On the Energy Efficiency of Sorting Algorithms

Catarina Gonçalves
Universidade do Minho
Portugal
pg50180@alunos.uminho.pt

Francisco Toldy
Universidade do Minho
Portugal
pg50379@alunos.uminho.pt

Joana Alves
Universidade do Minho
Portugal
pg50457@alunos.uminho.pt

Abstract

This document details the work done throughout the semester, in the context of the SDVM (Software Development Validation and Maintenance) profile of the Software Engineering Masters Degree. The proposed problem was to evaluate the different performance of languages when implementing several sorting algorithms, with a focus on energy efficiency. Therefore, we monitored and measured three implementations of sorting algorithms programmed using two programming languages. After doing so, we gathered several metrics to easily perform the statistical analysis of the performance of each case, and then follow that analysis with a multicriteria optimization analysis to evaluate which where the preferable algorithms and languages. This allowed us to compare each case and conclude that **C** is the language with overall best performance and **Quick Sort** is the best performing algorithm in the languages that were used.

Keywords Green Software, Programming Languages, Sorting Algorithms, RAPL, PowerCap

1 Introduction

As environmental sustainability becomes increasingly important, the concept of "green software" has emerged as a crucial area of research. Green software focuses on developing energy-efficient software solutions that minimize power consumption and reduce the carbon footprint of computing systems. Sorting algorithms play a fundamental role in software applications, and their performance directly impacts energy consumption and system efficiency. This paper aims to compare the performance of different sorting algorithms from a green software perspective, analyzing their energy consumption and execution time.

2 Methodology

In this section we will present the methodology followed by the group in order to achieve the most correct results possible, including every measured metric and its description and objective.

2.1 Implementations of Sorting Algorithms

We started by choosing the algorithms and then the languages we wanted to implement them in. Thus, we searched for sorting algorithms and ended up choosing the ones we thought had similar time complexity in their three possible run time scenarios: best, average and worst. Consequently, after analyzing this study, we chose the **merge sort**, **heap sort** and **quick sort** algorithms.

After that, we needed to choose the programming languages to implement this algorithms in, so we started searching in this particular GitHub repository ([diptangsu 2021]) which contained almost every sorting algorithm implemented in various languages. These implementations were all almost identical to each other in order to achieve maximum similarity in executed instructions in every language. After this research, we ended up choosing the languages **C** and **Python** to compare how fast or slow one is relatively to the other.

After that, for every algorithm, we created **three versions** according to an input size setting: small (1000 elements), medium (5000 elements) and large (10000 elements). We implemented them in both languages, which totals 9 programs in each language and **18 programs** in total. It's important to note that the content of the input is equal in both languages, no input has repeated elements and all arrays have N different positive numbers, with N being the array size.

2.2 Energy Consumption Monitoring

To gather metrics regarding Energy Consumption, we were faced with 2 alternatives, an external component to measure the energy consumed or a framework provided by chip manufacturers that can measure or estimate the consumption of their chips. Intel has, for some time, provided their own easy to use software framework by the name of **RAPL** (Running Average Power Limiter) to estimate the **energy consumption** of their CPUs.

However, it needs to be updated to the newest Intel architectures and being implemented as a single main file, it has no support for modular instrumentation of other programs. Due to this, our teachers at Universidade do Minho updated this interface to support all architectures and transformed its code in order to facilitate the manual instrumentation of any C program with energy measurements instructions. With this, we were capable of executing any program and receive its energy measurements as output.

Beyond this, by using the **sensors** library, we extended the functionality of the program to not only get the energy consumption estimation, but also the **CPU temperature** before and after every execution. With this, we defined a temperature **threshold** in which the program would not run if the CPU temperature was not below it. This threshold was

Perfil SDVM, 2023
2023.

determined by analysing the CPU temperature while it was in rest and adding 10 degrees (ºC) to it.

2.3 Limiting the Energy Consumption with PowerCap

Besides monitoring the energy consumption of each program, we also wanted to limit it by applying a mechanism to guarantee that the consumption is below a certain power cap. For this, we used **RAPL CAP** - C interface for getting/setting power caps with RAPL. Therefore, our program was extended once again to allow the setting of the upper bounds of energy consumption for Intel CPUs.

We defined these values taking into consideration the used CPU's (Intel Core i7-8550U) minimum and maximum **TDP** (Thermal Design Power) values: **10W** and **25W**, respectively.

2.4 Memory Use Monitoring

For the purpose of monitoring the memory usage, we had to search for a solution that worked similarly in C and Python to avoid skewing the conclusions. After some research, we arrived at the **Resource Usage GNU C** library and **Resource** for Python. These solutions allowed to measure the maximum resident set size (the largest amount of physical memory the process has ever been using at any one instant) for each sorting algorithm in both C and Python. This metric was considered to be useful as it provides insight into the peak memory usage for each algorithm. To allow for the inclusion of this metric in the statistical analysis, we had to then write the results in the corresponding .csv files.

2.5 Software Cost Estimation

In order to compare both languages and algorithms in terms of developing costs, we used **SLOCCount** - a set of tools for counting physical Source Lines of Code (SLOC) in a large number of languages.

This tool offers many cost metrics, including total physical source lines of code, schedule estimate (in years), estimated average number of developers, amongst others. However, for this paper in particular, we're interested in the **Total Estimated Cost to Develop** metric, which gives us an estimate in dollars for the cost of developing this project.

3 Testing Environment

In this section we will present the specifications of the laptop used for all the tests, specifically its CPU information. It's important to note that this processor is designed for efficient performance in thin and light laptops, providing a balance between power consumption and processing capabilities. Because of this, it's suitable for tasks such as office productivity, web browsing, multimedia consumption, and light to moderate content creation. That being said, these are it's specs:

Model Name	Intel(R) Core(TM) i7-8550U
Base Frequency	1.80GHz
Max Turbo Frequency	4.00GHz
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	39 bits physical 48 bits virtual
Byte Order	Little Endian
CPU(s)	8
CPU Family	6
Model	142
Thread(s) per core	2
TDP	15W
Configurable TDP-up	25W
Configurable TDP-down	10W

4 Benchmarking the Performance of the Implementation of Sorting Algorithms

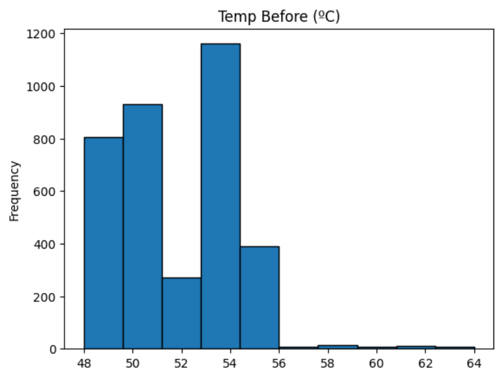
In this section, we'll present various tables and graphics in order to show the range of values, and its frequencies, for all the attributes present in the output (csv) file. Some attributes like 'Language', 'PowerCap' and 'ArraySize' won't be displayed in this section due to its static values, that is, each one of this attributes has a set of predefined values, which are:

- **Language:** c , python
- **PowerCap:** 10 , 25
- **ArraySize:** 0 , 1 , 2

That being said, we'll start by presenting the attributes in the order they appear in the header of the output file.

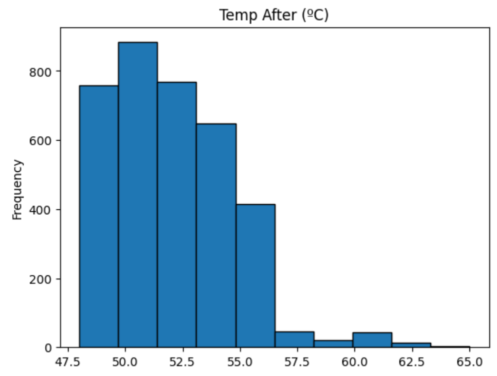
4.1 Temp Before (ºC)

This attribute represents the measured CPU temperature right before the execution of the program, in which we obtained the following distribution:



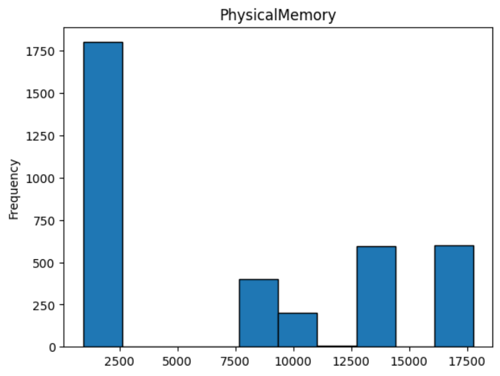
4.2 Temp After (°C)

This attribute represents the measured CPU temperature right after the execution of the program and it has the following distribution:



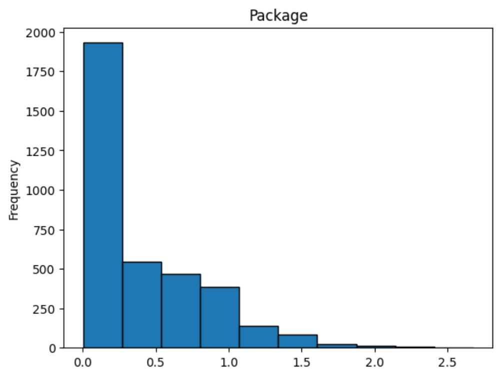
4.3 Physical Memory

This attribute measures the maximum resident set size (the largest amount of physical memory the process has ever been using at any one instant) and it has the following distribution:



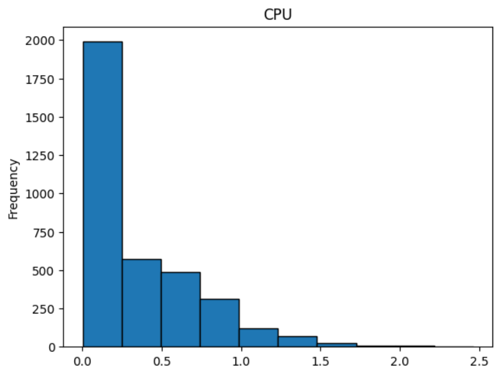
4.4 Package

This attribute 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) and it has the following distribution:



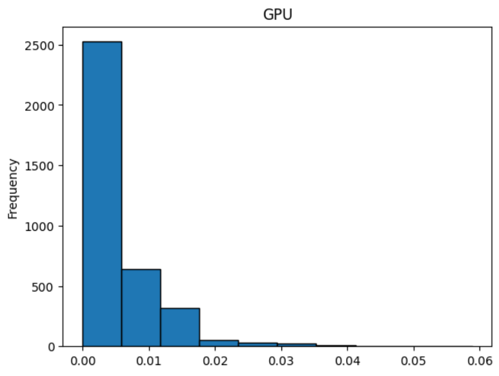
4.5 Core(s)

This attribute represents the energy consumed by all cores and caches and it has the following distribution:



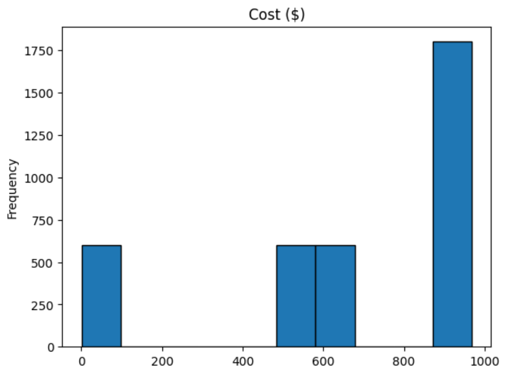
4.6 GPU

This attribute represents the energy consumed by the GPU and it has the following distribution:



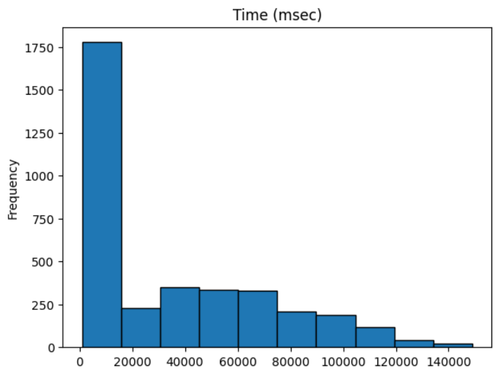
4.7 Cost ($)

This attribute represents the total estimated cost (in dollars) to develop the program and it has the following distribution:



4.8 Time

This attribute represents the execution time in milliseconds and it has the following distribution:



5 Towards a Ranking of Sorting Algorithms

In order to better visualize and understand all the data obtained in the previous section and how some attributes are related to others, we started by obtaining some basic statistics from the data and testing its distribution, implemented clustering algorithms and, finally, applied multi-criteria optimization. All of these steps will be explained in more detail in the following sections.

5.1 Basic Statistics

In this section we will present the first step we took in understanding our data, with the main objective of visualizing and determining data distribution that deserve some attention.

As it was expected, the rising of the package energy consumption implied a rising of the time it took the algorithm to finish, as it can be seen in the following scatterplot:

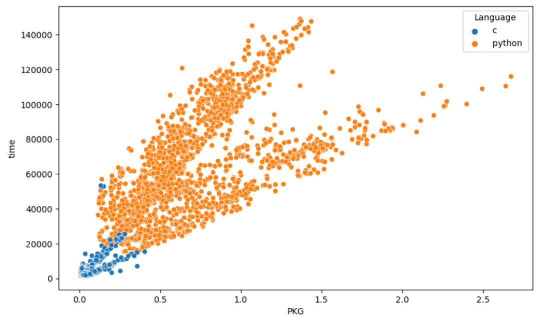


Figure 1. Scatterplot PKG by Time (hue=Language)

It can also be seen a great disparity between the values of time and Package consumption in the C language and Python, being C's values much lower. It can also be seen the clear difference between the measurements with the powercap limit at 25 and 10.

The scatterplot between CPU and time is very similiar to the previous one, since the Package consumption includes the CPU consumption. This means that there must be a clear linear relation between the CPU and PKG, which is confirmed:

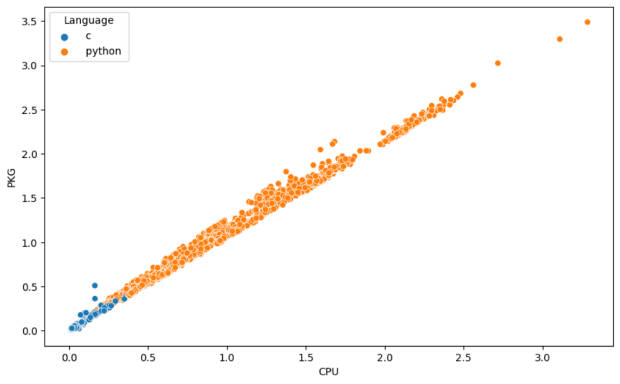


Figure 2. Scatterplot CPU by PKG (hue=Language)

However, the same does not happen when we focus on the GPU parameter, showing no apparent relation with time and neither of other the attributes.

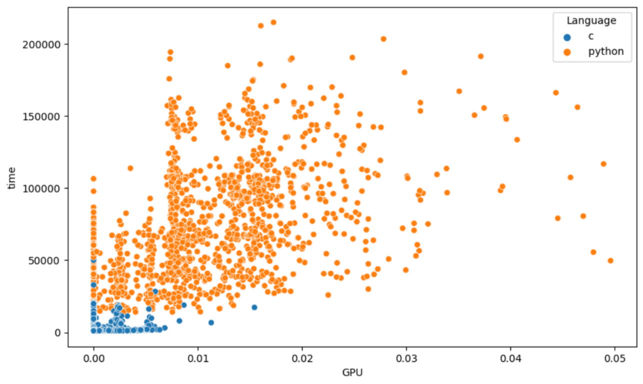


Figure 3. Scatterplot GPU by Time (hue=Language)

After analysing the behaviour between pairs of attributes, the next step was to observe and analyse the statistics (mostly the average) of the various parameters depending on the context each group of data was run on.

	Language	Temperature Increase
Medium Array Size + PowerCap=25	C	0.22
	Python	0.74
Big Array Size + PowerCap=25	C	0,33
	Python	0,69
Medium Array Size + PowerCap=10	C	0
	Python	0,077

We can confirm, that based on this results, Python executions showed an increase of temperature by more than double of the same parameter in a c execution. Therefore, C has better statistics based on rising of temperature.

Given this conclusion, let's proceed to the analysis of CPU and PKG energy consumption:

	Language	PKG	CPU
Medium Array Size + PowerCap=25	C	0,041	0.0358
	Python	0.724	0.6412
Big Array Size + PowerCap=25	C	0,0549	0,0464
	Python	1.2121	1.0951
Medium Array Size + PowerCap=10	C	0,041	0,0334
	Python	0,661	0,5411

In figure 1 and 2 we could already have taken some conclusions about the the average of CPU and PKG consumption between C and python, since the C values (in blue) were mostly at the bottom of the time axis. However, with this statistic we can evaluate more precisely the difference between them. We can then see that there is a great disparity between the values of C and Python, more precisely: in the first row the python is approximately 18 times worse than C in PKG and CPU consumption. In bigger arrays, is approximately 22 time worse, and with a power limit python is 16 times worse. Therefore, we can infer that C is also better at keeping the energy consumption low compared to Python.

Finally, let's analyse the memory and time results:

	Language	Memory	Time
Medium Array Size + PowerCap=25	C	961	2662
	Python	13029	50691
Big Array Size + PowerCap=25	C	974	3708
	Python	17482	74649
Medium Array Size + PowerCap=10	C	961	4436
	Python	13021	78912

As it is represented in this table, we can conclude that, even though the arrays used in each language were exactly the same, the memory usage doesn't seem to be the same, meaning python, with the same data, will use 15 times more memory in comparison with the C algorithm.

5.2 Density Estimation and Histograms

As a way to continue the work of studying the data distribution and it's visualization, the group considered it would be useful to study several Histograms and respective Density Estimation.

The group decided it would be useful to visualize the distribution of data according to 2 parameters:

- PKG
- Time

And use those parameters with all the algorithms, separated by input size (also known as the 'ArraySize' parameter) and then also make an overall comparison of each language with power limitations switched on and off.

5.2.1 PKG

For the PKG parameters, we first created a Histogram that shows how the PKG values are distributed when the input size is Small.

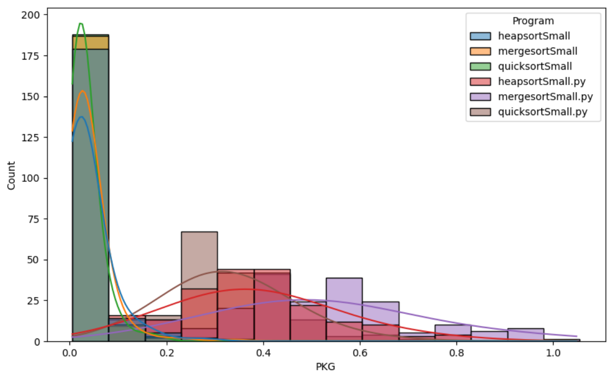


Figure 4. Small Input test data histogram

Then, we created a Histogram for when the input size is Medium.

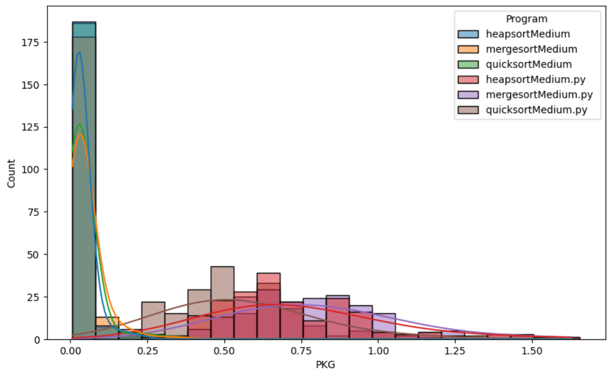


Figure 5. Medium Input test data histogram

Lastly, we created a Histogram for when the Input size is Large.

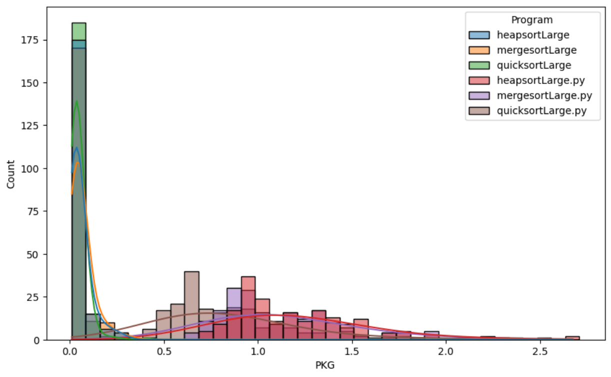


Figure 6. Large Input test data histogram according to Package value

We considered it would be useful to show how the PKG values are distributed in relation to the input size, to see if there are any major shifts in value distribution for each algorithm as the input size changes. As we can see, we can conclude that, as the input size gets progressively bigger, some changes start to become noticeable, for example how the QuickSort's C implementation has bigger count difference in the first bin, compared to the other input size's and other algorithms. We can also visualize that, with an increase of input size, the PKG values range for the Python algorithms also increases, a phenomena that is not as prevalent in the C implementations.

After the analysis of the way the input size affects the data distribution, we decided to see how the PKG value is distributed across each algorithm for each language, starting with C.

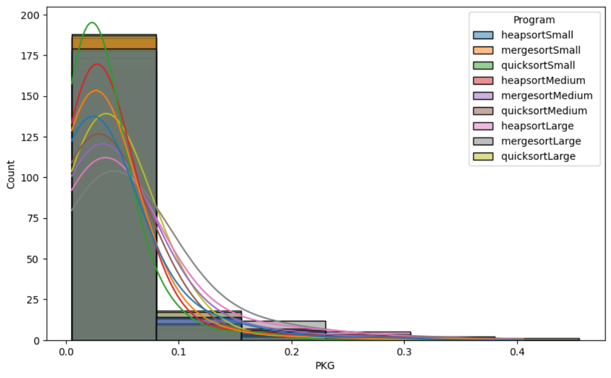


Figure 7. Test data histogram for the C algorithms according to Package values

And then we did the same for Python.

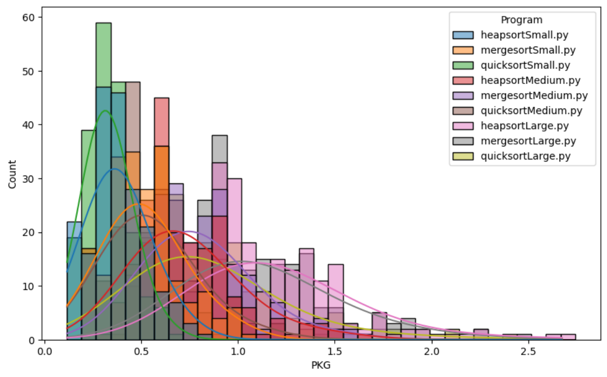


Figure 8. Test data histogram for the Python algorithms according to Package values

The rationale with these 2 graphics was to understand how the PKG values are distributed according to the Language used. It is also worth noting that both graphs have the same binwidth. The two graphics allowed us to better see what was already starting to become noticeable in the first set of histograms, which is that the C algorithms present a much smaller range of PKG values and distribution than the Python algorithms. We can also visualize in the 2nd graph that there is a much bigger variation among the algorithm tests values than there is in the 1st one.

Finally, we decided to observe how the PKG values distribution changes for each Language with the PowerCap values, as to observe if there are major shifts when we employ a lower limit.

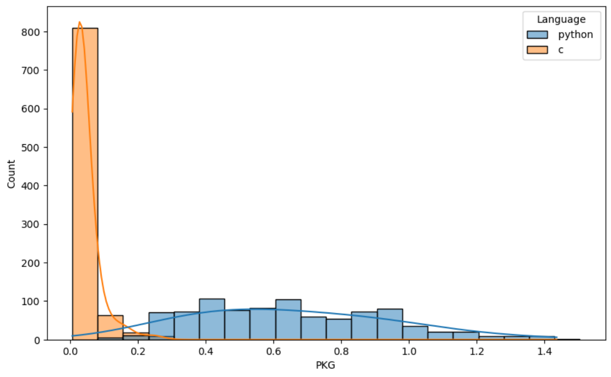


Figure 9. Test data histogram for the each language with a power limiter

And then without the limiter:

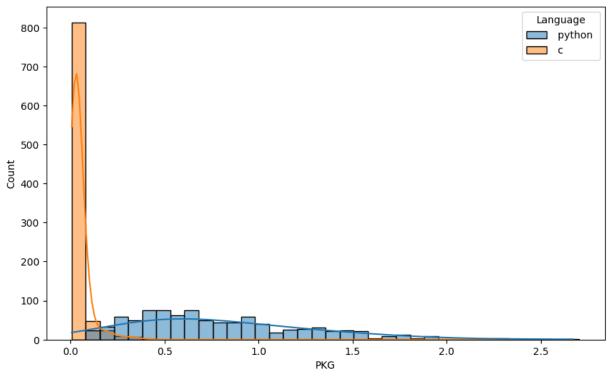


Figure 10. Test data histogram for the each language without a power limiter

From these two graphs we can conclude that the C algorithms are largely unaffected by the lower limit, as the distribution of the values doesn't change noticeably between the histograms. However, we cannot infer the same conclusions for the Python algorithms, as a large relative increase in range of PKG values is visible.

5.2.2 Time

Following the analysis of the data distribution according to the Package values, the group shifted it's focus to the Time parameter. As such, the same reasoning was used, starting with a breakdown of the data distribution for each input size.

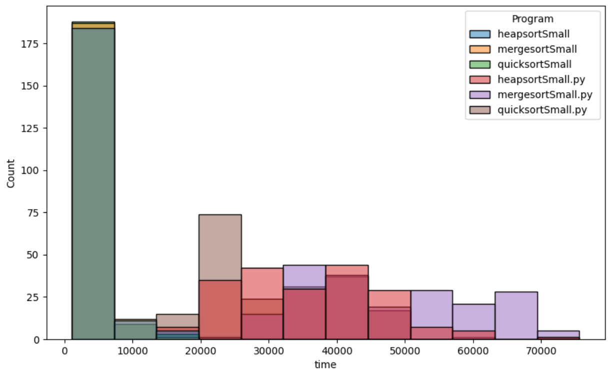


Figure 11. Small Input test data histogram according to 'Time' value

And then for the medium input size:

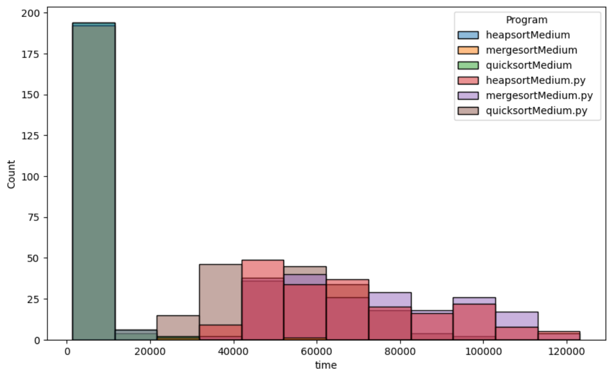


Figure 12. Medium Input test data histogram according to 'Time' value

And, finally, the large input size:

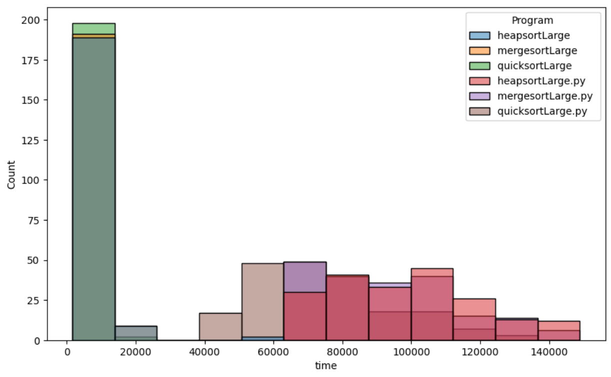


Figure 13. Large Input test data histogram according to 'Time' value

There are some conclusions we can take from observing these three histograms. The first is that, similarly to the Package histograms, there is a clear difference in distribution between the C algorithms and the Python algorithms. The second conclusion we can take is that, as the input size increases, the gap between the times of the C algorithms and the Python algorithms becomes increasingly bigger, indicating that the Python algorithms might not scale as well as the C algorithms. The third conclusion is that, as the input size changes, the distribution of each C algorithm also changes, with , for example, the Quick Sort seemingly bigger count in the first bin, compared to the rest.

The group then proceeded, once again, to further investigate how the data was distributed for each algorithm and each language,first for the C algorithm instances:

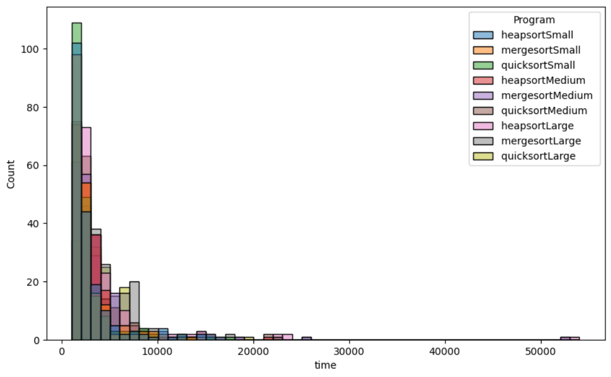


Figure 14. Test data histogram for the C algorithms according to Time values

And then for the Python:

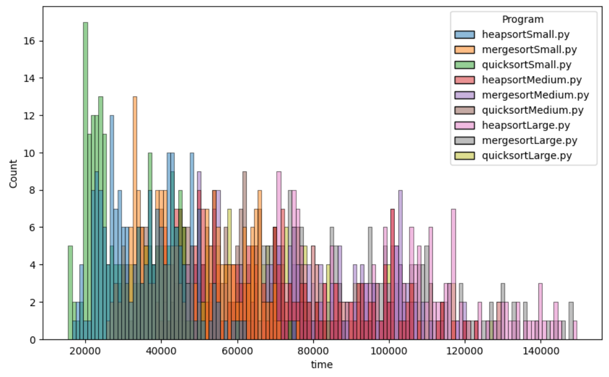


Figure 15. Test data histogram for the Python algorithms according to Time values

Once again, the same binwidth was used (1000) for both histograms . This allowed us to visualize how, at the same scale, the C algorithms, once again, have a much smaller range of time values. However, we can also detect some apparent outliers in the C histogram. For the python, we once again can conclude that the range of values is much bigger than for the C algorithms.

The last step was to analyse how the Time values distribution changes for each Language with the PowerCap values,as to observe if there are major shifts when we employ a lower limit, first with the limiter on:

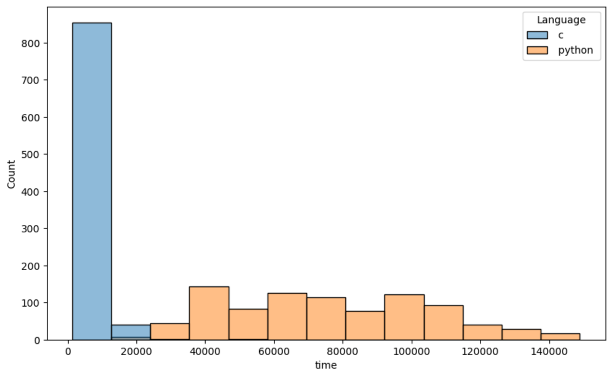


Figure 16. Test data histogram for the each language with a power limiter

And then without the limiter:

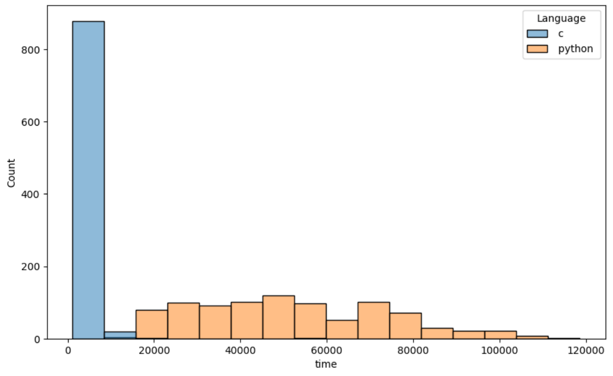


Figure 17. Test data histogram for the each language without a power limiter

As we can see, without the power limiter, the C algorithms perform relatively better than before, as expected, as the count on the first bin increases. However, even though the overall range decreases, the Python algorithms maintain a much more spread out distribution than the C algorithms, similarly to what occured in the Package Histograms.

5.3 Correlation Coefficients

After the density estimation analyzed, we then proceeded to the computation of the 3 most known correlation coefficients: Pearson, Spearman and Kendall.

5.3.1 Pearson Correlation

The Pearson Correlation Coefficient translates in the existence of a perfect linear relation between 2 attributes. In this section there was defined as the maximum of the pvalue, the default 0.05.

In the 'Basic Statistics' section of this document, there was some mentions of linear relations between CPU and PKG. Given this hint, with the Pearson Coefficient we can prove if the correlation is significant.

	Value	pValue
Pearson coefficient	0.997	0.0

Given these values, we can now confirm, that the correlation is indeed significant (pvalue < 0.05), which also confirms that it is a positive linear relation.

5.3.2 Spearman Correlation

The Spearman Correlation Coefficient translates to the existence of a perfect monotonic relation between 2 attributes, this meaning, that the two attributes increase/decrease in the same direction but not necessarily at the same rate.

	Value	pValue
Spearman coefficient	0.99	0.0

Given these values, we can now confirm, that the correlation is indeed significant (pvalue < 0.05) and it also confirms that there is a positive monotonic relation.

5.3.3 Kendall Correlation

The Kendall Correlation Coefficient translates in the existence of a perfect rank relation between 2 attributes.

	Value	pValue
Kendall coefficient	0.93	0.0

Given these values, we can now confirm, that the correlation is indeed significant (pvalue < 0.05) , which also confirms that there is a positive rank relation.

5.4 Outliers

Even though it is important to make use of all of the records obtained, some of them may be the result of a bad measurement or a software error. Therefore , it would be wise to take some time to analyze the outliers for consequent removal. This removal, can improve the results of the clustering algorithms without the outliers disturbing the results.

After the visualization of the box plots, the following conclusions were taken:

- **Cost:** doesn't have outliers
- **Memory:** outliers could be removed since they are only 0 to 15 among 200
- **Remaining:** the number of outliers in the remaining attributes are much larger and therefore the removal would possibly bias the results.

5.5 Anderson-Darling Test

In short, the Anderson-Darling test is a statistical test used to determine if a data set follows a specific distribution, typically the normal distribution. It measures the discrepancy between the observed data and the expected values from the assumed distribution. Thus, we implemented this test using the package 'anderson' from the library 'scipy.stats', in order to check if the data follows a **Gaussion** distribution.

With all tests done, we concluded that no program in any attribute followed a Gaussion distribution.

5.6 Wilcoxon Test

The Wilcoxon test, also known as the Wilcoxon signed-rank test, is a non-parametric statistical test used for comparing paired samples or repeated measures. It is an alternative to the paired t-test when the assumptions of normality or equal variances are violated or when dealing with ordinal or non-normally distributed data.

With this in mind, we started our **first set** of tests by applying this test to every pair of programs with the same input size and in the same programming language, noting that the order of the tests mattered. After this, we implemented a **second set** of tests in every pair with programs of both languages, for example: heap sort (Python) and merge sort (C) and vice-versa, both with the same input size. We concluded with a **third set** of tests that compared the same algorithm (in each language) with its three implementations, that is small, medium and large, to see if there was a significant difference between them.

Note that in all test cases we tested the pairs with the attribute **CPU** with a significance level of **0.05** and with the attribute **less** which means the algorithm will perform a one-sided test with the alternative hypothesis that the median difference is less than zero, so it tests if the first program in the pair has a smaller median than the second one.

5.6.1 First Set Results

The first set of tests aimed to verify within the same language and input size, which algorithm had the lower median. After obtaining the results, we concluded that for language C and input size small, there was no significant difference within the algorithms. We also noted that for language C and input size medium and large, the results were equal. Finally, the Python language had the same results in every test. That being said, we present the "best" algorithm(s), that is, the algorithm(s) with the lower median in each language:

- **C:** Heap Sort and Quick Sort
- **Python:** Quick Sort

By analyzing the results, we can conclude that the **Quick Sort** algorithm has a lower median in both languages, in every input size, leaving a hint that it might be the best choice of algorithm.

5.6.2 Second Set Results

The second set of tests aimed to verify the best performing language by comparing the algorithms in both languages in each input size. After obtaining the results, we concluded that the **C** language has an overall lower median in all versions of the algorithms, comparing to Python.

5.6.3 Third Set Results

Finally, in the third set of tests, we aimed to verify for each algorithm, in each language, if their medians in each input size (small, medium and large) had significant differences. With this, we obtained the following results:

- **C:**
  - **Quick/Heap Sort:** no significant differences in small and medium input sizes
  - **Merge Sort:** significant differences between all input sizes
- **Python:** all algorithms in all input sizes had significant differences

After analyzing the results, we concluded that the algorithms **Heap** and **Quick Sort** (in **C** language) have no significant differences between their small and medium versions, which means that only with larger inputs we start to notice an increase in CPU's energy consumption. This means that these two algorithms, implemented in C, are probably the best choice in smaller to medium inputs (<5000 elements).

5.6.4 Aggregated Results

After the three sets of tests concluded, we analysed the conclusions drawn in each of them. In this way, we observed that the best combination of algorithm and programming language is: **Quick Sort implemented in C**.

5.7 Clustering Algorithms

With the previous steps done, we started implementing clustering algorithms in order to visualize and verify if the data followed some form of "natural grouping". To achieve this, we used various algorithms, starting with K-Means.

5.7.1 K-Means

K-Means clustering algorithm partitions the data into a predefined number of clusters (specified beforehand), where each data point belongs to the cluster with the nearest mean (centroid).

Consequently, we needed to obtain the reasonable number of clusters to look for, so, as we had 18 different types of programs, we implemented the algorithm setting the number of clusters 18 in order to try to verify a possible correct allocation of all programs and their due points. However, the results were inconclusive and confusing, due to the "layer" clustering the algorithm was performing:

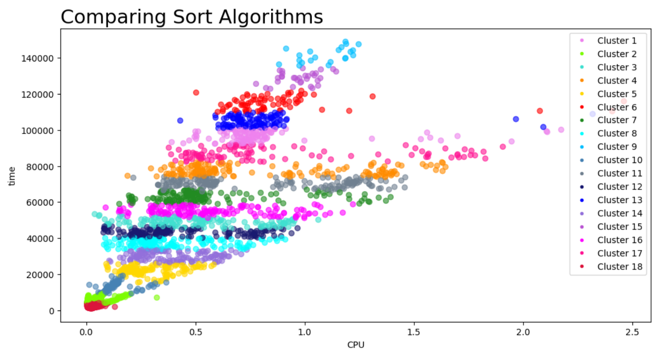


Figure 18. K-Means (18 clusters) - CPU x Time

Thus, we decreased the number of clusters to 2, representing the two languages of the programs we had implemented: C and Python. With this, we obtained the following result where we can see the supposed division between this categories:

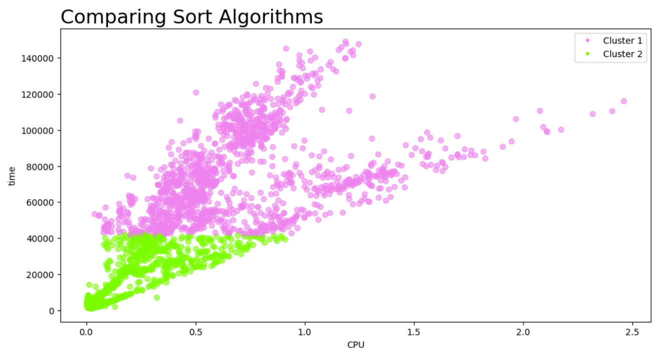


Figure 19. K-Means (2 clusters) - CPU x Time

To verify if these clusters were well build, we compared the results with the scatterplot of the data distinguished by Language:

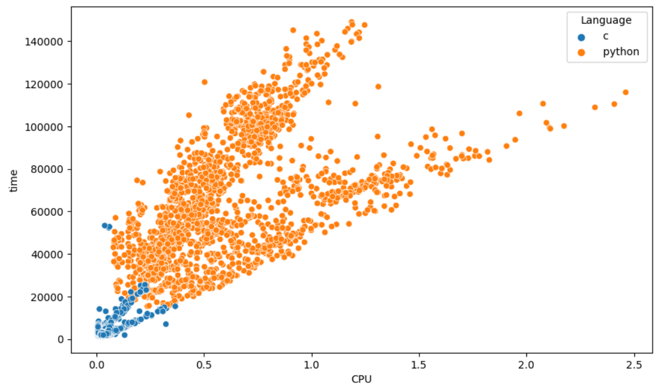


Figure 20. Data distinguished by Language

After analyzing these two scatterplots, we concluded that the clustering did not perform well, since it considered too many elements of the Python language to be from C.

Because of this, we wanted to give it one last try, by executing the algorithm but now with **6 clusters**, which are supposed to be the three input sizes in each language. We obtained the following results:

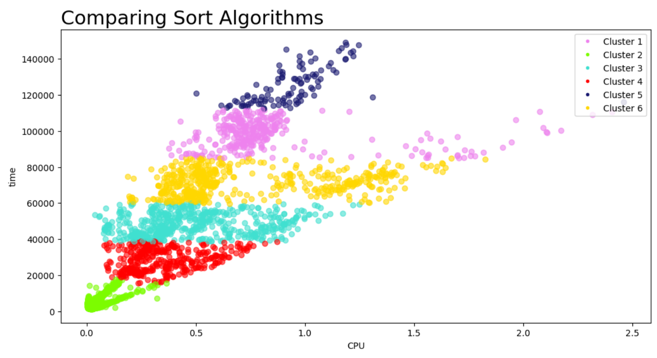


Figure 21. K-Means (6 clusters) - CPU x Time

Again, with the purpose of verifying the results above, we compared the results with the scatterplot of the data distinguished by input size, being 0 (small), 1 (medium) and 2 (large). We got the following result:

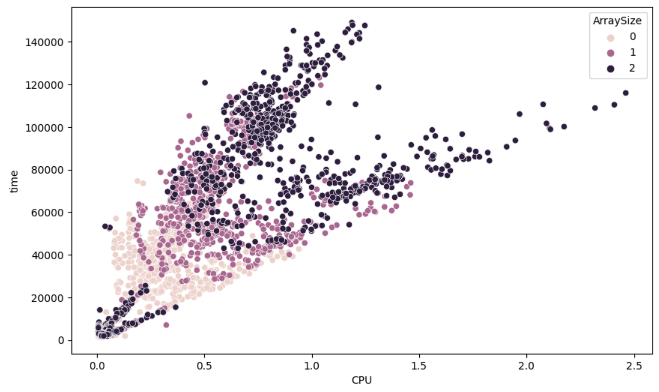


Figure 22. Data distinguished by Input Size

After analyzing these two scatterplots, we concluded that the clustering did not perform well once again, since it considered the C language area with the three input sizes as one cluster and divided the Python Language in five clusters, which does not match with the intended result.

After analyzing all the results, we concluded that the K-Means algorithm did not perform well in this particular context, due to its need to pre-define the number of clusters and the consequent "layered" clustering that it performed, which resulted in confusing and sometimes misleading clustering.

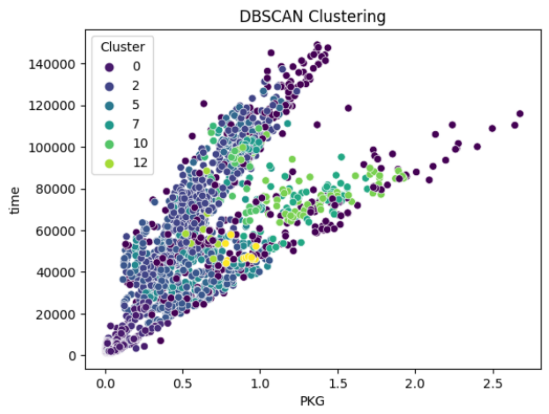
5.7.2 DBSCAN

DBSCAN is a density based clustering algorithm. This meaning: it is an algorithm that uses density as the criteria for clustering, allowing to find clusters of other shapes than spherical.

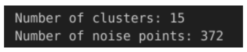
Before beginning the clustering process, it is reasonable and suggested that the data should be normalized before any of the clustering, therefore, the *StandardScaler* was used to allow this normalization.

To see the resulting figure of the resulting clusters it was imperative to change the input parameters for the DBSCAN , mostly due to that fact that , with an epsilon of 0.5 and 5 minimum samples, the resulting number of clusters was 72. However, we do not have that division in the data, therefore, after some tries, the resulting parameters was 0.7 and 7 for epsilon and minimum samples correspondingly, reducing the number of clusters to 15 (closer to 18 different programs). This cutback is easily explained: the epsilon determines the radius, this is, the area considered for analysis of each point. And the minimum samples determines the minimum number of points to stay inside of that area. Therefore, increasing both of this parameters results in the decrease of the number of clusters.

Even though this clustering method allows for non-spherical clusters, the final result does not display the groups of data completely correctly, resulting in the following figure:



Even though the data shows the different clusters that each belong, not all of them are shown in the figure.

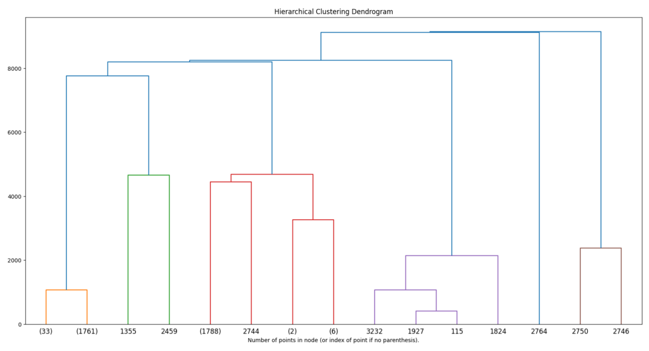


```
Number of clusters: 15
Number of noise points: 372
```

In summary, this clustering method has proved to be less than ideal for the dataset in question, as the clusters created are not even similar to the ones to be expected.

5.7.3 Agglomerative Clustering (bottom-up construction)

As opposed to the previous clustering, the Agglomerative Clustering approaches the data in a Bottom-Up order. This means that it starts by having the same number of clusters as the number of objects. It will then proceed to find the best matched between them and join the best matches into the same cluster.



As we can confirm, the final division is based on 15 leafs, which is very close to the number of programs that were used in the data set (18 programs). We can then conclude that a bottom-up hierarchical clustering approach did indeed return good results.

6 Multi-Criteria Optimization

Multi-Criteria Optimization, also known as **Multiple Criteria Decision Aiding** (MCDA), is a field of study of decision-making problems involving multiple conflicting criteria or objectives. It's designed to support decision-makers in situations where there is no single optimal solution, but rather a set of trade-offs between different criteria.

Having concluded the data analysis and having implemented several clustering methods, the group considered that it would be valuable for this study to employ multi-criteria optimization, as taught in the course lectures.

The group decided to compare each algorithm according to each input size, to realize if the preference for what was the best one would change or not. Further, the group decided to also compare the languages using several parameters gathered across all the algorithm implementations.

With an abundance of testing parameters, it was important to de-clutter the data, so, for these scenarios, the majority parameters were left out. The ones that were included were the following:

- **Package:** with the focus of this study being the energy efficiency of the sorting algorithms, this would be one of the most important parameters and the one given the most weight.
- **Time:** although the focus is the energy efficiency, the execution time was still deemed as important as a metric for each algorithm.

The remaining parameters were removed since they not considered to be useful at all for the comparison's being made, or due to the fact that their values were relatively too close for each of the algorithms.

It is also worth noting that the excel spreadsheets used for this analysis were created using the mean of each of the data parameters.

6.1 Language Comparison

First, the group created a model to compare the Languages used in this study, as to ascertain what was the best language in terms of Energy Efficiency. As such, and as previously mentioned, the group attributed a larger weight to the Package parameter. For this comparison, we defined the indifference as 10% of the subtraction of the maximum value and the minimum value for each parameter. It is worth noting that in some parameters the gap was so big that 10% meant a value larger than the minimum value. As such, it was decided that the indifference value should be 5%. Since there were only 2 candidates being compared, the group considered the veto value to be unnecessary in this particular scenario. Regarding the weight of each parameter, the group attributed the larger weight to Package parameter, as energy efficiency was the focus of the study. The parameter table was set as follows:

	Package	**Time**
Indifference	0,0005	1755,20
Weight	0,65	0,35

The results were as follows:

	C	**Python**
C		1
Python		

From this table we can conclude that C is preferable to Python, a conclusion that is inline with the previous statistic analysis and clustering employed.

6.2 Algorithm Comparison

After a preliminary comparison of the data sorted by language, the focus turned to the sorting algorithms themselves, as such, the group started by comparing each algorithm in terms of small inputs, with and without the Power limitations, and then applying the same methodology for the remaining input sizes. This separation would theoretically allow a further understanding of how a power limitation scenario affects which algorithm is considered best.

As with the language scenario, we started by defining the indifference value as 5% of the difference between each parameter's maximum and minimum values. However, the group acknowledges that there is too big of gap between the values of C and Python algorithms. That being said, the comparison was divided in 2, comparing each algorithm in each language, with a more appropriate value of indifference and veto for each. The veto threshold was defined as the double of the indifference value, so 10%

6.2.1 Small Input Cap On - C

The indifference and veto values are as follows:

	Package	Time
Indifference	0,0002571	1775,513
Weight	0,65	0,35
Veto	0,0005142	3551,026
Lambda	1,00	

The graph that resulted was the following:

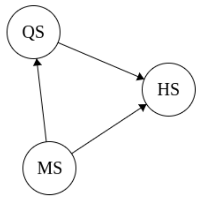


Figure 23. Credibility index graph for Small Input with Cap On in C

Caption: The acronyms for in the graph dots translate to the following:

- **HS** : Heapsort in C
- **MS** : Mergesort in C
- **QS** : Quicksort in C

As we can see, the MergeSort algorithm is the preferred algorithm for these particular circumstances, followed by the QuickSort algorithm.

6.2.2 Small input Cap Off - C

With the comparison made for the Cap On test measurements, we decided to focus on the Cap Off measurements, to see if the power limitations influenced in any way what algorithm is preferable.

	Package	Time
Indifference	0,00050535	21,015
Weight	0,65	0,35
Veto	0,0010107	42,03
Lambda	1,00	

The final credibility index graph is as follows

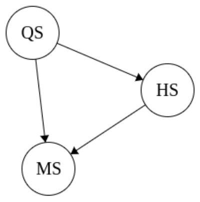


Figure 24. Credibility index graph for Small Input with Cap Off

As we can see, the situation has shifted from when the power limiter was turned on. As such, the preferred algorithm now is the QuickSort.

6.3 Small Input - Cap On - Python

With the first results for the C algorithms gathered, it was time to observe if their Python counterparts behaved similarly. The starting point is, once again, the scenario with the power limitations. We maintained the Indifference and Veto percentages, that resulted in the following values.

	Package	Time
Indifference	0,0085029	985,1875
Weight	0,65	0,35
Veto	0,0170058	1970,375
Lambda	1,00	

The resulting graph is as follows :

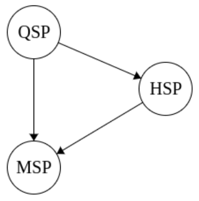


Figure 25. Credibility index graph for Small Input with Cap On in Python

Caption: The acronyms for in the graph dots translate to the following:

- **HSP** : Heapsort in Python
- **MSP** : Mergesort in Python
- **QSP** : Quicksort in Python

As we can see, with the power limitations turned on, the Quick Sort algorithm is the preferred option, with the HeapSort algorithm being preferable to the MergeSort.

6.4 Small Input - Cap Off - Python

After analysing which algorithm is best in a power limitation scenario, it was time to observe if those conclusions changed once the power limitations were turned off.

	Package	Time
Indifference	0,01046585	643,9895
Weight	0,65	0,35
Veto	0,0209317	1287,979
Lambda	1,00	

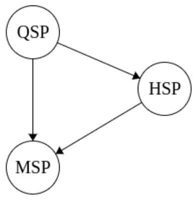


Figure 26. Credibility index graph for Small Input with Cap Off in Python

With this graph we can conclude that switching off the power limitation does not affect the algorithm preference of the Electre method used

6.5 Medium Input - Cap On - C

With the Small input comparisons done, the focus now shifted to the Medium inputs. We maintained the same reasoning for the lambda and weight values as before, with the following results:

	Package	Time
Indifference	0,00009385	27,6775
Weight	0,65	0,35
Veto	0,0001877	55,36
Lambda	1,00	

The final Credibility Index with lambda graph was the following:

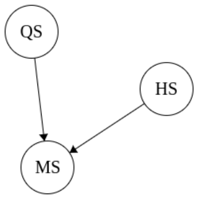


Figure 27. Credibility index graph for Medium Input with Cap On in C

As we can ascertain, for the medium Input, Quick Sort and Heap Sort are no longer comparable, whereas the MergeSort maintains it's status as the least preferable option compared to Quick Sort and Heap Sort.

6.6 Medium Input - Cap Off - C

With the previous results obtained and interpreted, now it was time to verify if there was any chances in preferability between the algorithms now that the power limiter was off.

The indifference and Veto values were as follows:

	Package	Time
Indifference	0,0008803	32,061
Weight	0,65	0,35
Veto	0,0017606	64,122
Lambda	1,00	

The resulting graph is the following:

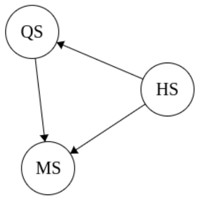


Figure 28. Credibility index graph for Medium Input with Cap Off in C

As we can see, with the power limitation turned off, the HeapSort becomes the preferable option amongst the 3 algorithms.

6.7 Medium Input - Cap On - Python

With the impact of medium inputs analysed for the C algorithms, we now turned the focus to the way medium inputs affects the algorithm choice for the Python algorithms.

	Package	Time
Indifference	0,01252445	1280,7265
Weight	0,65	0,35
Veto	0,0250489	2561,453
Lambda	1,00	

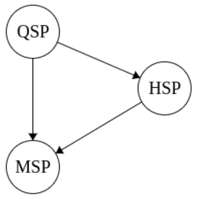


Figure 29. Credibility index graph for Medium Input with Cap On in Python

Once again, we can conclude that the Quick Sort is the preferred option among the 3 choices. The Merge Sort algorithm is once again the worst one.

6.8 Medium Input - Cap Off - Python

It was now time to check if the switching off of the power limitations could have any effects in the preference order visualized in the previous test.

	Package	Time
Indifference	0,012202	666,533
Weight	0,65	0,35
Veto	0,024404	1333,066
Lambda	1,00	

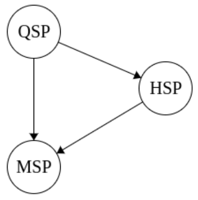


Figure 30. Credibility index graph for Medium Input with Cap Off in Python

As we can see there were no changes once the power limitations were switched off, maintaining what was starting to become a pattern for the Python algorithms.

6.9 Large Input - Cap On - C

With the Medium Input results gathered and interpreted, only the Large Input tests were left for analysing. Once again, the group started by visualizing the the Cap On measurement test results. The indifference and veto values are the following:

	Package	Time
Indifference	0,00079515	69,8675
Weight	0,65	0,35
Veto	0,0015903	139,735
Lambda	1,00	

And the resulting graph is as follows:

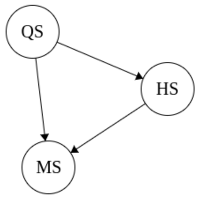


Figure 31. Credibility index graph for Large Input with Cap Off in C

As we can observe, the preferable option is now the Quick Sort Algorithm, followed by the Heap Sort and finally the Merge Sort.

6.10 Large Input - Cap Off - C

After analysing the results with the power limiter on, the only thing left to do was visualize the results with the power limiter off and observe if the preferences changed or not.

	Package	Time
Indifference	0,00066665	48,7255
Weight	0,65	0,35
Veto	0,0013333	97,451
Lambda	1,00	

The resulting graph was the following:

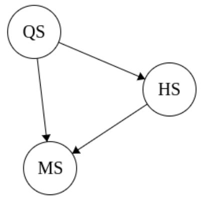


Figure 32. Credibility index graph for Large Input with Cap Off in C

As we can see, there was no change in preferences , with the QuickSort remaining the best option

6.11 Large Input - Cap On - Python

With the last evaluations completed for the C algorithms, the focus switched once again to the Python counterparts for the final batch of test cases. As such, the indifference and veto values where as follows:

	Package	Time
Indifference	0,01430635	1315,9335
Weight	0,65	0,35
Veto	0,0286127	2631,867
Lambda	1,00	

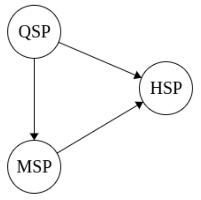


Figure 33. Credibility index graph for Large Input with Cap On in Python

As we can see, there actually is a change from the previous scenarios, with the Merge Sort becoming preferable to the Heap Sort. However, the Quick Sort maintains it's status as the best option.

6.12 Large Input - Cap Off - Python

The last scenario analyzed was the impact of switching off the power limitations for large inputs in the python Algorithms.

	Package	**Time**
Indifference	0,02150905	1137,601
Weight	0,65	0,35
Veto	0,0430181	2275,202
Lambda	1,00	

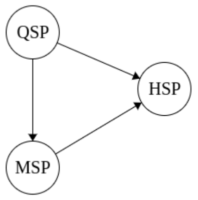


Figure 34. Credibility index graph for Large Input with Cap Off in Python

As we can see, the results were the same once the power limitations were switched off. However, it is interesting to point out that these results solidify the change from Merge-Sort to HeapSort as the least preferable option amongst the 3 algorithms.

7 Conclusions

This study examined the energy efficiency of sorting algorithms implemented in two programming languages. Through ranking the algorithms and conducting statistical analysis on the collected data, valuable insights were gained regarding their energy consumption and impact on computational systems.

Three sorting **algorithms**, including Heap Sort, Merge Sort and Quick Sort, were implemented and executed in C and Python. Energy consumption was measured accurately using RAPL, ensuring reliable data for analysis.

Regarding the two **languages** chosen, besides only having two options, we already had expectations on which language would be the most energy efficient (C), given the already documented difference in performance between the two.

The **statistical analysis** revealed that **Quick Sort** implemented in **C** exhibited the highest energy efficiency, closely followed by Heap Sort also in C. These algorithms achieved faster sorting times while consuming relatively less energy, making them favorable choices.

The statistical analysis conclusions were largely supported by the extensive analysis conducted using multi-criteria optimization. This analysis allowed to visualize, for each input size and each language, with power limitations both on and off, which algorithms were best in which scenario. These various test options allowed for a more complete analysis of how each of the circumstances may or may not alter the preferability of the 3 algorithms used, while laying the groundwork for further expansion of the study.

Overall, this study has allowed to confirm C's place as much faster and more efficient option as a whole, while also proving that the quick sort algorithm is the fastest and more eco-friendly algorithm in both languages used.

In conclusion, in terms of future work, the aim would be to expand this work by considering additional sorting algorithms and programming languages, as we considered that both the statistical analysis and the analysis using multi-criteria optimization done in this study provide a robust and complete basis for further implementations.

References

diptangsu. 2021. Sorting-Algorithms. https://github.com/diptangsu/Sorting-Algorithms

Isaac Gouy. 2023. The Computer Language 23.03 Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2019. Energy Efficiency across Programming Languages. https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf