth and min_samples_leaf are 9 and 0.05, respectively. With these adjustments, our model achieved an accuracy of 90

As mentioned earlier, we will use our tuned model to predict the language based on values provided for other features. In figure 37, we can observe an example where Rust is the predicted language.

4.8 Multi-Criteria Optimization

In this analysis, it is crucial to consider the parameters we have defined. Our study primarily focused on time and performance, making these components the most valuable (30%). As our emphasis was on performance, we assigned greater importance to Memory over Cost and Temperature, which were deemed less significant by the group. Normally, this analysis is conducted using a fork graph. However, due to the substantial dataset, we found it more comprehensible to utilize a directions table.

The values used in the analysis were filtered from QuickSort (the fastest algorithm) with a Size of 5000, representing the heaviest workload where significant changes could be observed. Furthermore, no power limit was imposed on any of the values.

Based on this analysis, the group concluded that while C was one of the best programming languages, C++, Go, Haskell, Python (Compiled), and Rust were also excellent options with a considerable number of outNodes. On the other hand, Java, JavaScript, Kotlin, PHP, and Prolog received less favorable ratings, showing numerous inNodes and being perceived as less effective.

In our further analysis, we could have also included comparisons of the best algorithms under different power limitations. However, we decided to focus on other aspects of this paper and leave this aspect for future research.

4.9 Sorting Algorithms Rankings

4.9.1 By Performance

Ranking	Sorting Algorithm	Average Execution Time (in ms)
1	QuickSort	57.306190
2	BubbleSort	230.314762
3	SelectionSort	245.069048

Table 3. Sorting Algorithms Ranking by Execution Time (PowerLimit = 5)

Ranking	Sorting Algorithm	Average Execution Time (in ms)
1	QuickSort	44.736667
2	BubbleSort	138.713810
3	SelectionSort	149.520000

Table 4. Sorting Algorithms Ranking by Execution Time (PowerLimit = 1000)

4.9.2 By Energy Consumption

Ranking	Sorting Algorithm	Average Energy Consumption (in J)
1	QuickSort	0.229320
2	BubbleSort	1.080240
3	SelectionSort	1.159138

Table 5. Sorting Algorithms Ranking by Energy Consumption (PowerLimit = 5)

Ranking	Sorting Algorithm	Average Energy Consumption (in J)
1	QuickSort	0.340966
2	BubbleSort	1.921211
3	SelectionSort	2.058871

Table 6. Sorting Algorithms Ranking by Energy Consumption (PowerLimit = 1000)

4.10 Programming Languages Rankings

4.10.1 By Performance

Ranking	Programming Language	Average Execution Time (in ms)
1	C	8.786667
2	Rust	10.966667
3	C++	13.086667
4	Go	13.366667
5	Haskell	19.453333
6	Python Compilled	20.073333
7	C#	45.020000
8	Python Interpreted	61.146667
9	JavaScript	80.633333
10	Prolog	82.326667
11	Java	105.546667
12	PHP	108.573333
13	Ruby	108.653333
14	Kotlin	124.653333

Table 7. Programming Languages Ranking by Execution Time (Algorithm = QuickSort and PowerLimit = 5)

Ranking	Programming Language	Average Execution Time (in ms)
1	C	8.506667
2	Rust	10.673333
3	C++	12.746667
4	Go	13.020000
5	Python Compilled	18.460000
6	Haskell	19.060000
7	C#	40.753333
8	Python Interpreted	48.960000
9	JavaScript	54.220000
10	Prolog	61.606667
11	Ruby	76.740000
12	PHP	78.366667
13	Java	83.840000
14	Kotlin	99.360000

Table 8. Programming Languages Ranking by Execution Time (Algorithm = QuickSort and PowerLimit = 1000)

4.10.2 By Energy Consumption

Ranking	Programming Language	Average Energy Consumption (in J)
1	C	0.025449
2	Rust	0.029564
3	Go	0.036158
4	C++	0.036818
5	Haskell	0.056860
6	Python Compilled	0.064108
7	C#	0.114102
8	Python Interpreted	0.249034
9	JavaScript	0.332982
10	Prolog	0.360690
11	Java	0.452335
12	PHP	0.477769
13	Ruby	0.483041
14	Kotlin	0.491566

Table 9. Programming Languages Ranking by Energy Consumption (Algorithm = QuickSort and PowerLimit = 5)

Ranking	Programming Language	Average Energy Consumption (in J)
1	C	0.024377
2	Rust	0.028760
3	C++	0.035053
4	Go	0.036952
5	Haskell	0.056654
6	Python Compilled	0.067841
7	C#	0.131434
8	Python Interpreted	0.324925
9	Prolog	0.529621
10	JavaScript	0.541427
11	Java	0.696985
12	PHP	0.732024
13	Ruby	0.774072
14	Kotlin	0.793397

Table 10. Programming Languages Ranking by Energy Consumption (Algorithm = QuickSort and PowerLimit = 1000)

5 Conclusions

In this scientific paper, we conducted a study on several metrics for benchmarking three sorting algorithms: Bubble Sort, Selection Sort, and Quick Sort. Each algorithm was implemented in thirteen programming languages, including C, C++, C#, Rust, Go, Python, Java, Kotlin, Haskell, Ruby, JavaScript, PHP, and Prolog. The metrics taken in consideration in this study are execution time, memory consumption, energy consumption, development cost, core temperature, and power capping.

As demonstrated in numerous preceding figures, C emerged as the best programming language across multiple factors, even when altering the size or power cap values. Similarly, Quick Sort was identified as the optimal sorting algorithm for the same reasons. On the other hand, Kotlin was found to be one of the worst-performing programming languages in terms of both performance and energy consumption.

References

Pereira, R. *et al.* (2020). "Ranking Programming Languages by Energy Efficiency" *in* Science of Computer Programming, volume 205. Elsevier, 2021.

Marcus Hahnel, Bjorn Dobel, Marcus Volp, and Hermann Hartig. "Measuring energy consumption for short code paths using RAPL". SIGMETRICS Perform. Eval. Rev., 40(3):13–17, jan 2012.

Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. "Red Alert for Power Leakage: Exploiting Intel RAPL-Induced Side Channels "in ASIA CCS '21: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, May 2021

Ayse Bener *et al.* (2014). in "Green Software," p. 37 - 38

Paquete, L. (2023). Multi-criteria optimization workshop slides. Available at: https://eden.dei.uc.pt/~paquete/sustrainable/slides.pdf (Accessed: June 2023).

Python compiller (Codon). Official repository of Codon. Available at: https://github.com/exaloop/codon (Accessed: June 2023).

Complexity of sorting algorithms. Available at: https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/ (Accessed: June 2023).

Vince Weaver: RAPL interface (in C). Available at: https://web.eece.maine.edu/~vweaver/projects/rapl/rapl-read.c (Accessed: June 2023).

GeeksForGeeks website. Available at: https://www.geeksforgeeks.org (Accessed: June 2023).

GitHub repository of lm-sensors (2021). Available at: https://github.com/lm-sensors/lm-sensors (Accessed: June 2023).

GitHub repository of Sloc Cloc and code (2023). Available at: https://github.com/boyter/scc (Accessed: June 2023).

GitHub repository of PowerCap (2023). Available at: https://github.com/powercap/powercap (Accessed: June 2023).

Paulo Azevedo (2017): CAREN - Class project Association Rule ENgine. Available at: https://www.di.uminho.pt/~pja/class/caren.html (Accessed: June 2023).