MICROBENCHMARK

Performance comparison of Python, Java and C++

Drăgan Iulia - Andreea

Contents

1. Introduction 2
   1.1. Context 2
   1.2. Specifications 2
   1.3. Objectives 2
2. Bibliographic study 2
   2.1. Memory management 3
      2.1.1. Python 3
      2.1.2. Java 3
      2.1.3. C++ 3
   2.2. Thread management 4
   2.3. Benchmarking methods 4
      2.3.1. Java Microbenchmark Harness (JMH) 4
      2.3.2. Google Benchmark library for C++ 4
      2.3.3. Python timeit and cProfile modules 5
3. Analysis 5
   3.1. Problem solving analysis 5
   3.2. Benchmarking examples 5
      3.2.1. Java 5
      3.2.2. C++ 6
      3.2.3. Python 7
4. Design 7
   4.1. C++ 8
   4.2. Java 9
   4.3. Python 10
5. Implementation 11
6. Testing and validation 14
7. Conclusions 45
8. Bibliography 46
9. Appendix ..... **Error! Bookmark not defined.**

Extra: Project plan 47

1. Introduction

1.1. Context

The goal of this assignment is to design and implement a benchmarking program in order to compare the performance of several programming languages of choice, i.e. Python, Java and C++. The three programming languages will be evaluated by measuring and analyzing their response to similar snippets of code, which aims to reduce discrepancies and to obtain results that are as accurate as possible.

The assessments will be conducted on the same machine / computer and they will aim to study the behavior and the response of the system for each of the languages. In order to perform a successful comparison, the program should thoroughly evaluate the following aspects:

- Memory allocation
- Static and dynamic memory access
- Thread creation
- Thread context switch
- Thread migration

1.2. Specifications

The program will be implemented using specific IDEs for each of the three programming languages, with the latter possibility of easily comparing them based on the conducted tests. The results, displayed either grouped by language or evaluated characteristics can then be further interpreted by the user / developer. The focus will mainly be on the elements mentioned in the previous subsection.

1.3. Objectives

As specified above, the objective of this project is to successfully and effectively design a program whose purpose is to carry out measurements of similar pieces of code on three different programming languages and to correctly study the results and present the conclusions in a precise, comprehensive manner.

In order to achieve this, it is required to implement several algorithms of varying complexities which will highlight the required aspects. Therefore, it is necessary to develop code which will make use of them, by allocating / accessing memory in several ways and manipulating threads in order to study their performance and behavior. It is also possible to analyze similar data structures and compare their efficiency for a more detailed examination.

2. Bibliographic study

In computer science, benchmarks are a program / a set of programs whose purpose is to assess and evaluate the performance characteristics of computer hardware by running several tests and trials. (1)

Microbenchmarks are specifically designed to measure the performance of a specific piece of code, which aids in studying the differences and similarities between two or more programming languages.

Particular aspects of each of the three mentioned programming languages will, such as memory and thread management, will be further discussed in the following subsections.

2.1. Memory management

- Python

Memory management involves a private heap which contains Python objects and data structures. The management of this heap is ensured by the Python memory manager, which handles several dynamic storage management aspects, i.e. sharing, segmentation, pre-allocation and caching. (2)

Since Dynamic Memory allocation underlies the memory management, when an object is no longer needed, the memory manager will automatically reclaim memory from them. Memory is not necessarily directly released into the operating system, but returned to the python interpreter.

In terms of structure, there is a private heap that contains all Python objects and data structures. This heap is managed on demand. The memory manager has specific allocators to allocate memory distinctly for different data structures.

Methods and variables are created in Stack memory. A stack frame is created whenever methods and variables are created and destroyed automatically when the methods are returned.

Objects and instance variables are created in Heap memory. When the variables and the functions are returned, dead objects are garbage collected. (3)

- Java

Memory management in Java is also done automatically and is divided into two major parts: JVM Memory Structure and the Garbage Collector. JVM creates various run time data areas in a heap, which are then used during the execution. They are destroyed when the JVM exits, while the data areas are destroyed when the thread exits. (4)

Stack memory is responsible for holding references to heap objects for storing value (primitive) types, holding the value itself rather than a reference to an object from the heap.

Heap memory is the part of memory which stores the actual object and is created when the JVM starts up. It is sometimes divided into two areas, called the young space and the old space. The reasoning behind this is that most objects are temporary. Therefore, the young collection defined to be swift at finding newly allocated and moving them to the old space, to free up memory for new allocations. (5)

- C++

The basic memory architecture in a C++ program consists of:

- Code Segment – contains the compiled program with executive instructions
- Data Segment – global variables and static variables
- Stack - pre-allocated memory; stores local data, return addresses, arguments passed to functions, memory status
- Heap – allocated during program execution using the new operator and deallocated using the delete operator

Since there is no automatic garbage collection as in Java and Python, it is possible that the program might suffer from memory leaks due to mismanagement of memory allocations and deallocations, especially in the case of

dynamic memory allocation. (6) The performance of manual garbage collection in comparison to the automatic one will be studied further using benchmarks.

2.2. Thread management

In terms of threads, they are the smallest sequence of programmed instructions that can be managed independently by a scheduler. Their implementation differs between operating systems. Multiple threads can exist within one process, executing concurrently and sharing resources. (7) Their behavior and performance will be studied for each language individually, by examining multithreading and scheduling mechanisms.

2.3. Benchmarking methods

The benchmarking process itself can be done independently for each programming language by highlighting key differences and similarities between the same algorithms, as well as similar data structures. This is a short introduction to what each one is capable of measuring and displaying to the user.

Before introducing the mechanisms through which the benchmarking processes will be performed, it is essential to present two keywords which will have to be taken into consideration when designing the benchmarks:

- Dead code – it is likely that the compiler will eliminate dead code, i.e. the program does no longer perform operations or declarations which will then not be used.
- Constant folding – it implies that a calculation based on constants will produce the same result; for this reason, the code might be optimized in such a way that the result will be predefined. (8)

The compilers will try to optimize these two aspects as much as possible, therefore it is important to perform the operations such that this will be avoided in order to perform a correct evaluation.

- ### Java Microbenchmark Harness (JMH)

JMH is a powerful tool for benchmarking specific pieces of code in a Java program. Essentially, it offers the necessary properties to efficiently measure the performance as simple as possible, by handling Java Virtual Machine warm-up and code-optimization paths. (9)

Integrating the JMH into a Java Maven project can be done quite easily, simply by specifying the required dependencies. Afterwards, the programmer is free to use all the features the API provides. Avoiding code optimization can be done either manually or making use of the JMH Blackhole objects.

- ### Google Benchmark library for C++

Google Benchmark is a library which can be used to examine code snippets, similarly to unit tests. It is a powerful tool used to measure the execution time in C++ on a different number of iterations. It also offers customization options, allowing the user to create their own counter and to evaluate specified attributes.

The library can be found on GitHub, along with the installation steps and the features it offers for efficient benchmarking.

- Python timeit and cProfile modules

While Python does not offer a specific benchmarking module, there are different ways in which the performance can be analyzed and evaluated. Two of these methods entail using the tpytest library. What it does is to measure the performance of each function marked with a "test_" at the beginning.

3. Analysis

3.1. Problem solving analysis

It is important to note that there are certain challenges which will have to be overcome, besides the basic features which have to be evaluated. While memory management is quite simple to analyze and evaluate, problems arise when dealing with threads, more specifically, Python threads.

Since the Global Interpreter Lock allows only for one native thread to execute at once, Python threads are always experiencing context switching. Therefore, it is difficult to ensure thread parallelism. A possibility might be to use the multiprocessing.dummy library from the Pool module. (12)

In terms of context switching in the other programming language, it is possible to simulate the action by implementing a variant of the Producer-Consumer problem, where one thread is dependent on the result of the other one.

3.2. Benchmarking examples

The examples below have been done purely for experimental purposes and they will not necessarily be added to the final benchmark comparisons. They were presented to highlight the usage of the aforementioned mechanisms.

- Java

It is important to note the following aspects and annotations (Please note that these are not all of them. More will be presented as they are used in the project). (9)

- @Benchmark – specified the function which will be evaluated by the benchmark mechanism
- @BenchmarkMode – as presented above, it helps choose between the available benchmark types
- @State – state variables help eliminate the possibility of constant folding; the declaration of state variables is not measured by the JMHpackage newPackage;
- Blackhole – helps eliminate dead code by consuming the unused result. This way, the operation will not be optimized by the compiler and it will instead be performed.

Further annotations will be presented as they are used within the project.

```
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

public class MainClass {

   @State(Scope.Thread)
```

```
    public static class States {
        public int a = 2;
        public int b = 3;
    }

    @Benchmark
    public void test(Blackhole blackhole, States state) {
        int sum = state.a + state.b;
        blackhole.consume(sum);
    }

    public static void main(String[] args) throws Exception {
        org.openjdk.jmh.Main.main(args);
    }
}
```

```
Benchmark                    Mode  Cnt          Score          Error  Units
newPackage.MainClass.test   thrpt   25  193076882.495 ± 13912288.159  ops/s
```

- C++

To be implemented. The code presented below can be found in the examples sections of the GitHub page of the library.

It can easily be noted that the library can be used by including the <benchmark/benchmark.h> library into the project. It is then required to define the benchmarking functions, in our case, BM_StringCreation and BM_StringCopy which will receive the required parameters for benchmarking.

In order to run this code, the BENCHMARK(function_name) is used to register the benchmark, followed by the BENCHMARK_MAIN() function which replaces the traditional main function of a C++ program.

```
#include <benchmark/benchmark.h>

static void BM_StringCreation(benchmark::State& state) {
  for (auto _ : state)
    std::string empty_string;
}
// Register the function as a benchmark
BENCHMARK(BM_StringCreation);

// Define another benchmark
static void BM_StringCopy(benchmark::State& state) {
  std::string x = "hello";
  for (auto _ : state)
    std::string copy(x);
}
BENCHMARK(BM_StringCopy);

BENCHMARK_MAIN();
```

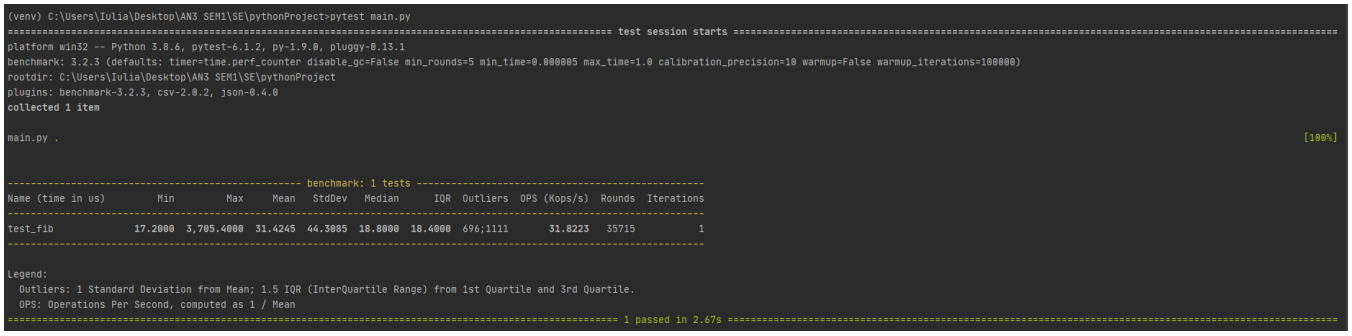
Results example:

```
Benchmark                                  Time(ns)    CPU(ns) Iterations
-------------------------------------------------------------------------
BM_SetInsert/1024/1                           28928      29349      23853  133.097kB/s   33.2742k items/s
BM_SetInsert/1024/8                           32065      32913      21375  949.487kB/s   237.372k items/s
BM_SetInsert/1024/10                          33157      33648      21431  1.13369MB/s   290.225k items/s
```

- Python

The following code (The nth Fibonacci number) was evaluated to display the results generated by each module:

```
def fib(n):
    if n <= 0:
        return -1
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```



4. Design

The main idea for the project is to divide each program in a series of closely-related sequence of tests which will highlight the similarities and the differences between both the programming languages as separate entities and within one programming language.

As an example, memory access of static arrays might differ from one programming language to another, fact which can be observed through the in-depth post-implementation analysis. However, it is also possible to differentiate between certain data structures within each of the three languages, such as ArrayLists and LinkedLists in Java, or vectors and lists in C++, as each have their own insertion / removal / access mechanisms.

Therefore, the main modules / features that will be tested can be divided in the following section (if the language at hand allows for such analysis and implementation):

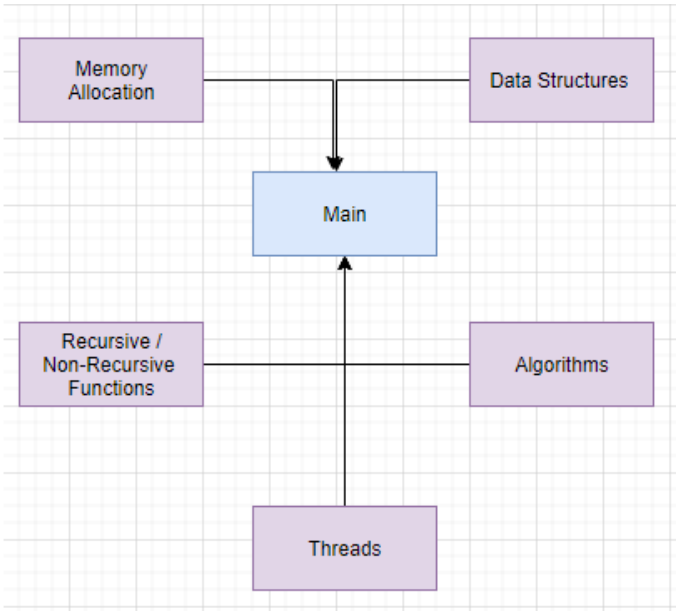
- Memory allocation and usage – static and dynamic – can mostly be observed in C++

- Predefined data structures – common data structures such as lists will be compared both to each other and to their respective correspondent from the other programming languages, e.g. lists
- Threads – while multithreading is difficult to examine in Python, both Java and C++ offer support for such actions. Context switches can be easily simulated by pausing and resuming a certain thread, whereas thread migration might prove to be difficult but attainable in Java.

Further will be presented a rough sketch of the collections of measurements which will be considered for each programming language, with the mention that additional details will be added as the implementation phase progresses. Should a language not allow a certain set of measurements to be performed due to incompatibility, the plan and the design will be subjected to minor changes.

It is also essential to note that so far in the development process, C++ is the only one which gets measured on a virtual machine, through VMware workstation, as compared to the other two programming languages, whose performance is currently tested on the main operating system, i.e. Windows 10. This might be modified in the future in order to offer a data set as accurate and close to reality as possible either by porting all the measurement sets on the virtual machine or by implementing the C++ set of measurements on Windows 10.

A general idea of the structure of the project would be:



Please note that this is a brief example and will be further enhanced by detailed UML diagrams once the implementation stage is complete.

4.1. C++

Since at its core, C++ is an enhancement of C, an inherently lower-level programming language, with additional object-oriented features which improve the versatility of the C family, it is somewhat apparent that it will offer the most measuring possibilities out of all the three languages which will be analyzed through this paper.

Therefore, a rough sketch of one of the design possibilities for displaying the time efficiency of certain pieces of code would be:

- A source file whose purpose is to study the efficiency of simple memory allocation through static and dynamic methods, as well as accessing objects of such nature. This set of, albeit quite simple, functions follow to portray the differences in performance in the most mundane operations, i.e. variable / array / object declaration and initialization, to further prove the effort necessary to utilize a variable in a static and a dynamic way respectively.

- A set of functions which highlight the time necessary to operate on predefined data structures and classes, such as lists, vectors, stack, queue, in order to observe the advantages and disadvantages of each of them and compare them with similar ones from the other two programming languages.

- A set of simple and more complex functions, both recursive and non-recursive, to measure the performance of C++ on larger snippets of code and to properly compare it with that of the other two. The main function ideas for this batch of operations would be simple mathematical operations (additions, subtractions, multiplications, divisions, min / max functions), as well as functions which require the implementation of loops and conditional clauses, such as the common divisor problem. Examples of recursive functions would be calculating the nth factoria. Other operations might include string manipulation, such as verifying if a given sequence is a palindrome.

- Lastly, since C and C++ offer several thread customization and manipulation features, it is possible to measure not only the creation of threads and the efficiency of the program in case of multithreading, but also to analyze its behavior in case of artificially induced context switching (through locks and semaphores) as well as defining which core of the processor a certain thread will run on by specifying its affinity.

4.2. Java

Although Java is a few levels above C++, it still offers a plethora of possibilities in regards to testing methods / possible measurements. Since JMH is a powerful benchmarking tool, it is necessary that the developer customizes each benchmark function based on the requirements that the certain function has to meet.

The goal is to measure the average performance time for each function, therefore it is essential to avoid dead code and constant folding as much as possible and to perform the same set of tests on variable numbers of iterations / elements. This can all be done through the JMH API.

In terms of design structure, the main Java benchmarking program will follow a similar structure to that presented above for C++, since it allows, for the most part, the generation of complementary sets of measurements:

- In the case of memory allocation, the difference between static and dynamic memory usage can be portrayed through static arrays and list classes objectively.

- Java also offers the possibility of comparing the aforementioned Collection classes between themselves, in order to make an evaluation of the performance differences between Objects such as LinkedLists, ArrayLists and TreeSets, HashSets. To attempt to measure the differences of similar object structures between Java and C++, for example, it is possible to analyze the discrepancies between ArrayLists and

vectors respectively, since they are both formed on the idea of a dynamic array and behave in a relatively analogous manner.

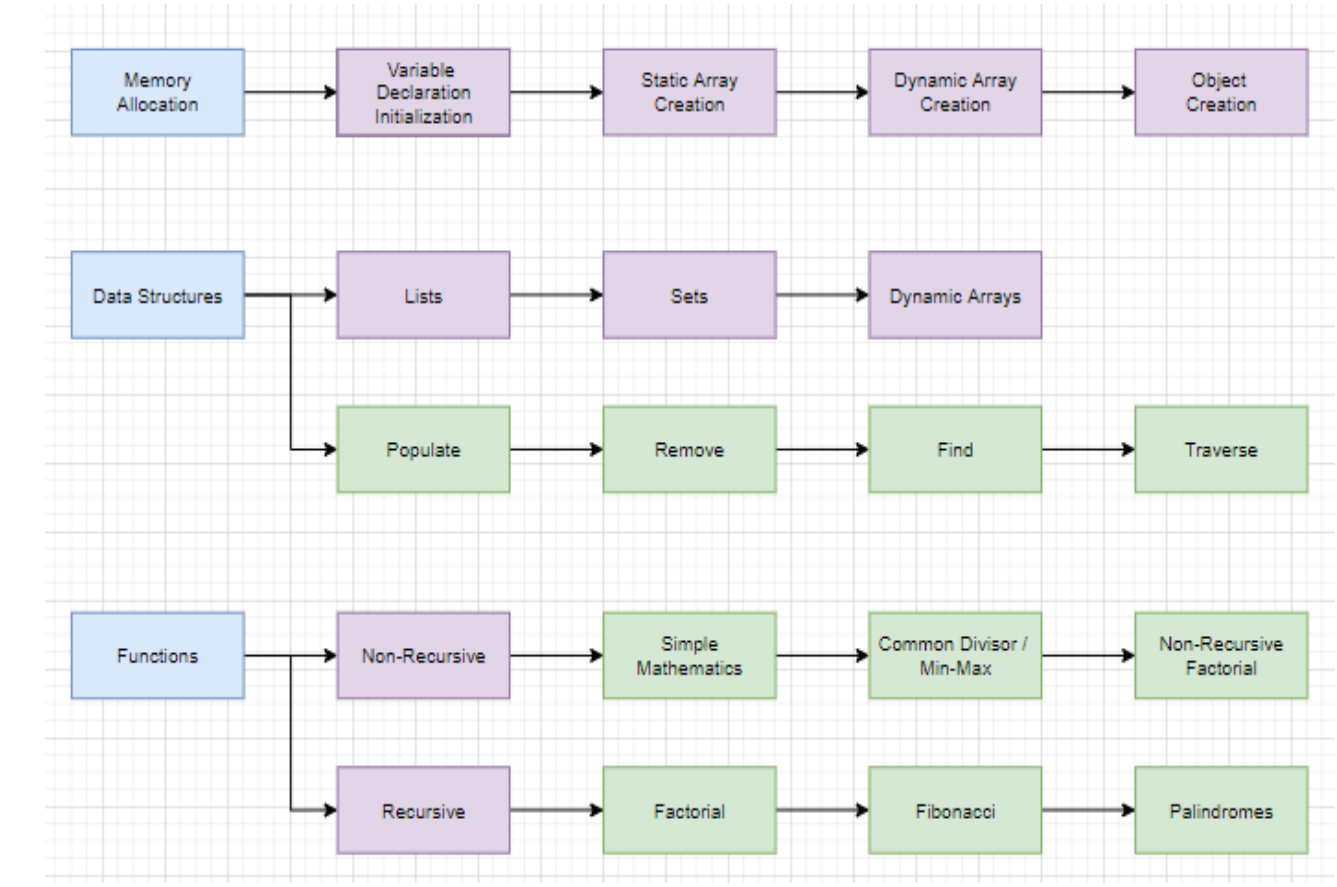
- In terms of threads, while it is possible to simulate both single-threading and multithreading, as well as context switching (through the wait() and notify() methods), it is not possible to do so for the thread migration section of the project. Therefore, thread migration will solely attempt to be tested with the help of the available C++ features.

4.3. Python

Since Python is a high-level programming language, it is more difficult to perform certain sets of measurements, such as static / dynamic memory allocation comparisons and multithreading. However, it is possible to implement all the aforementioned functions and algorithms in order to observe the functionality of a purely dynamic programming language.

Although Python does not inherently offer the possibility of multithreading, thread creation / manipulation, as well as context switching is still very much possible to implement and analyze through the available features.

To summarize the design stage, the following diagram briefly presents the classes / modules / source files that will be implemented as well as their respective capabilities and functions. Please note that this can be subjected to change, i.e. if the overall result ends up being too simple, more tests will be added.



5. Implementation

As stated above each language contains similar sets of tests in order to get measurements as accurate as possible and to reduce the risk of incorrect comparisons.

5.1. Java

In case of Java, each set of tests can be found in its respective package and class. The idea was to create all the necessary tests for each required case. In that sense, there are the following packages: DataStructures, Functions and Threads, each with its own purpose. The results of the tests are generated by running the Main class from the App package.

- ### DataStructures

The purpose of this package is to display and analyze the efficiency of various predefined data structures, such as lists and sets. Therefore, the package is divided into four classes: Creation, Deletion, Find, Insertion, each of

them applied on the following data structures: Array, ArrayList, LinkedList, TreeSet, HashSet, in order to compare then and to visualize the differences in efficiency between them.

- Creation

The Creation class measures the performance of each data structure in terms of creation and populating the structures with equal numbers of elements. Therefore, it contains simple functions with initialize and define the aforementioned data structures.

- Deletion

The Deletion class follows to portray the differences in performance when it comes to deleting a certain element from a data structure. The goal was to test the predefined remove methods for ArrayList, LinkedList, TreeSet, HashSet to delete the middle element from the list / set. In the case of the array, an additional test was made: deletion from the end, since it requires little to no time to perform, as there are no shifts needed within the array.

- Find

The purpose of the Find class is to analyze the performance of attempting to find a certain element in the data structure, whether it is a successful trial or not. Therefore, two sets of tests were made for each data structure: successful find and unsuccessful find respectively, in order to view the time necessary to traverse each data structure until it reaches the required element (in the successful case) or it reaches the end (in the unsuccessful case).

- Insertion

Insertion is similar to the Creation class, the difference being that the methods do not create and populate new data structures, but they insert a single element in an already existing data structure.

- Functions

This package contains two classes: NonRecursive and Recursive, which each come to portray the time efficiency of recursive functions and their non-recursive counterparts.

- NonRecursive

This class contains three benchmark methods: primeNumbers, factorial and commonDivisor. The last two do have a recursive version, which are also evaluated in the Recursive class.

- Recursive

The benchmark methods in this class contain auxiliary methods which are called when performing the benchmark, which is something to be mindful of. The three methods are primeNumbers and commonDivisor, along with stringReversion. The first two were created in order to compare the performance differences between similar recursive and non-recursive functions.

- Threads

Java threads initialization / usage does imitate C++ threads in certain ways, therefore it was possible to use semaphores in order to pause / continue a thread in the case of context switching. Unlike Python, it also allows

for multithreading, therefore there are two existing classes: one for Singlethreading / context switch and another one for Multithreading, which measures the behavior of two threads existing at the same time.

5.2. C++

While the C++ tests are similar to those in Java, it is essentially to note that C++ is more permissive in terms of memory allocation, therefore it was possible to perform tests of static and dynamic memory allocation, along with the tests that are similar to the Java ones. This programming language also allowed more testing possibilities for threads and thread interactions through locks and semaphores.

- ### DataStructures

These sets of tests are closely related to the ones performed in Java, with the following differences: the ability to test the operations on static / dynamic arrays, as well as the option to calculate and measure the performance of the following data structures, as opposed to those from Java: vector, list, queue, stack, priority queue.

- ### Functions

These tests are similar to the ones performed in Java, therefore there is no need to go into detail about the structure and implementation of this section.

- ### Threads

Since C++ is the most permissive language in terms of thread manipulation, it also meant that most thread tests were performed in this programming language. Therefore, there are three main test groups to analyze and visualize: running a single thread, context switch simulation and multithreading (running several threads in parallel. The threads were synchronized using semaphores.

5.3. Python

Python is the most high-level language of all three and therefore it is not possible to perform certain tests on memory allocation. It is also Important to note the fact that Python does not generally allow multithreading to happen, which meant that the closest one could get to performing “multithreading” tests in Python is to simulate this behavior.

- ### DataStructures

In terms of data structures, the same set of tests as in the other programming languages was performed: Creation, Deletion, Find, Insertion. Since memory management is fully ensured by the Python compiler, it is not possible to manually assign memory to structures such as arrays. Therefore, this set of tests was performed on the following data structures: Lists, Sets and Dictionaries, in order to highlight the performance differences between them.

- Functions

Also similar to those performed in the other two languages. It is however important to note that certain functions were too data-heavy to be performed for a large amount of input data, therefore those particular tests are missing.

- Threads

It is known that Python does not allow multithreading in its traditional form, since it can only handle one thread at a time. This is ideal in terms of context switching, but attempting to simulate multithreading poses certain difficulties. Therefore, in the case of multithreading, one way to simulate it would be to use a Threadpool object, a child class of the Multiprocessing library.

6. Testing and validation

While the tests were performed as accurately as possible, there is always the possibility that they will not only differ from one another, but also generate some errors or deviations based on the current state and / or workload of the computer, as well as its specifications.

The test results will firstly be presented for each programming language, followed by comparative graphs of all three programming languages, grouped based on the implementation grouping criteria. The time measurement unit used in this case is the nanosecond.

To be noted that the lowest value on the axis represents the smallest resulting value on the graph. (i.e. the origin is not 0).

The tests were created using the matplotlib library and the pandas module in python.

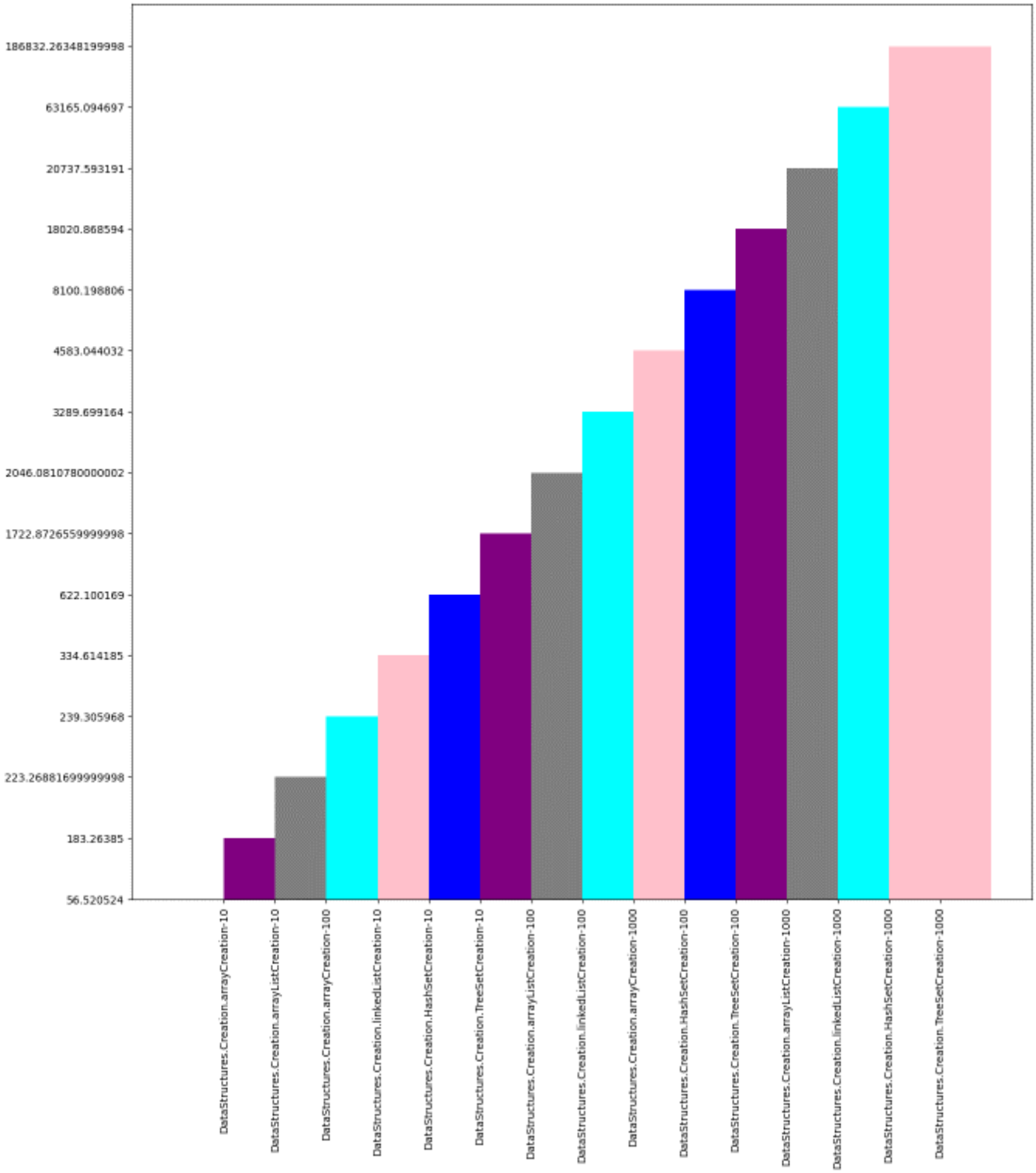
6.1. Individual Results

6.1.1. Java

- DataStructures

- Creation

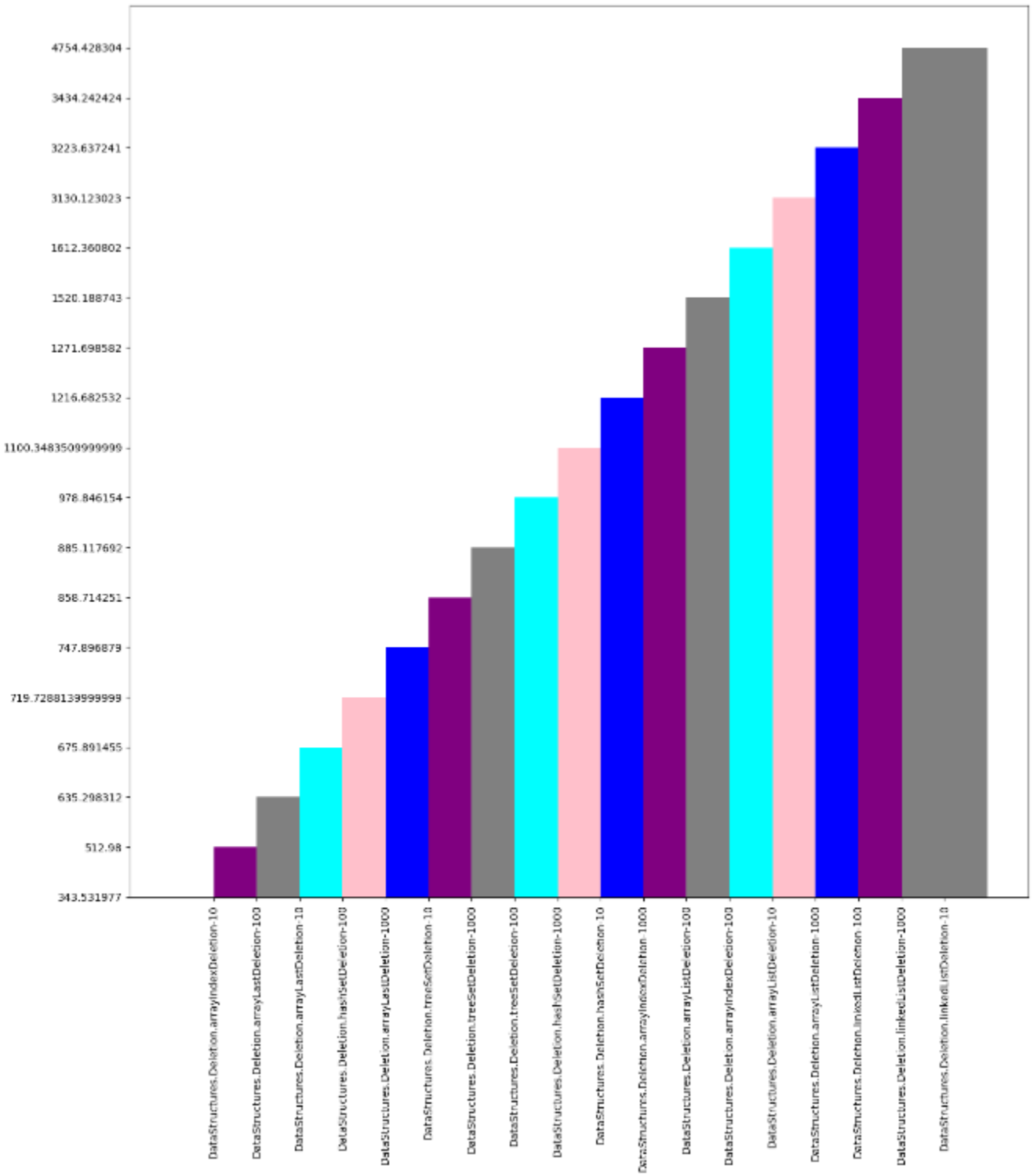
In the following graph, the differences between the data structures can easily be noticed. It is only natural that simpler data structures, like arrays, are overwhelmingly more efficient in terms of time when it comes to creation and initialization. Since ArrayLists are array-based, they are the closest in terms of time efficiency with the usual arrays. However, as the data structure becomes more complex, the time necessary to create objects of identical sizes (10, 100, 1000) grows rather quickly. HashSet and TreeSet do take up more time because of the constraints they have to fulfill.



- Deletion

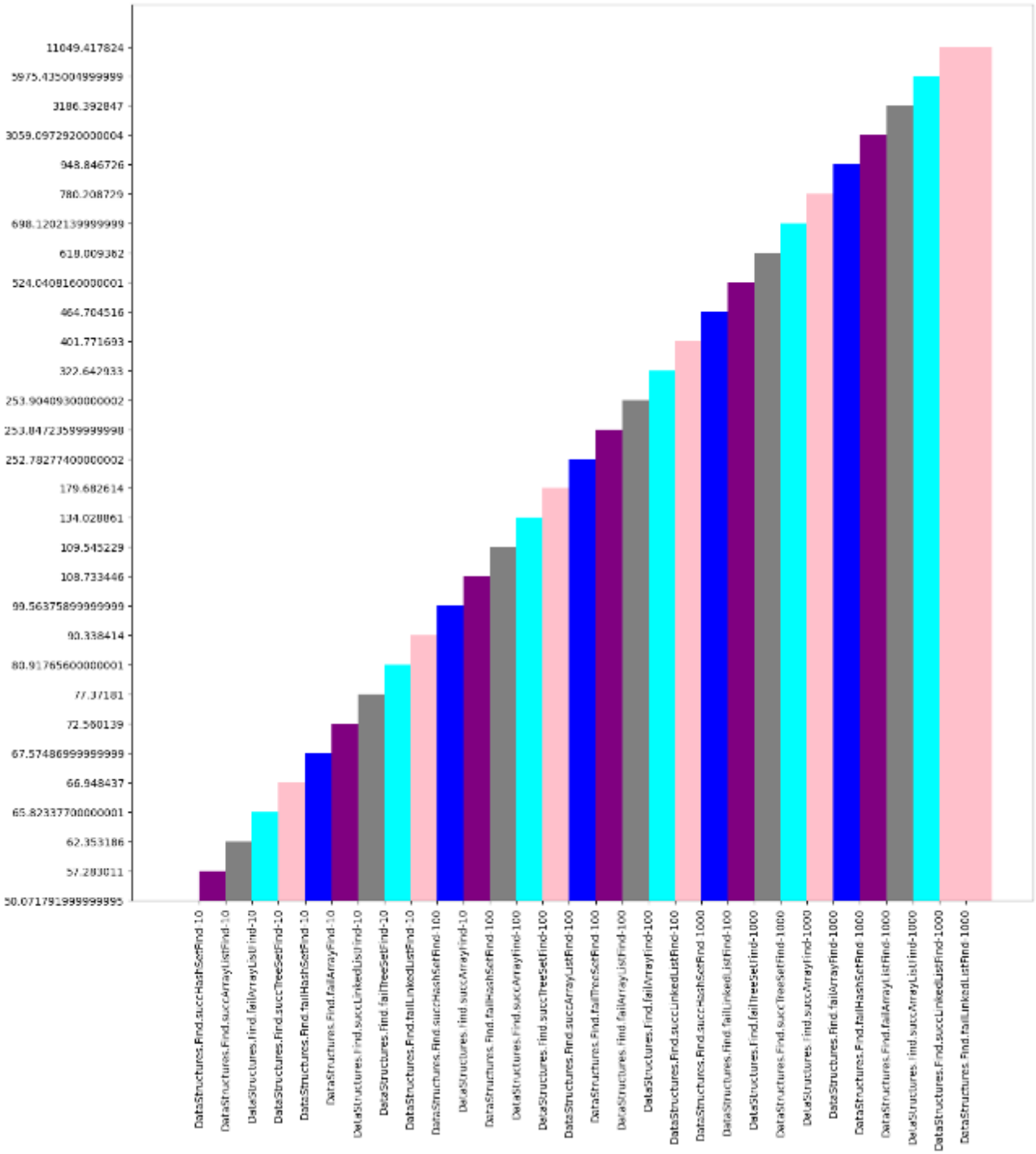
When it comes to deletion, while arrays still prove to be the most efficient in terms of time, in terms of more complex data structures it can be seen that the order has changed quite dramatically. Objects such as HashSet

and TreeSet prove to be quicker than classic objects like LinkedList and ArrayList since finding the required element is a less taxing task.



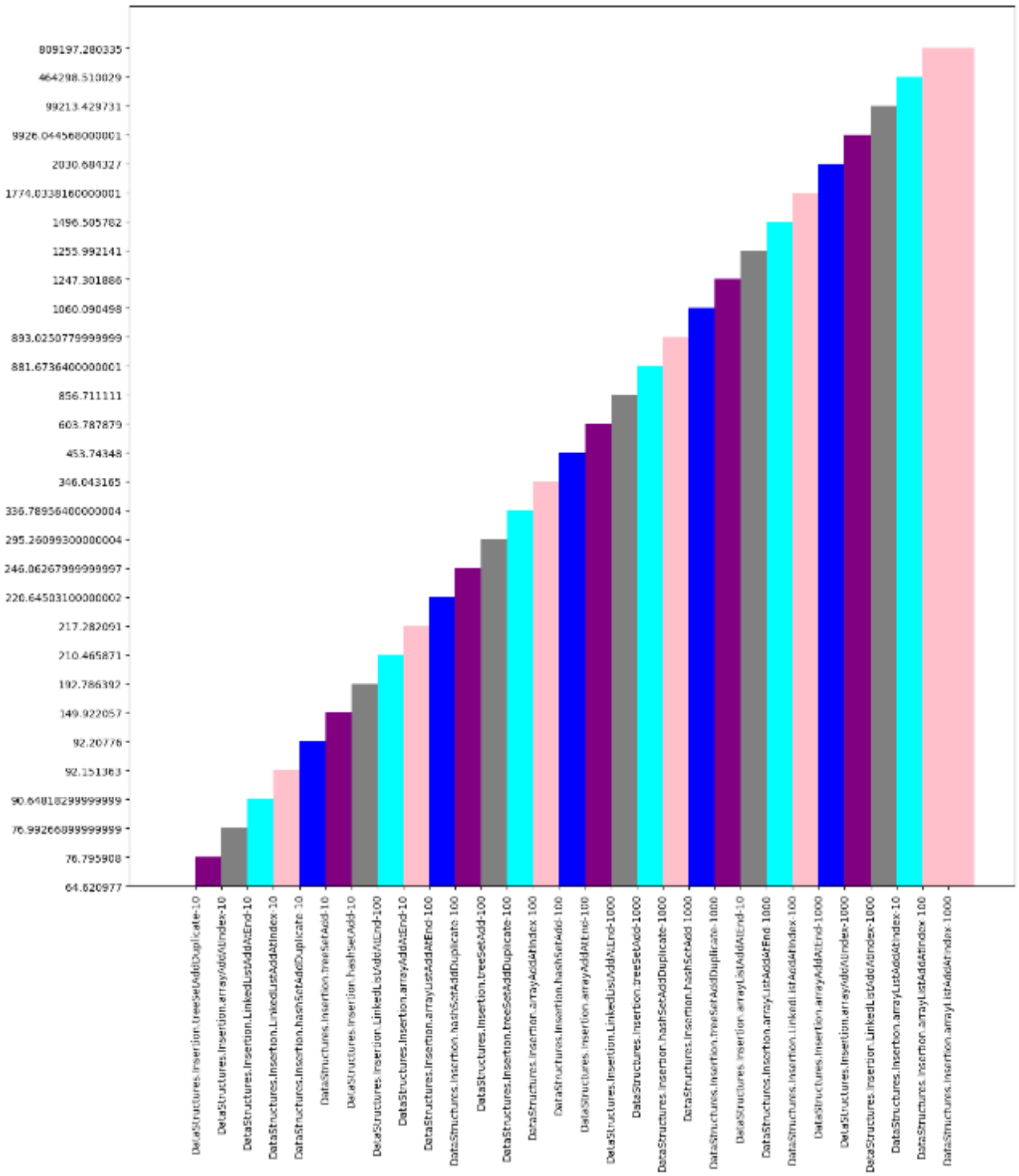
- Find

This is one of the most balanced graphs in terms of steady growth. As with deletion, in terms of finding a certain value in the data structure, it is important to note the way the respective data structures operate, i.e. whether they require a full traversal or there are quicker ways to obtain the result. Generally, failed attempts require a full traversal in the case of simple data structures, such as the lists. This can be seen in the resulting graph, as those kinds of tests took a longer time to perform. Clearly, as the number of elements increases, so does the necessary time.



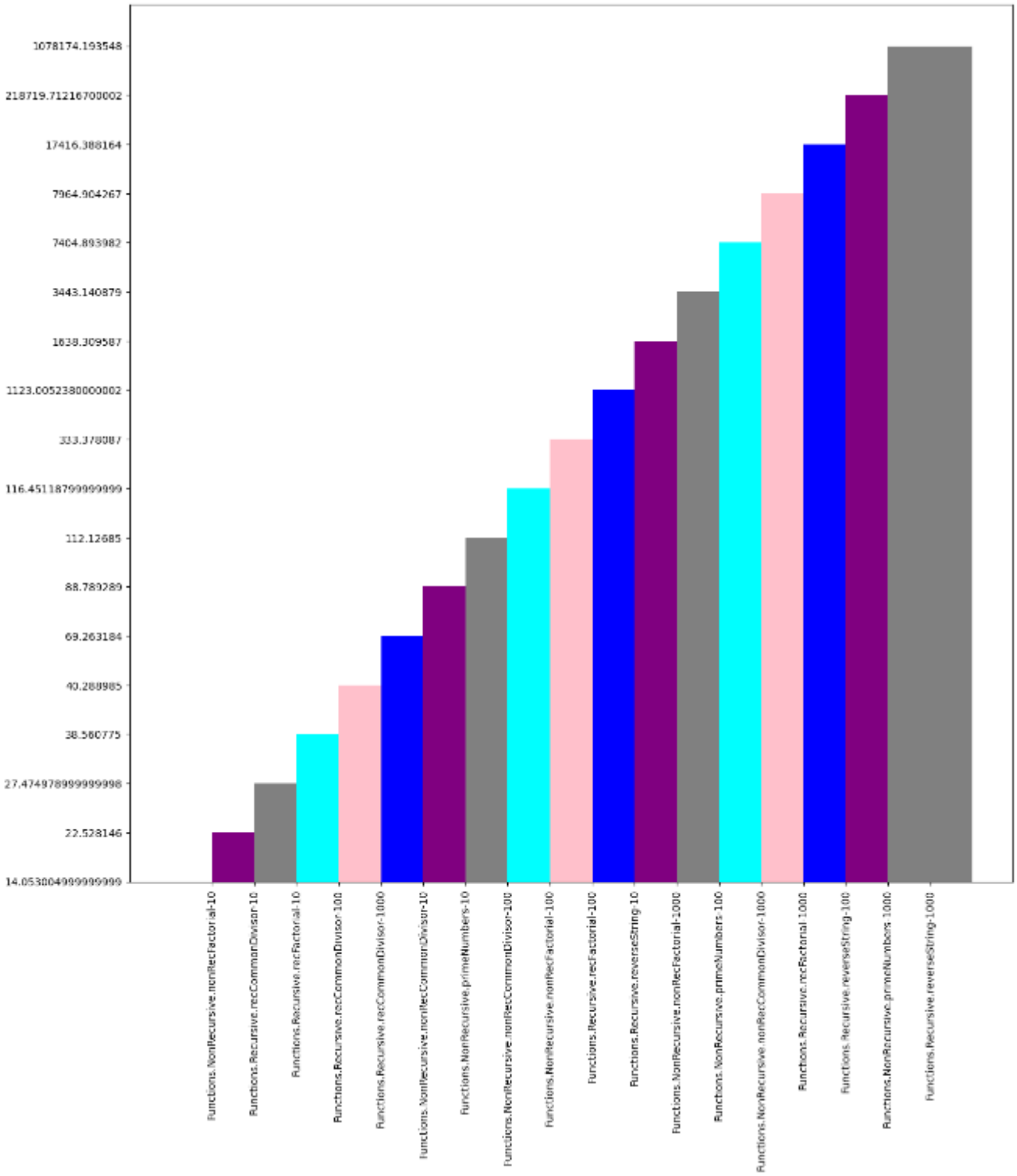
- Insertion

Insertion, depending on the data structure at hand, requires either a slight change in parameters or a full shift of the data structure. An example of the latter would be the classic array, which proves to be on the side of less efficient operations especially in terms of index insertion. On the other hand, LinkedList, since it operates with relationships between elements rather than a fixed order requiring shifting for each insertion operation, proves to be one of the most efficient data structures in terms of inserting a new element.

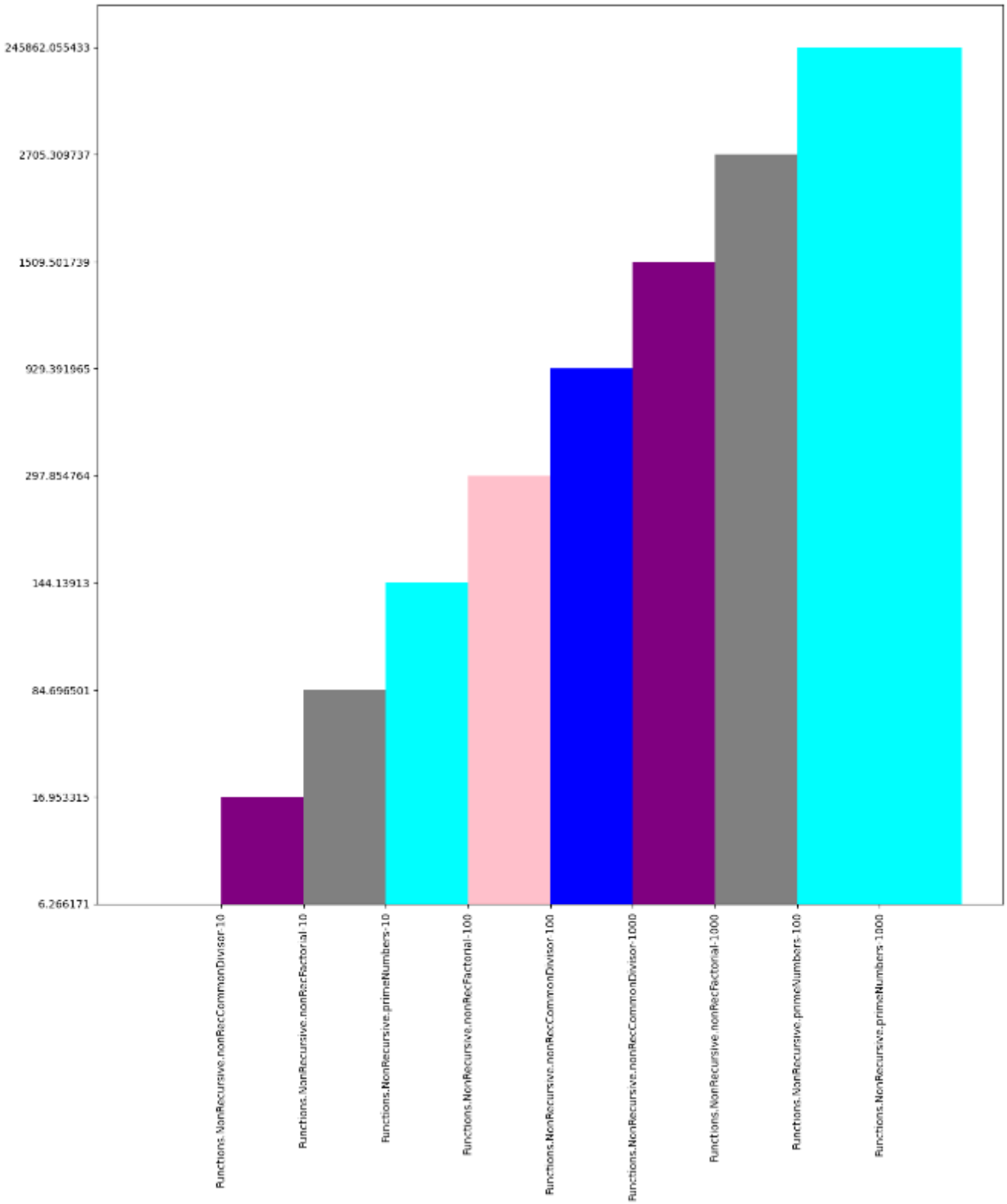


- Functions

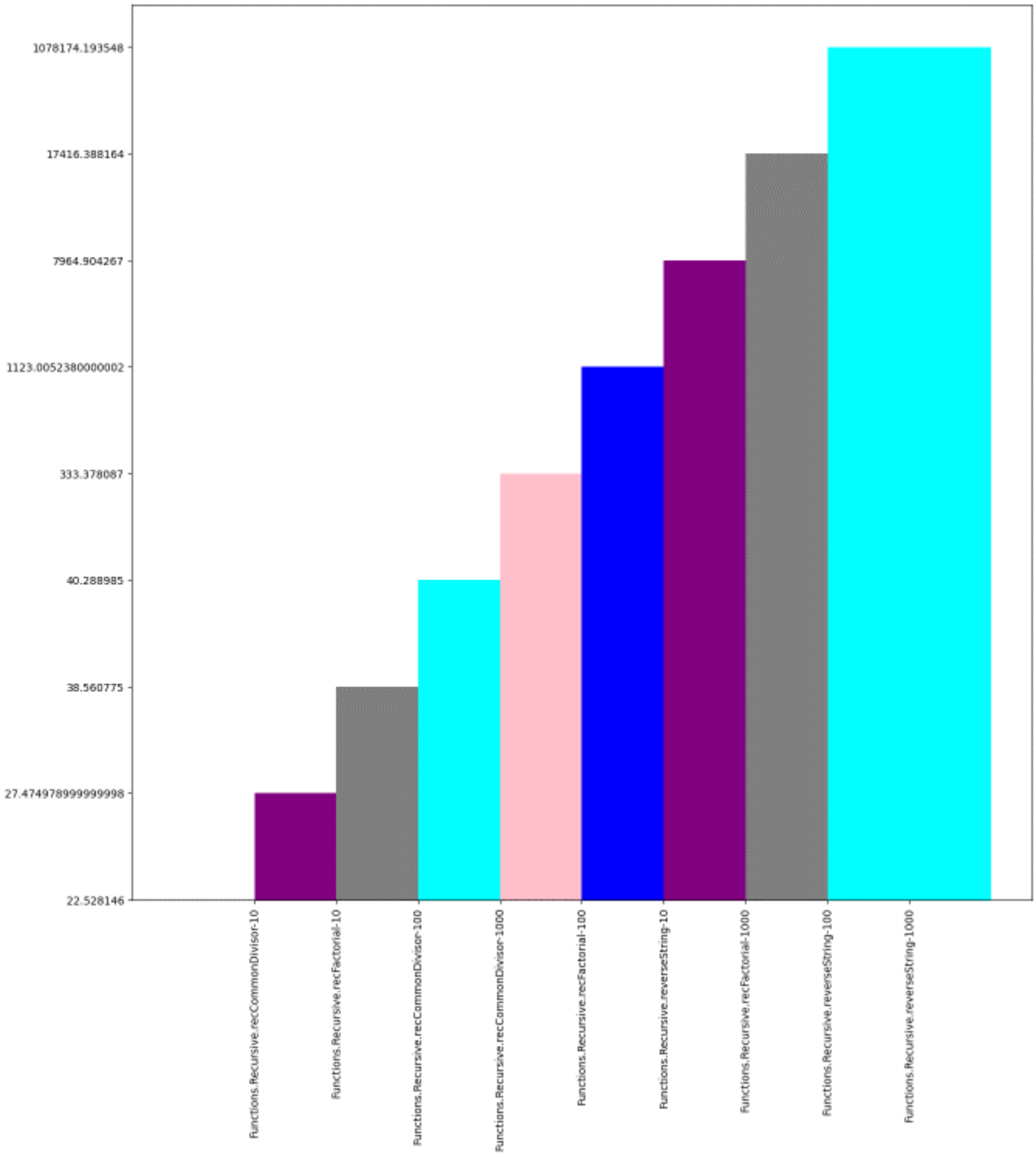
The functions will be compared in terms of recursive versus non-recursive implementations. In this situation, the recursive counterparts of the same functions are, for the most part, more complex in terms of time usage than the non-recursive ones. That is because of the necessary function calls on the stack, which require more time to perform.



- NonRecursive

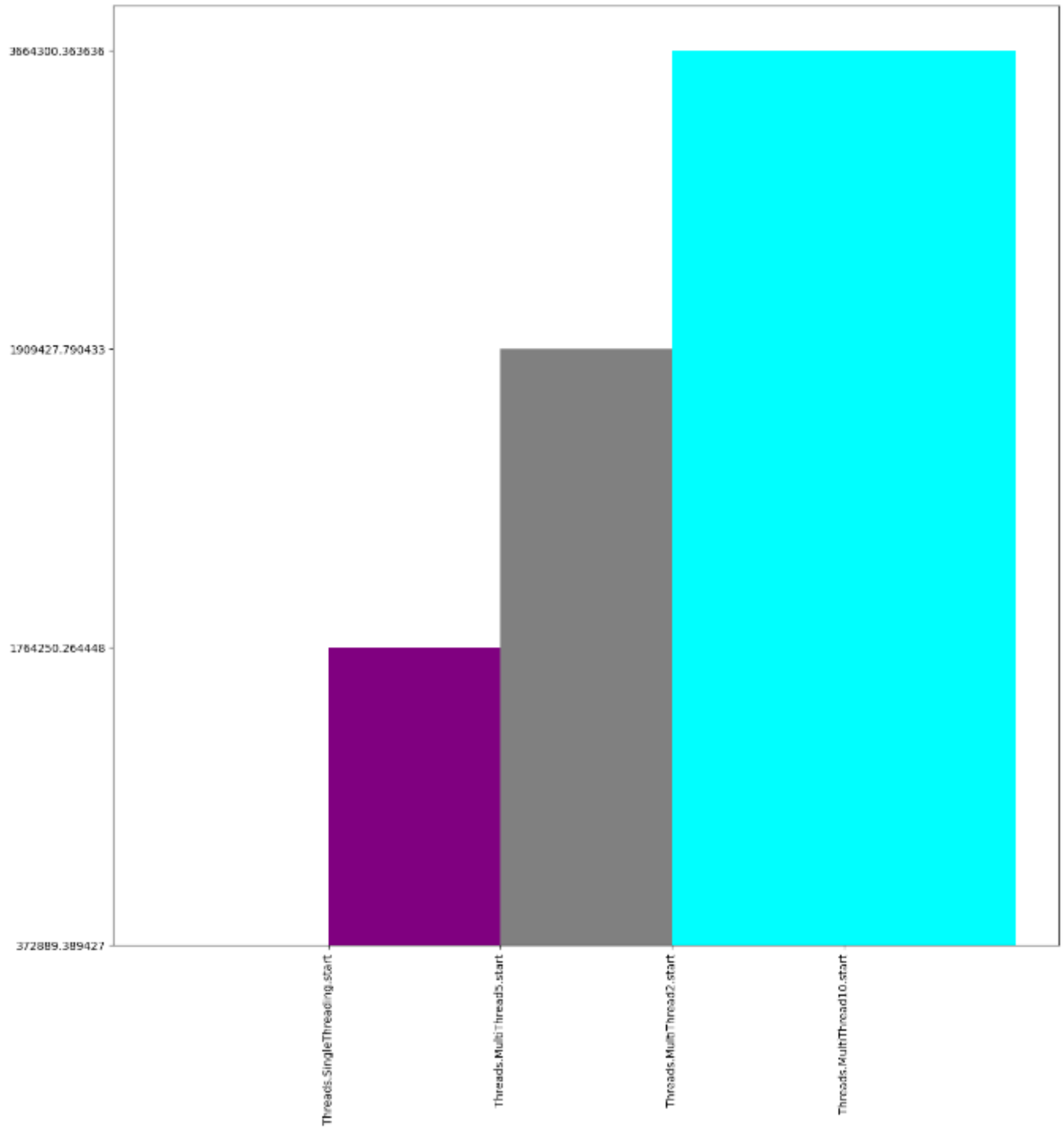


- Recursive



- Threads

In terms of multithreading, it is possible to see that more threads require more time.



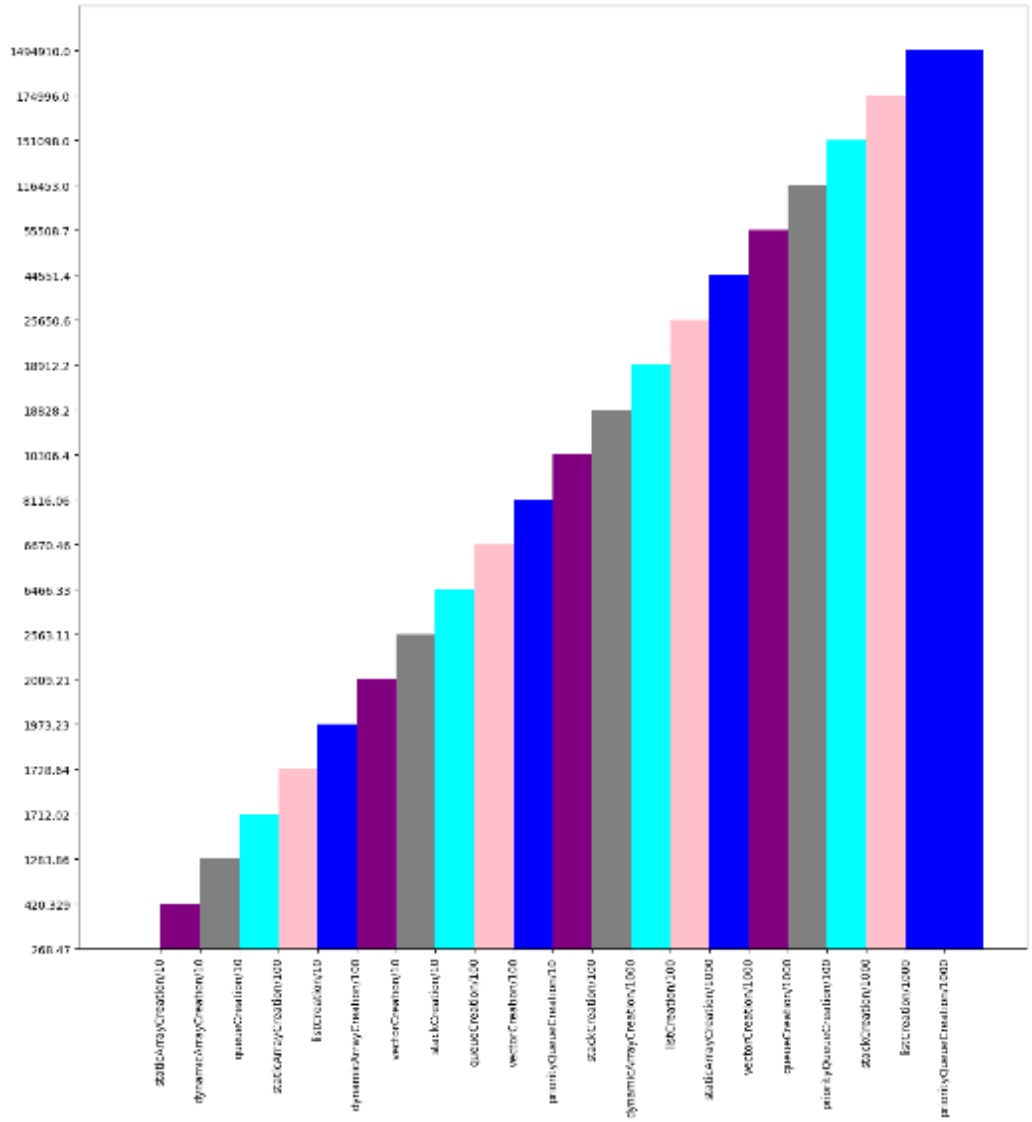
6.1.2. C++

Note: C++ tests were performed through WSL (Windows Subsystem for Linux).

- DataStructures

- Creation

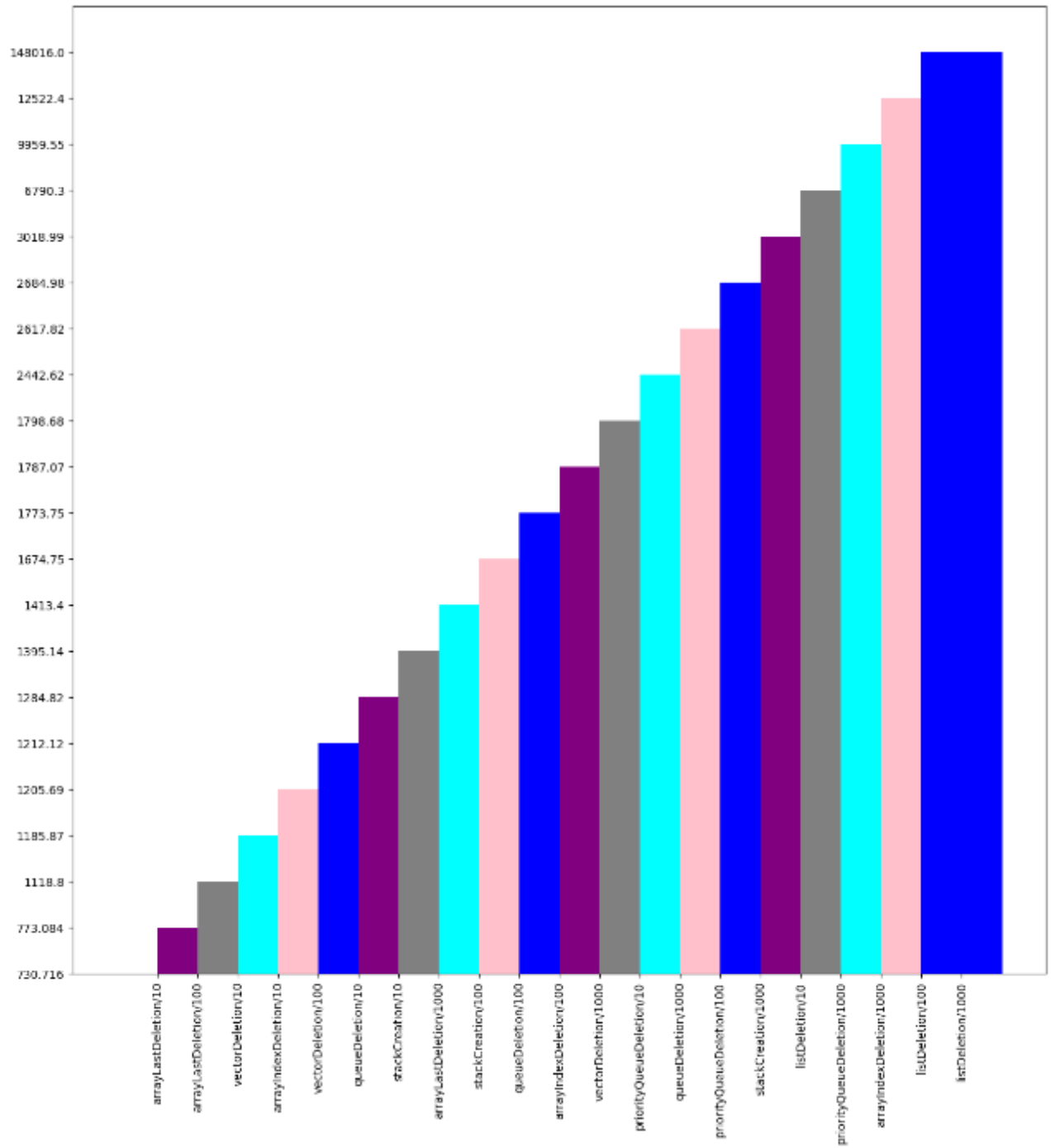
The order in which the graph grows is rather self explanatory. Since static memory allocation does not happen at runtime, it takes a significantly lower time to perform, as compared to dynamic memory allocation. Because of this, the simplest data structure (the static array) is the most efficient in terms of time, closely followed by the dynamic array. While the differences are not that noticeable for most data structures, it is essential to note the time difference between them and the priority queue, as the latter contains the ordering / priority constraint. Because of this, the time necessary to perform the creation operation is much larger.



- Deletion

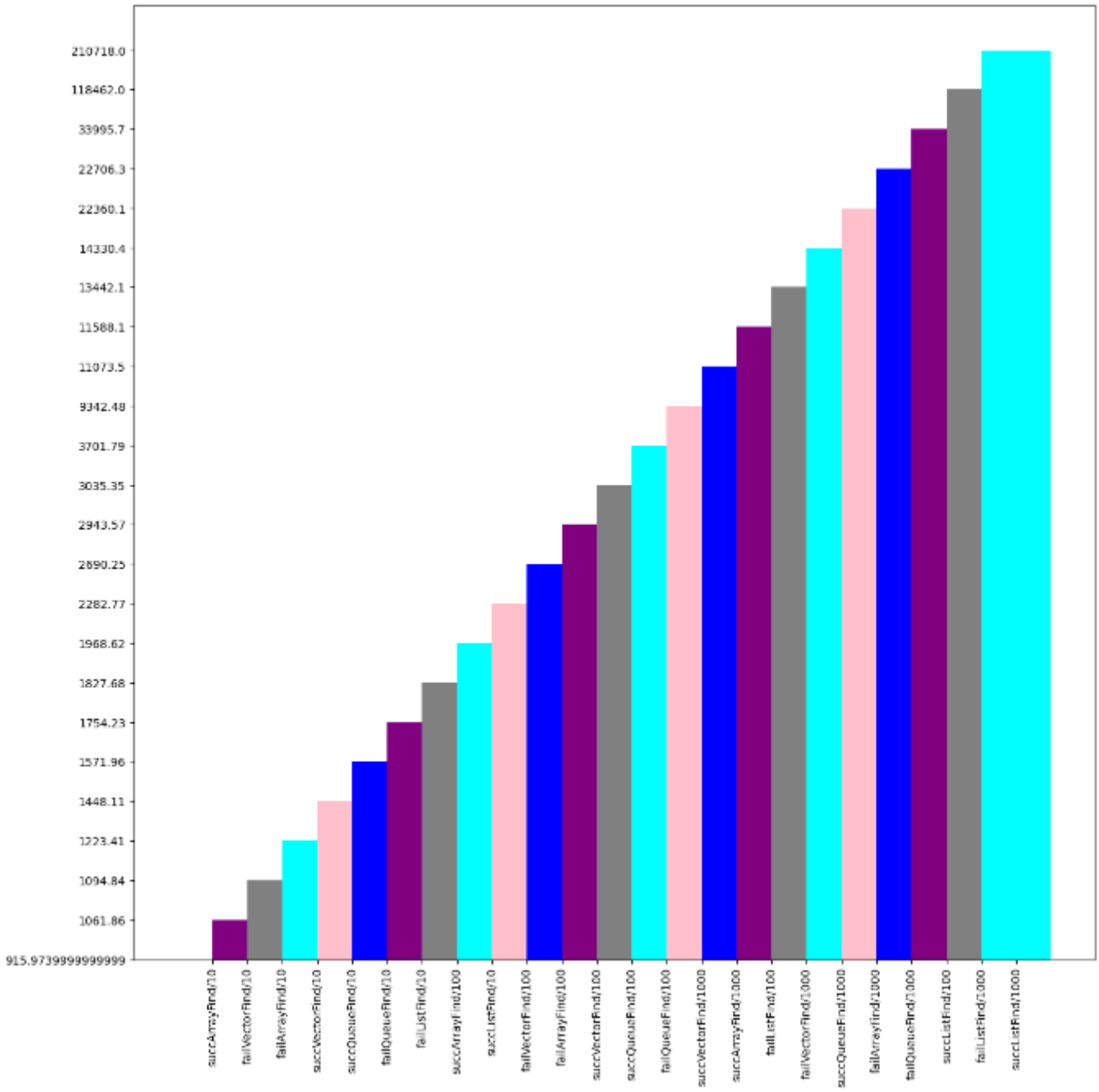
Similarly to the graph above, the deletion operation takes the shortest time for the simplest data structures, especially in terms of last index deletion. The pop operations for queue and stack also take a relatively short time, which varies very little in terms of required time as the size of the object increases. That is because the operations remain the same, despite the fact that the object at hand is larger. Lists however, and especially index array

deletion do take a long time to perform. The latter requires a shift of the entire array which might be worth it when its size is small, but it will prove to be time expensive if the size off the array is large.



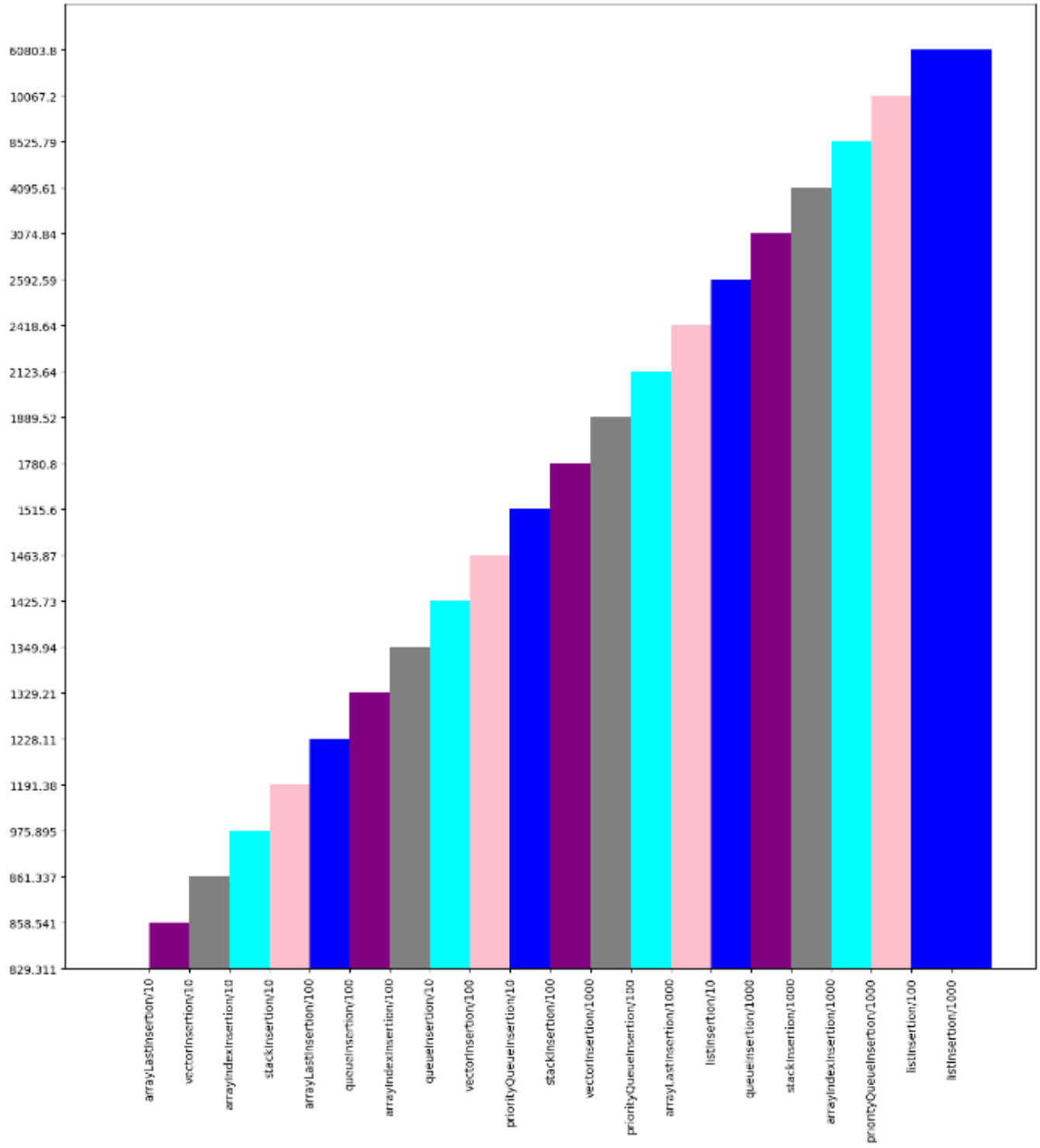
- Find

Concerning the find operation, the differences are not relevant for most tests. It is important to note that since the find operation generally requires a traversal of the data structure, as the size of the object increases, so does the time necessary to find a certain element, whether the find is successful or not.



- Insertion

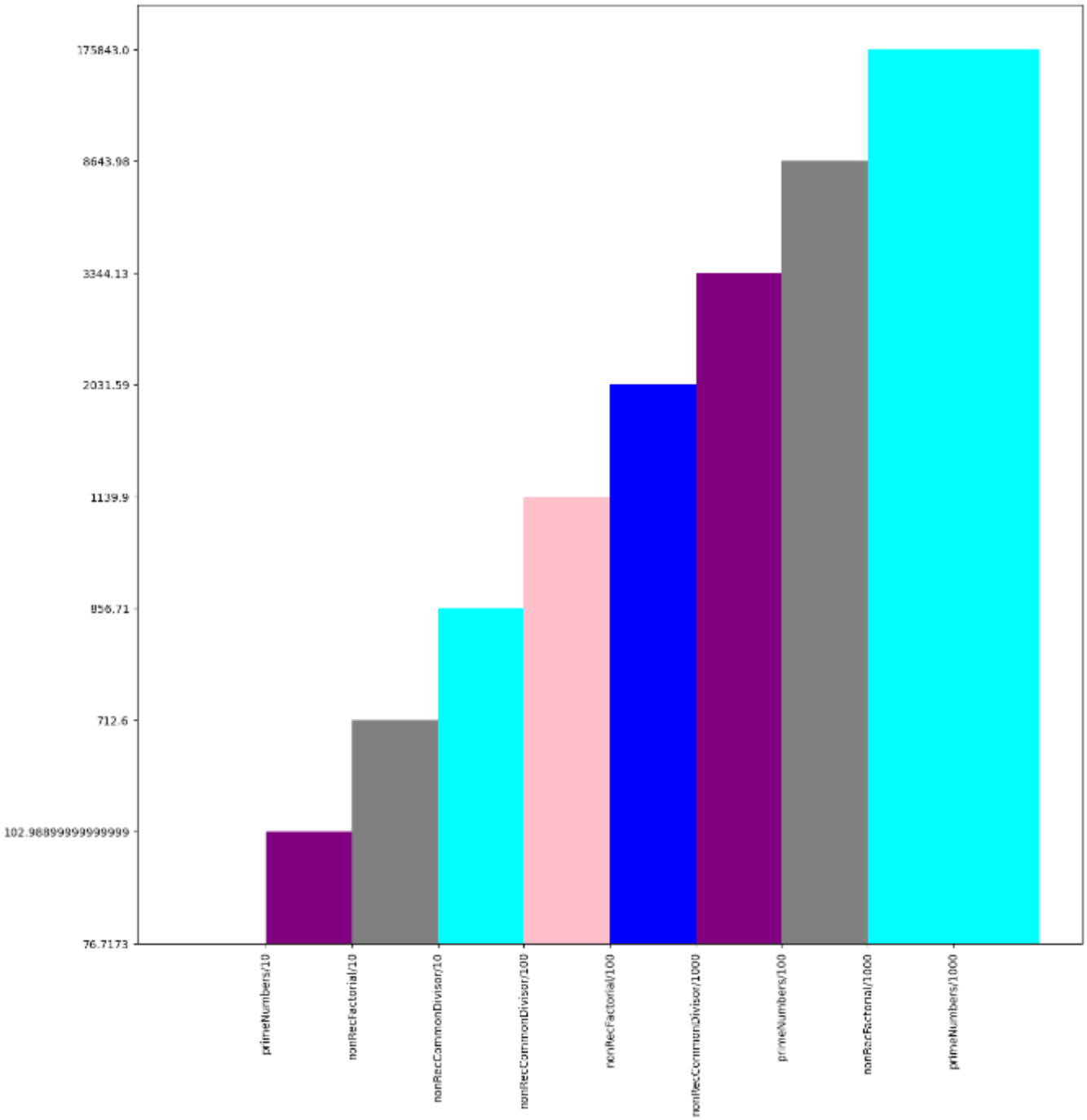
The insertion operation portrays the differences between the data structures quite well. Inserting into a pre-existing data structure is not time expensive, unless the insertion is performed at a certain index, as in the case of the array. In the other cases, the differences are not quite as visible, since the operation is essentially the same and only the total number of elements differs.



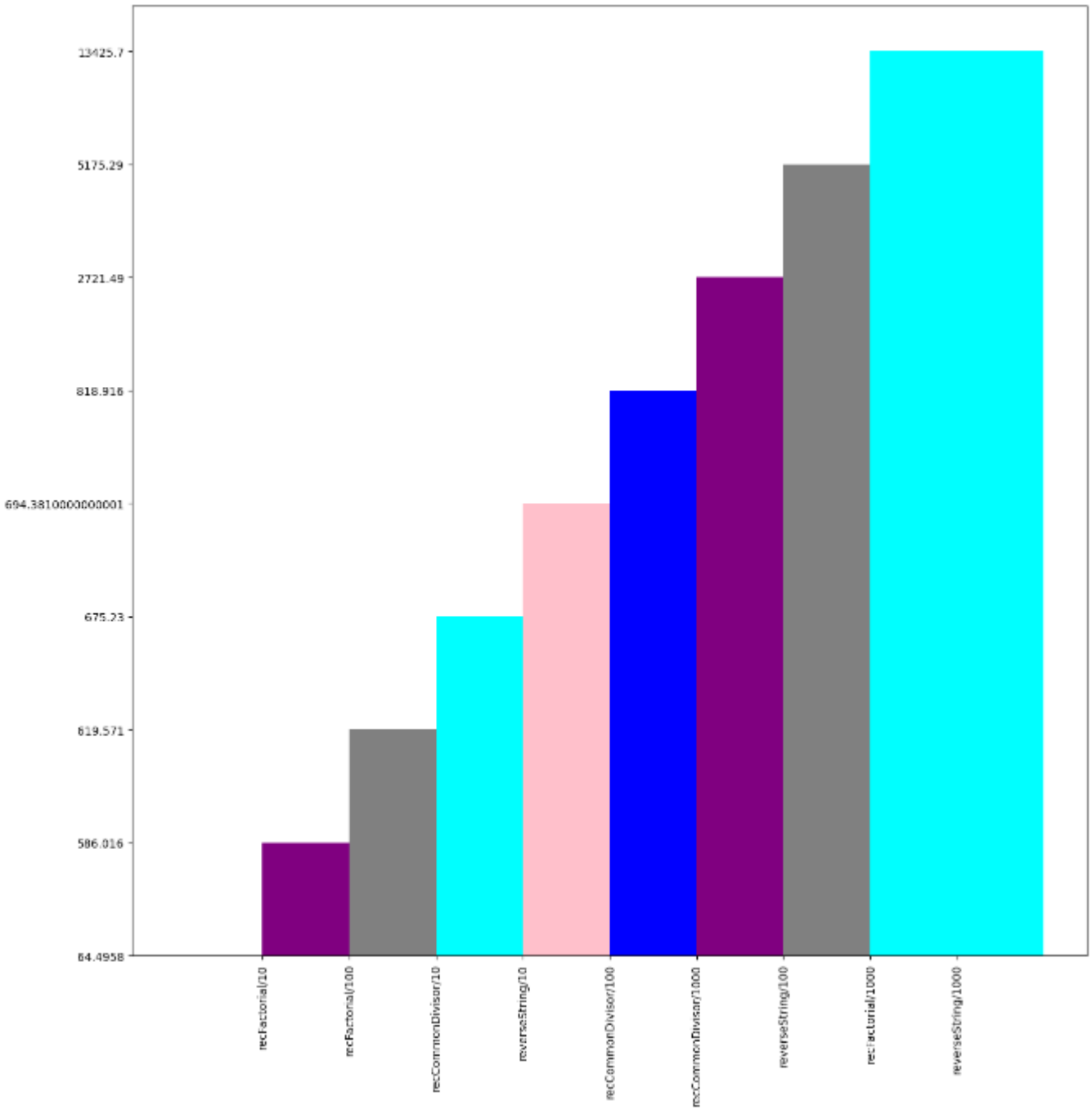
- Functions

As expected, in both cases, a larger number of elements and calls requires a larger amount of time to be performed.

- NonRecursive

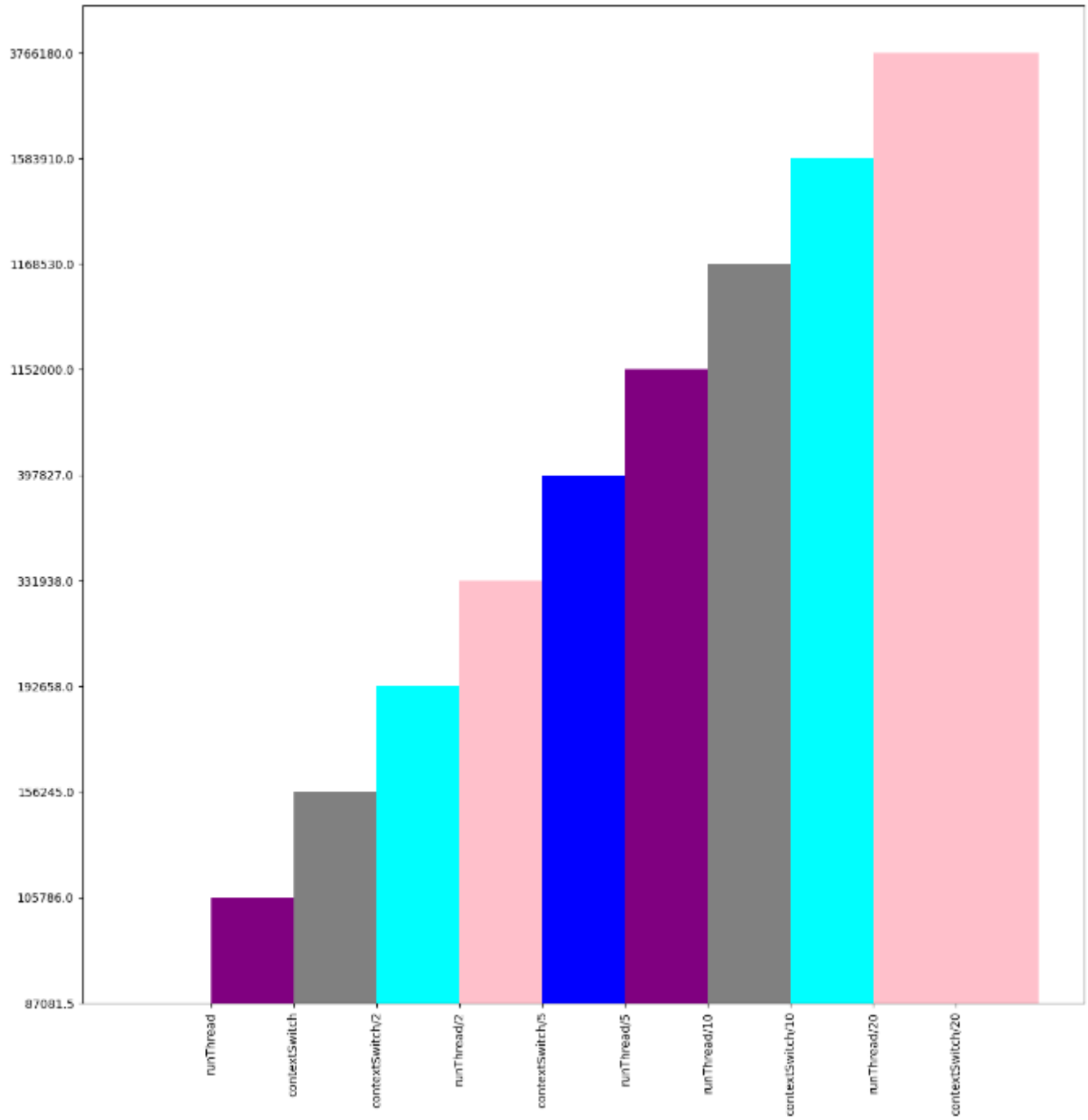


- Recursive



- Threads

In terms of thread creation alone, it is normal that a larger number of threads requires a longer time to initialize and start. Context switch, however, is relatively slower than a normal thread run, since it requires more pauses during the execution of the program.



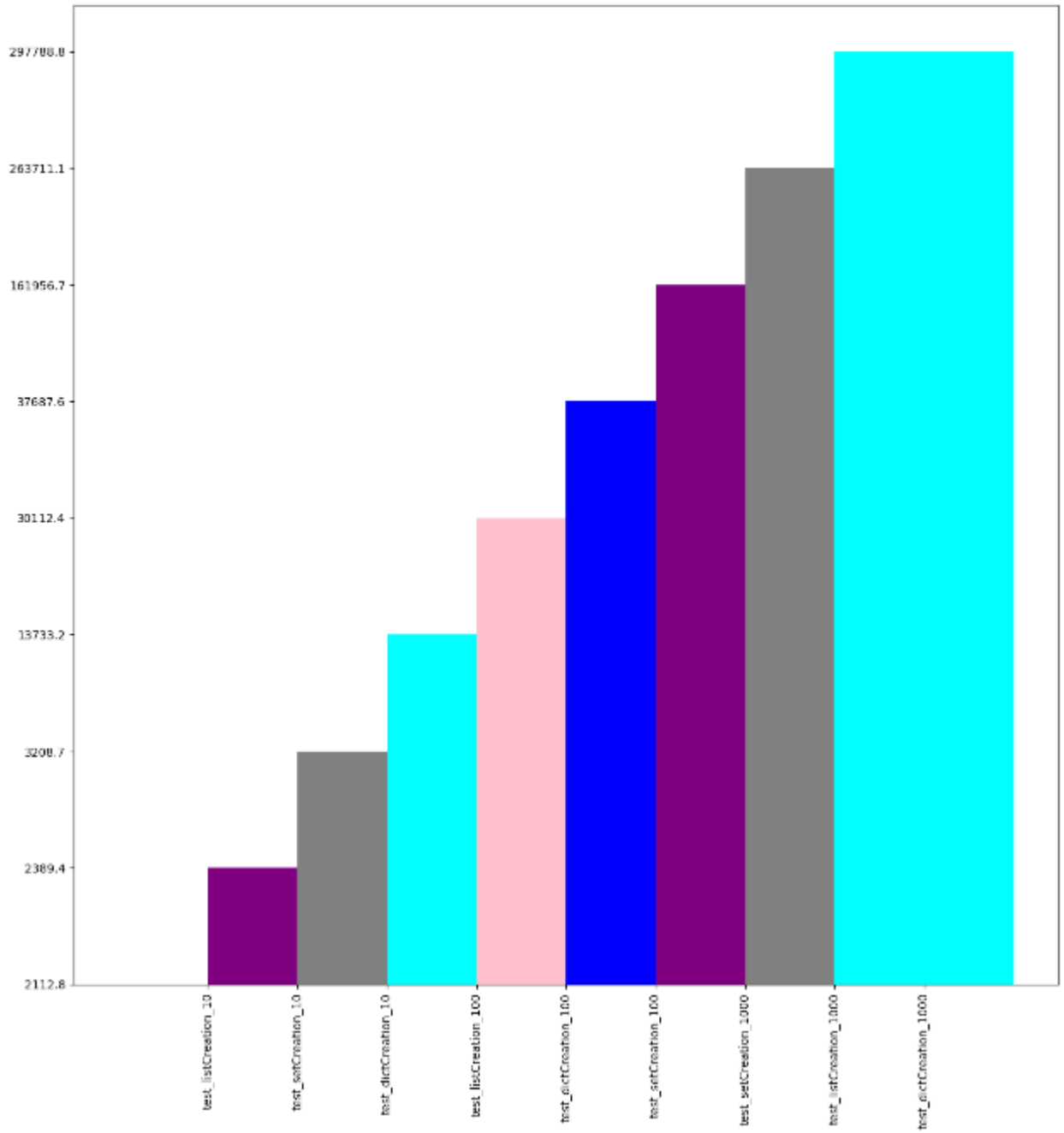
6.1.3. Python

Compared to the other programming languages, the test results in Python seem to vary more when it comes to time differences.

- DataStructures

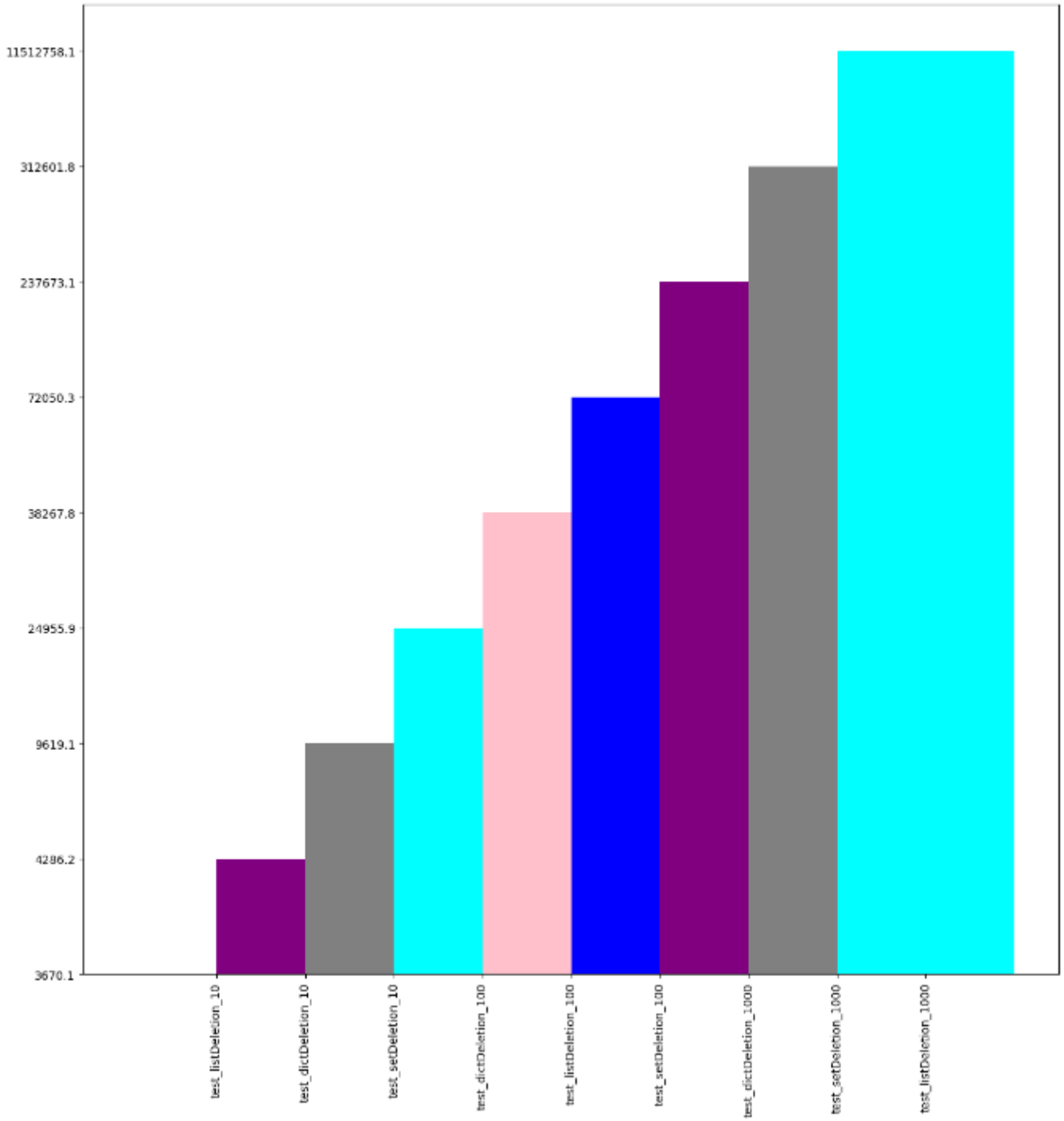
- Creation

The creation operation is quite easy to imagine. Despite the constraints and features each data structure has, the graph maintains a steady growth, based on the number of elements added into the object and the differences are rather negligible.



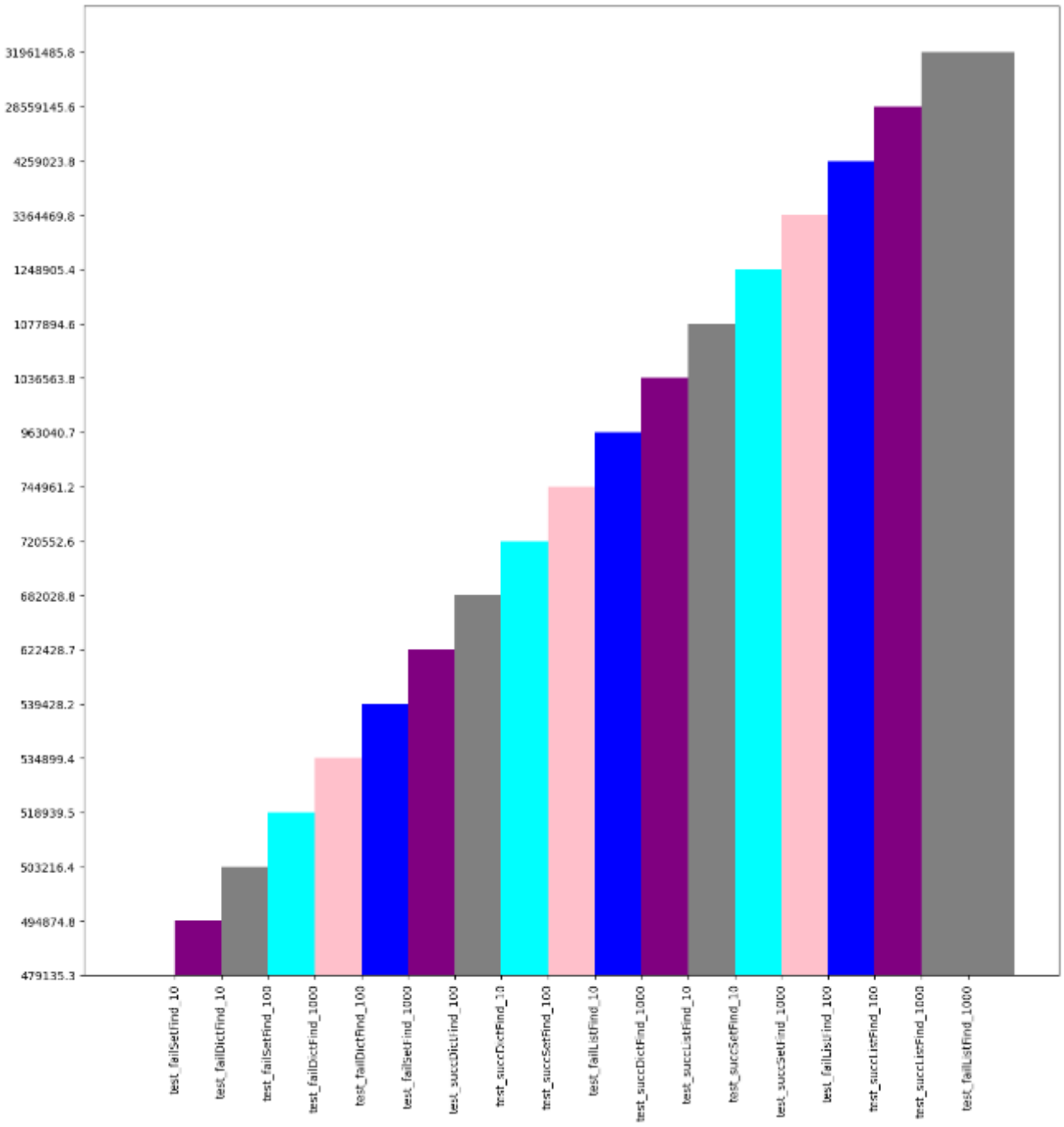
- Deletion

To be noted that the Python deletion also contains the list / set / dictionary creation.



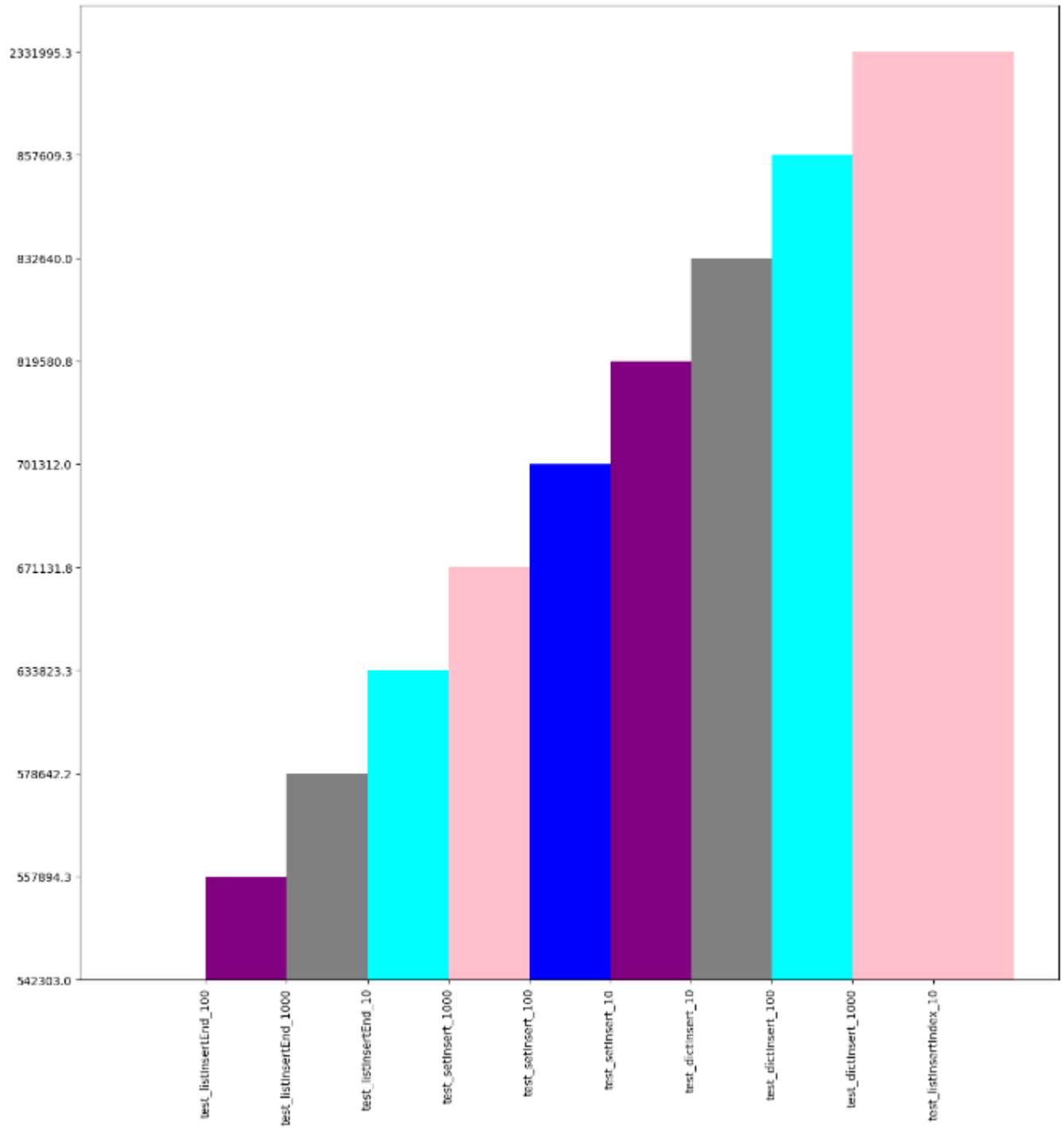
- Find

As expected, since dictionaries work similarly to hash function, they are the quickest and most efficient when it comes to finding a certain object, whether the find is successful (the object is part of the dictionary) or not. Following the same pattern, lists are the least efficient concerning the find operations since they require a full traversal of the least. Therefore, the unsuccessful find proves to be very time expensive.



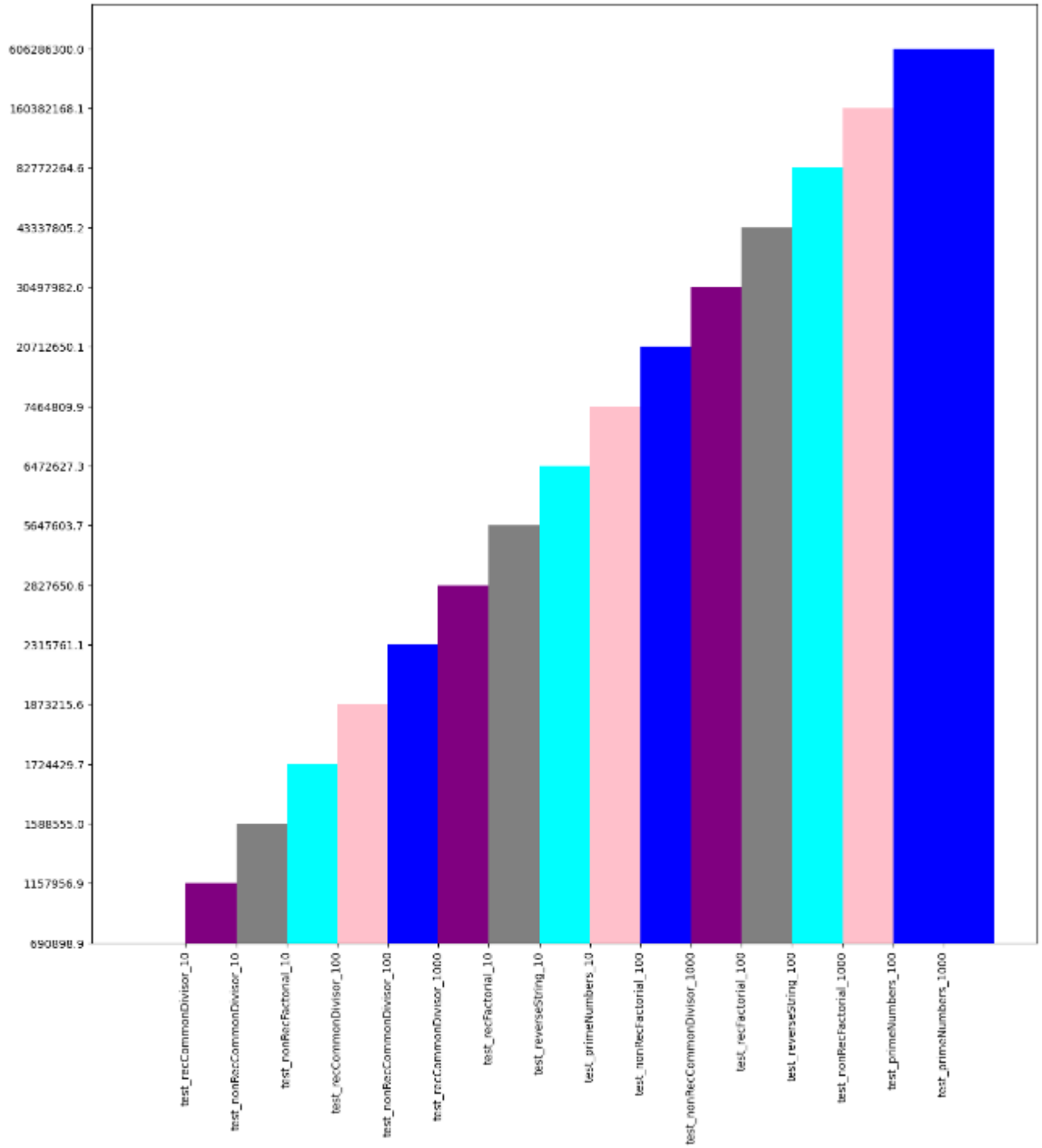
- Insertion

For the insertion operation, the most prominent difference can be seen in relation to the list object. Inserting at the end of the list proves to be especially efficient no matter the size of the list. However, inserting at a certain index requires a large amount of time since it requires a full shift of the elements. As for the other data structures, the changes are negligible in terms of the ratio between the amount of terms the object contains and the time cost of the insertion operation.

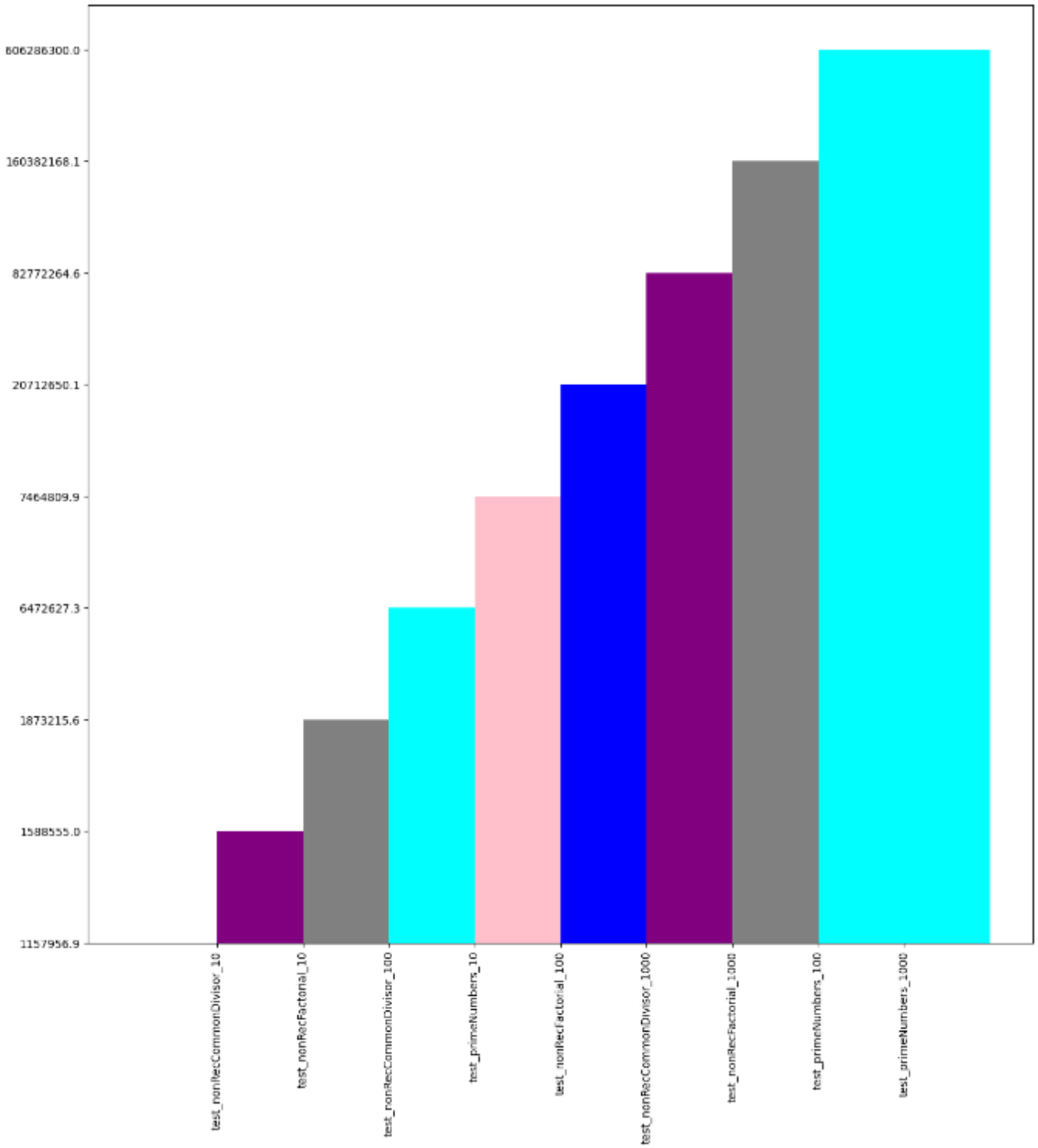


- Functions

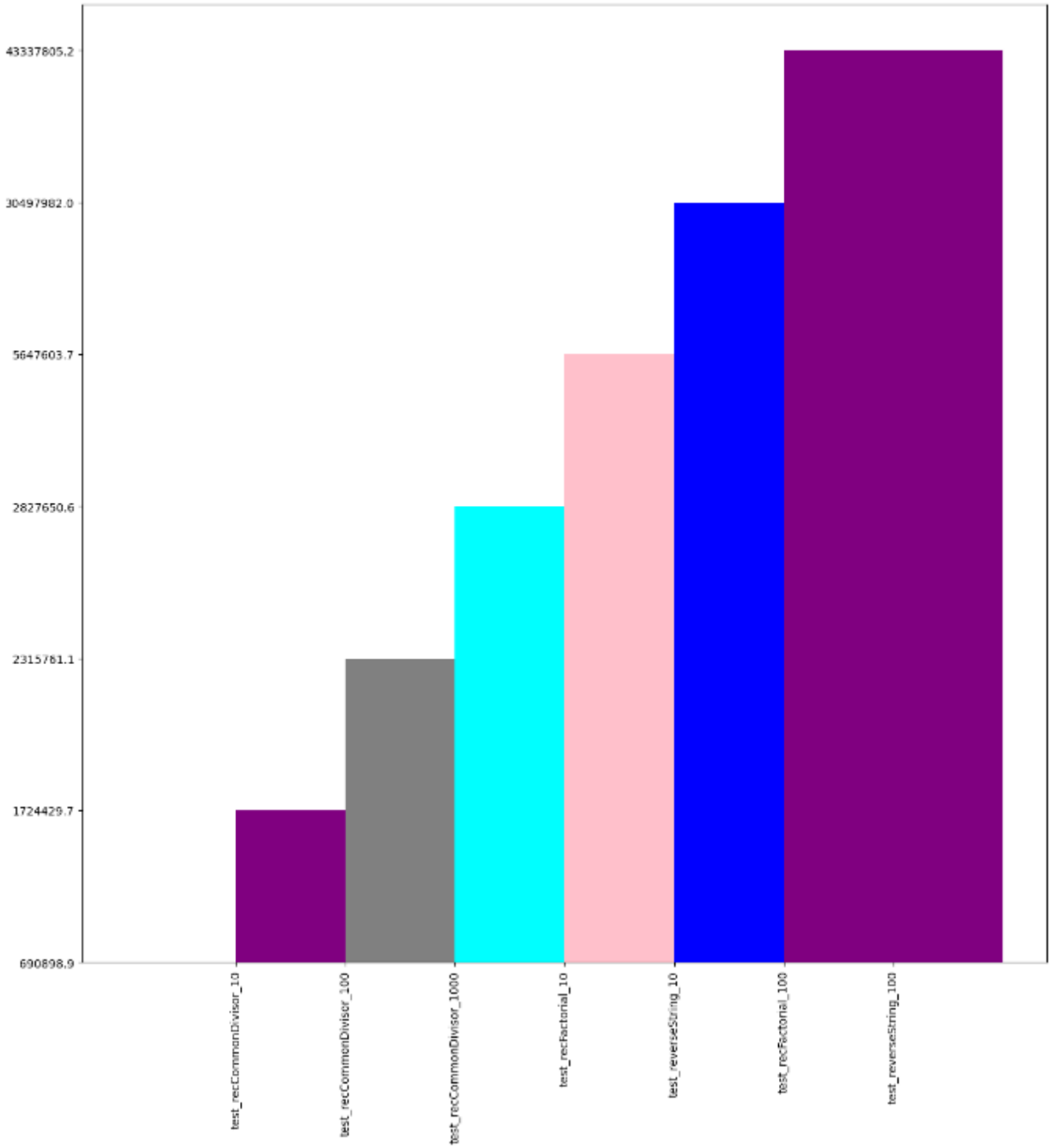
The results are the expected ones, with the recursive functions being more taxing in terms of time when compared to the non-recursive functions. It is also important to note the fact that, as well as for the other programming languages, the recursive functions do require one more function call / execution, since it makes use of utility functions in order to be performed properly.


606286300.0
160382168.1
82772264.6
43337805.2
30497982.0
20712650.1
7464809.9
6472627.3
5647603.7
2827650.6
2315761.1
1873215.6
1724429.7
1588555.0
1157956.9
690898.9
test_recCommonDivisor_10
test_nonRecCommonDivisor_10
test_nonRecFactorial_10
test_recCommonDivisor_100
test_nonRecCommonDivisor_100
test_recCommonDivisor_1000
test_recFactorial_10
test_reverseString_10
test_primeNumbers_10
test_nonRecFactorial_100
test_nonRecCommonDivisor_1000
test_recFactorial_100
test_reverseString_100
test_nonRecFactorial_1000
test_primeNumbers_100
test_primeNumbers_1000

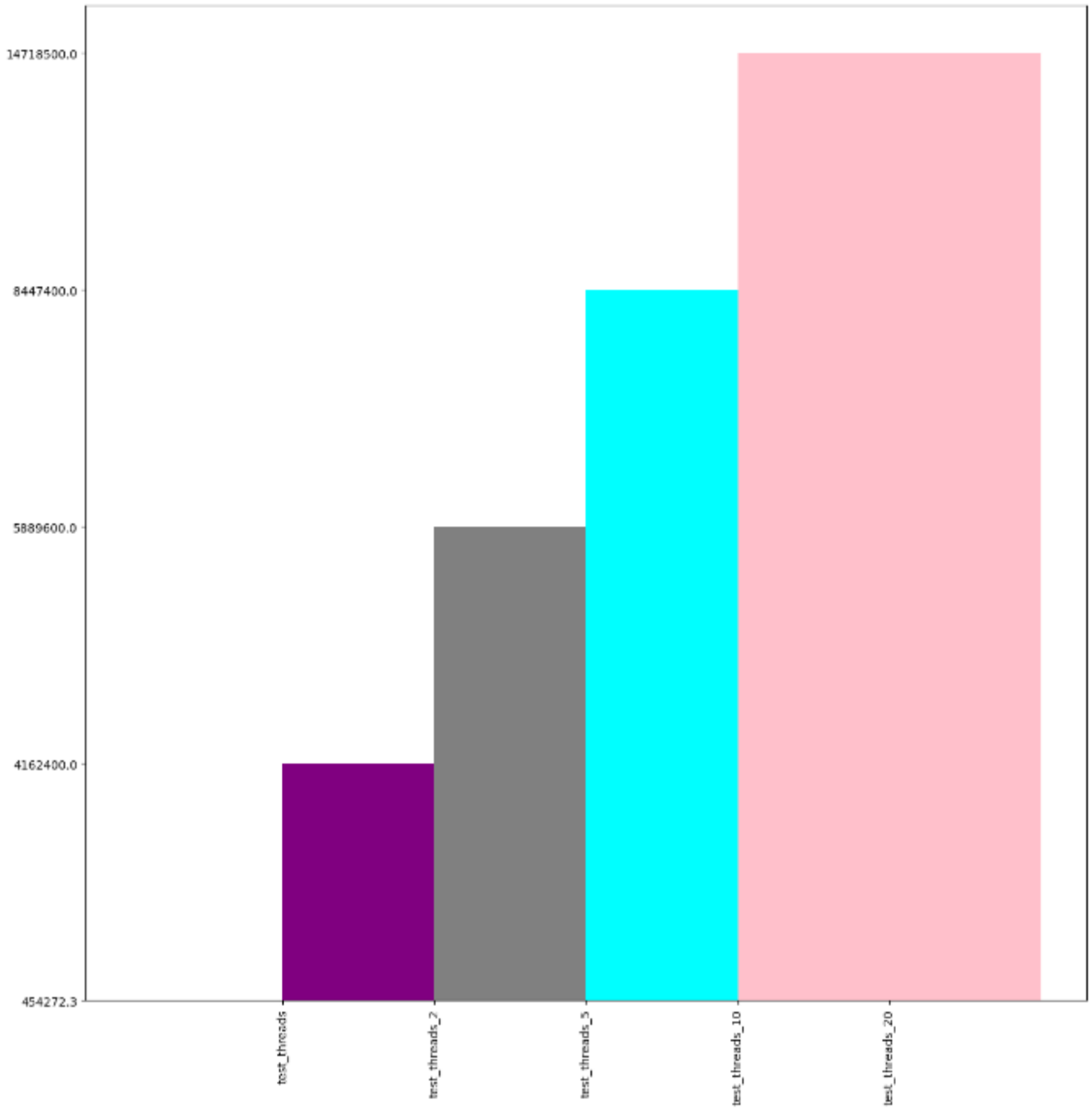
- NonRecursive



- Recursive


43337805.2
30497982.0
5647603.7
2827650.6
2315781.1
1724429.7
690898.9
test_recCommonDivisor_10
test_recCommonDivisor_100
test_recCommonDivisor_1000
test_recFactorial_10
test_reverseString_10
test_recFactorial_100
test_reverseString_100

- Threads



6.2. Comparative Results

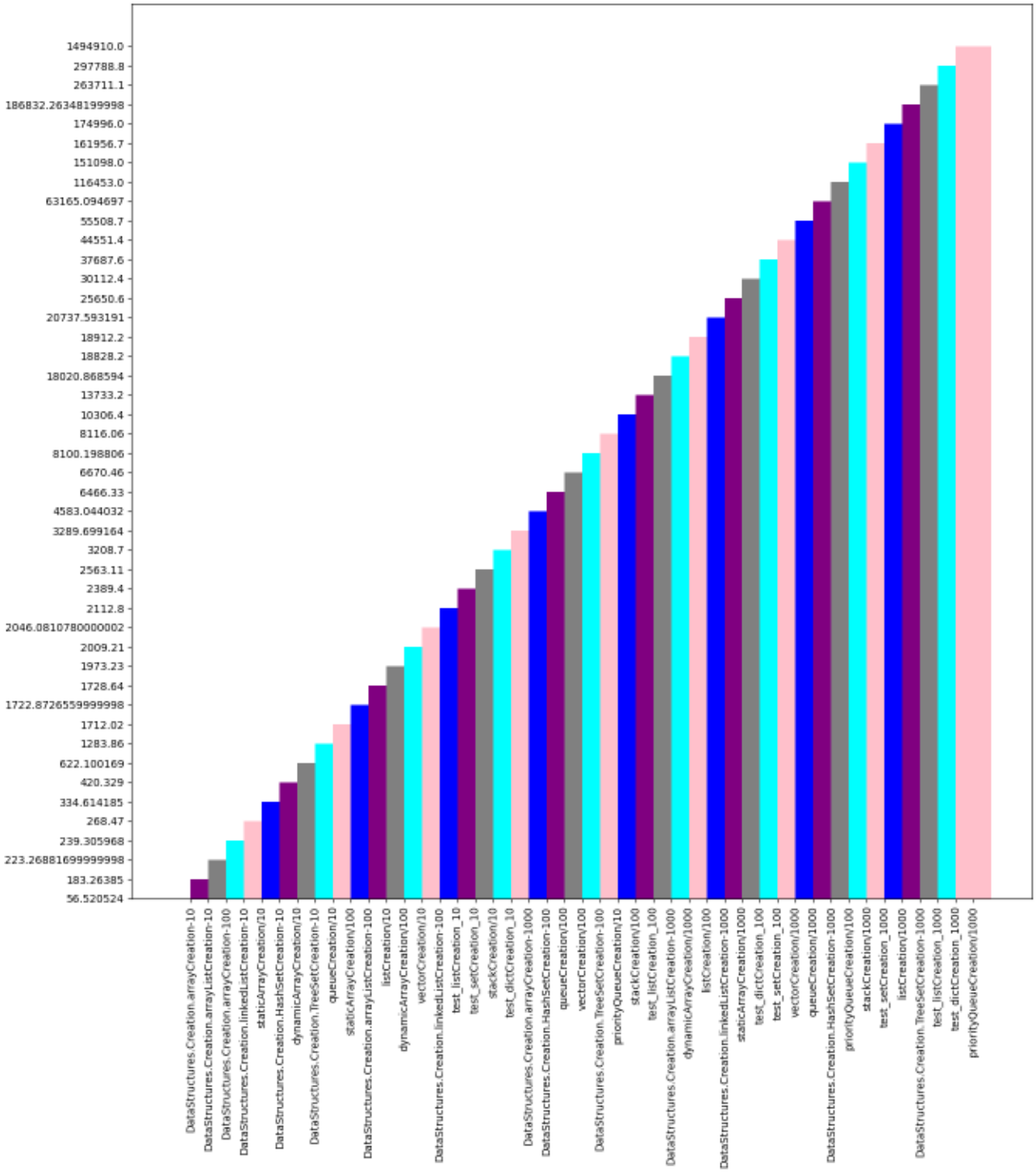
Legend:

"-" -> Java test "/" -> C++ test "_" -> Python test

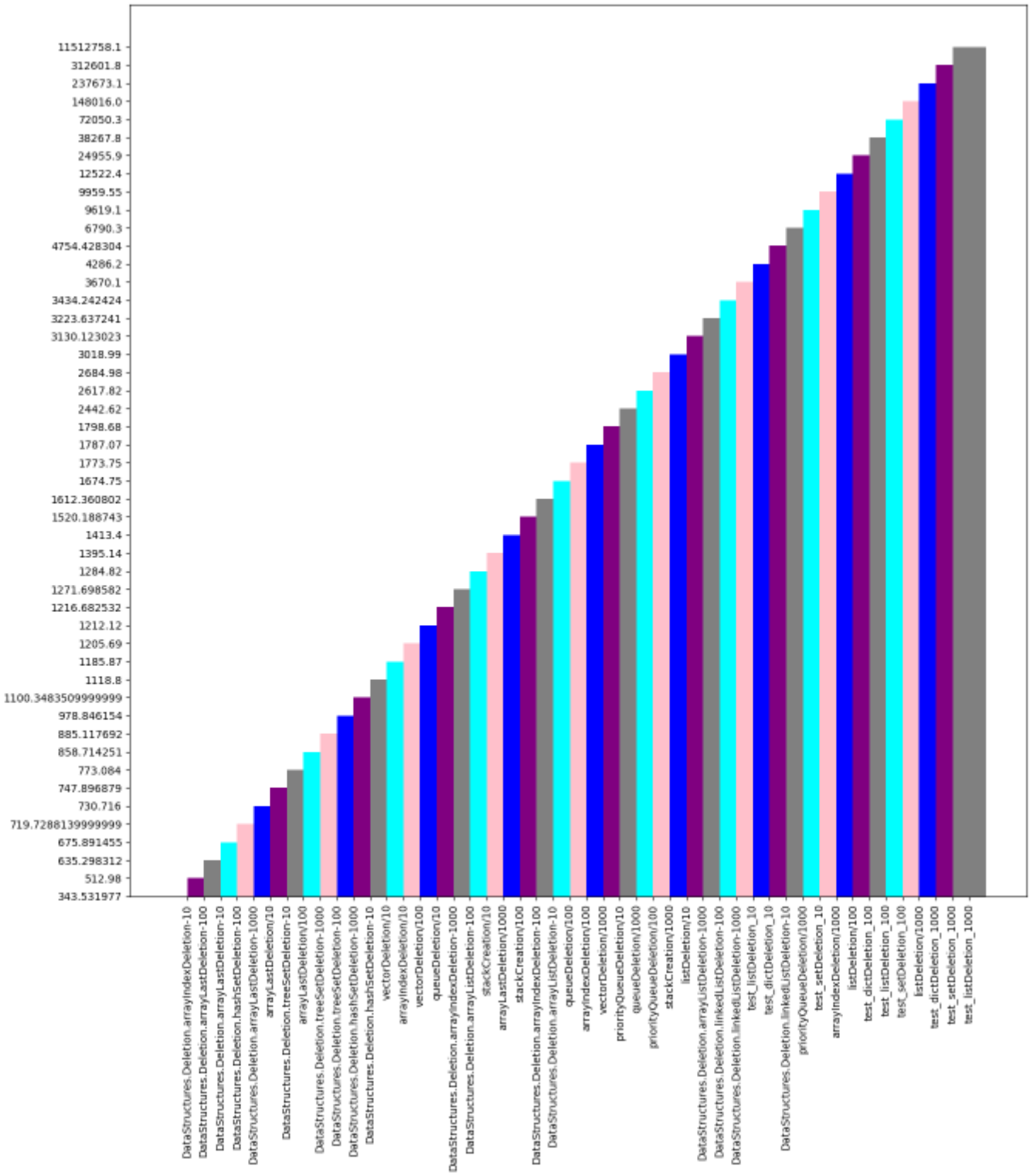
6.2.1. DataStructures

- Creation

It can be noticed that the simplest data structures, such as arrays, are at the bottom of the graph in terms of time requirements. While C++ was expected to be the fastest, especially in terms of array creation, it appears that Java generated better results. That may have to do with the fact that the Java Machine optimized t he code despite the attempts to avoid optimization.

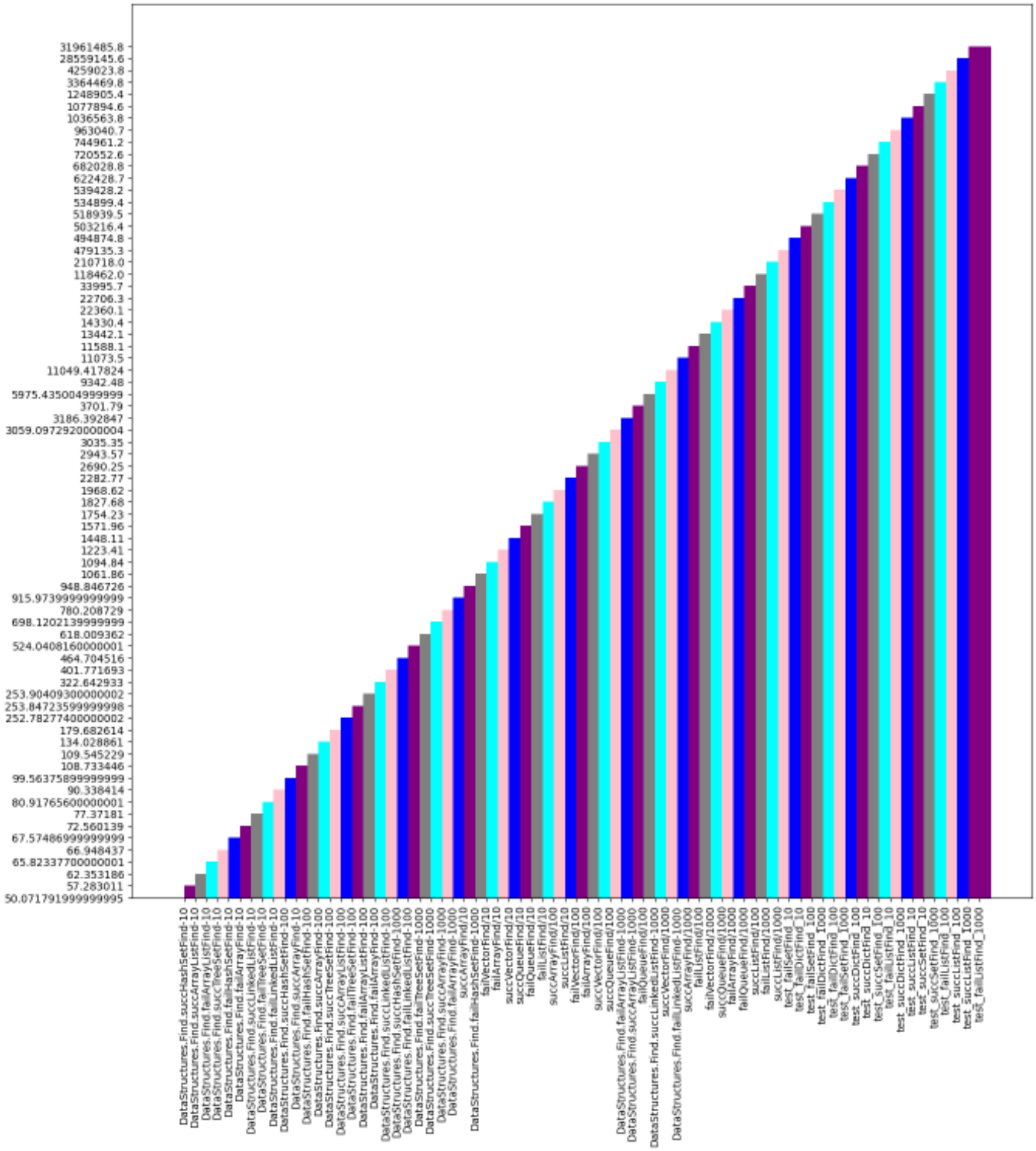


- Deletion

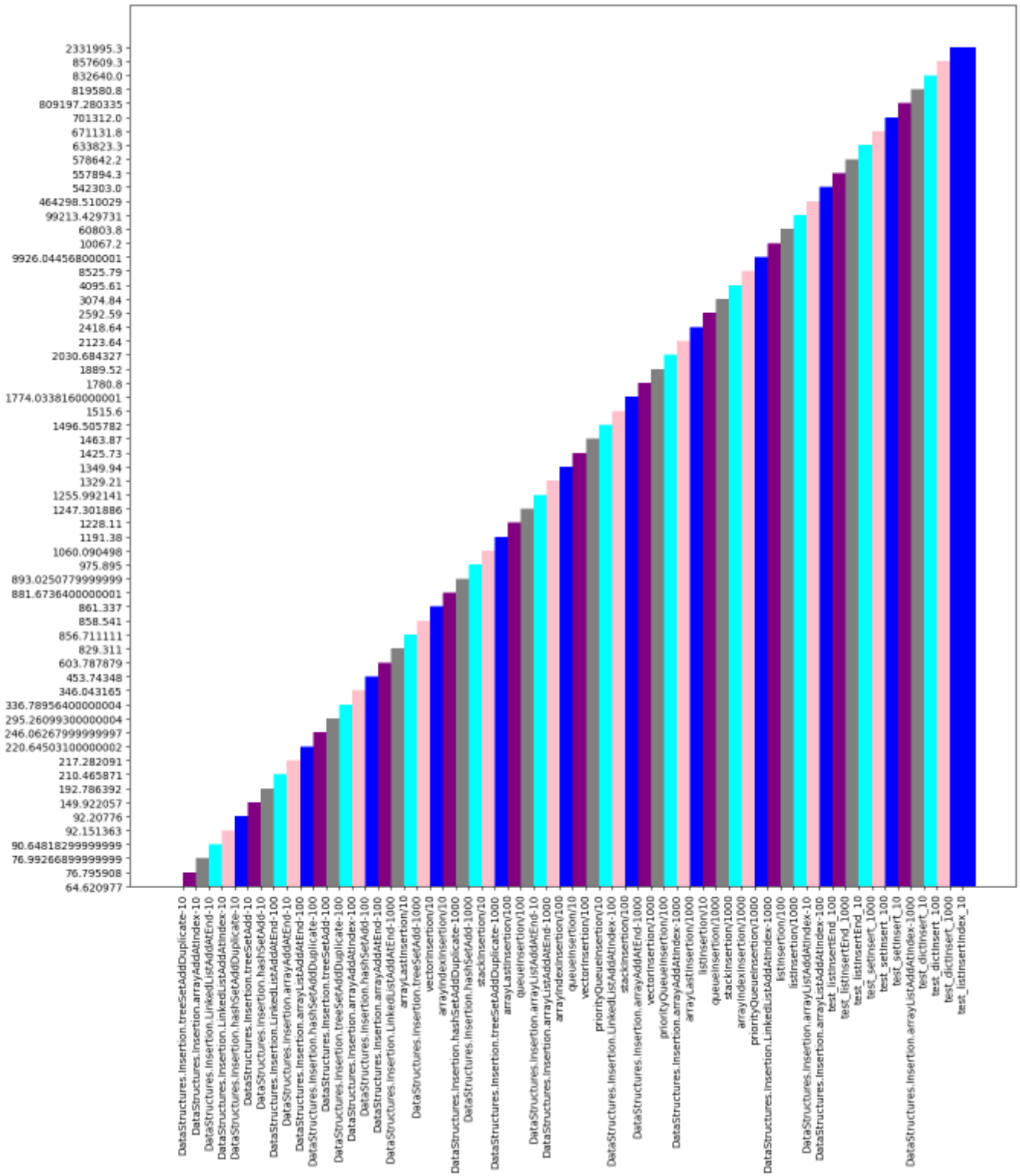

11512758.1
312601.8
237673.1
148016.0
72050.3
38267.8
24955.9
12522.4
9959.55
9619.1
6790.3
4754.428304
4286.2
3670.1
3434.242424
3223.637241
3130.123023
3018.99
2684.98
2617.82
2442.62
1798.68
1787.07
1773.75
1674.75
1612.360802
1520.188743
1413.4
1395.14
1284.82
1271.698582
1216.682532
1212.12
1205.69
1185.87
1118.8
1100.3483509999999
978.846154
885.117692
858.714251
773.084
747.896879
730.716
719.7288139999999
675.891455
635.298312
512.98
343.531977
DataStructures.Deletion.arrayIndexDeletion-10
DataStructures.Deletion.arrayLastDeletion-100
DataStructures.Deletion.arrayLastDeletion-10
DataStructures.Deletion.hashSetDeletion-100
DataStructures.Deletion.arrayLastDeletion/10
DataStructures.Deletion.arrayLastDeletion-1000
DataStructures.Deletion.treeSetDeletion-10
DataStructures.Deletion.arrayLastDeletion/100
DataStructures.Deletion.treeSetDeletion-1000
DataStructures.Deletion.hashSetDeletion-100
DataStructures.Deletion.hashSetDeletion-1000
DataStructures.Deletion.hashSetDeletion-10
vectorDeletion/10
arrayIndexDeletion/10
vectorDeletion/100
queueDeletion/10
DataStructures.Deletion.arrayIndexDeletion-1000
DataStructures.Deletion.arrayListDeletion-100
stackCreation/10
arrayLastDeletion/1000
stackCreation/100
DataStructures.Deletion.arrayIndexDeletion-100
DataStructures.Deletion.arrayListDeletion-10
queueDeletion/100
arrayIndexDeletion/100
vectorDeletion/1000
priorityQueueDeletion/10
queueDeletion/1000
priorityQueueDeletion/100
stackCreation/1000
listDeletion/10
DataStructures.Deletion.arrayListDeletion-1000
DataStructures.Deletion.linkedListDeletion-100
DataStructures.Deletion.linkedListDeletion-1000
test_listDeletion_10
test_dictDeletion_10
DataStructures.Deletion.linkedListDeletion-10
priorityQueueDeletion/1000
test_setDeletion_10
arrayIndexDeletion/1000
listDeletion/100
test_dictDeletion_100
test_listDeletion_100
test_setDeletion_100
listDeletion/1000
test_dictDeletion_1000
test_setDeletion_1000
test_listDeletion_1000

- Find

In terms of the find operations, data structures like those from java, which have a clear distribution of the elements, are more efficient in terms of this.

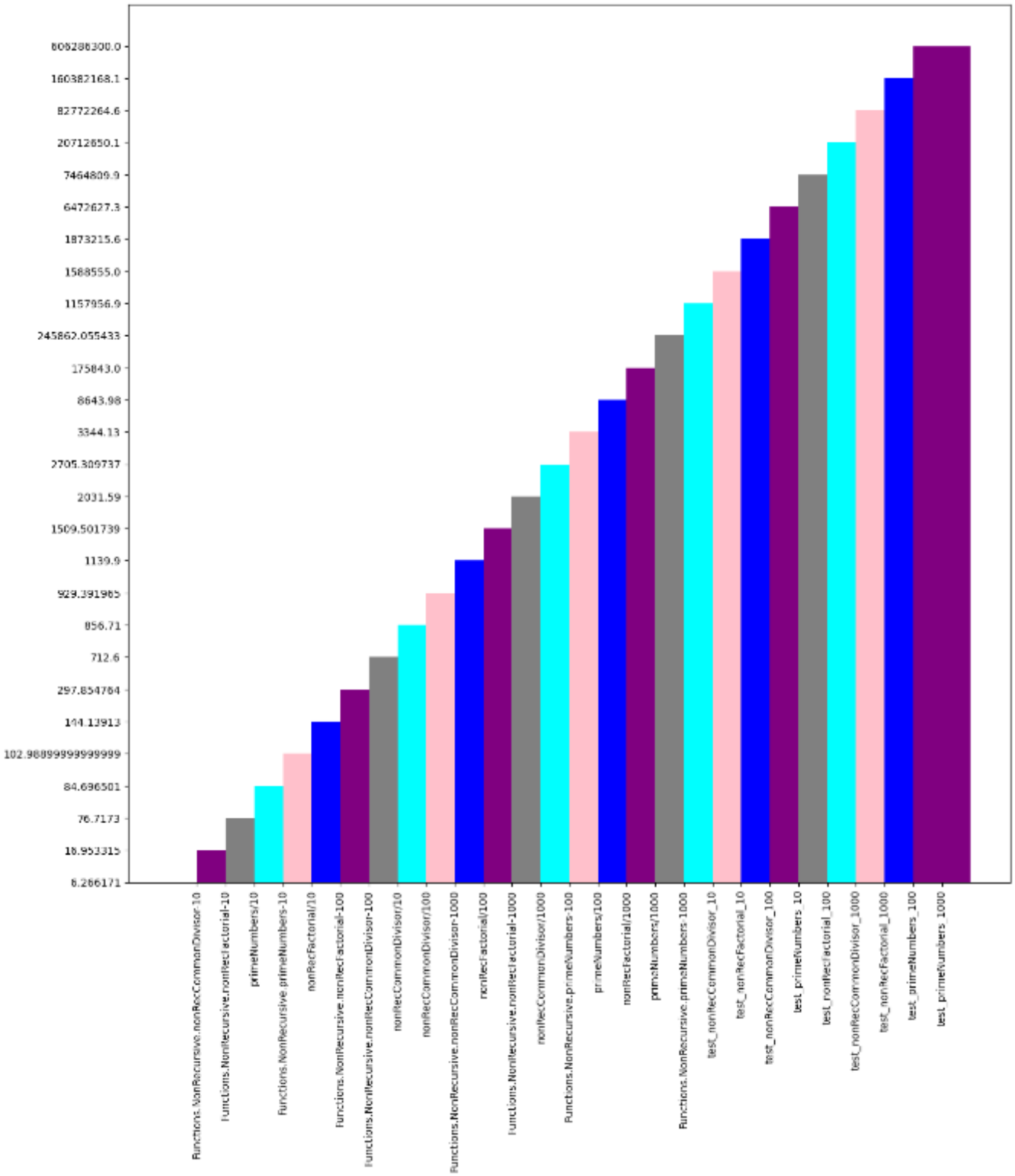


- Insertion

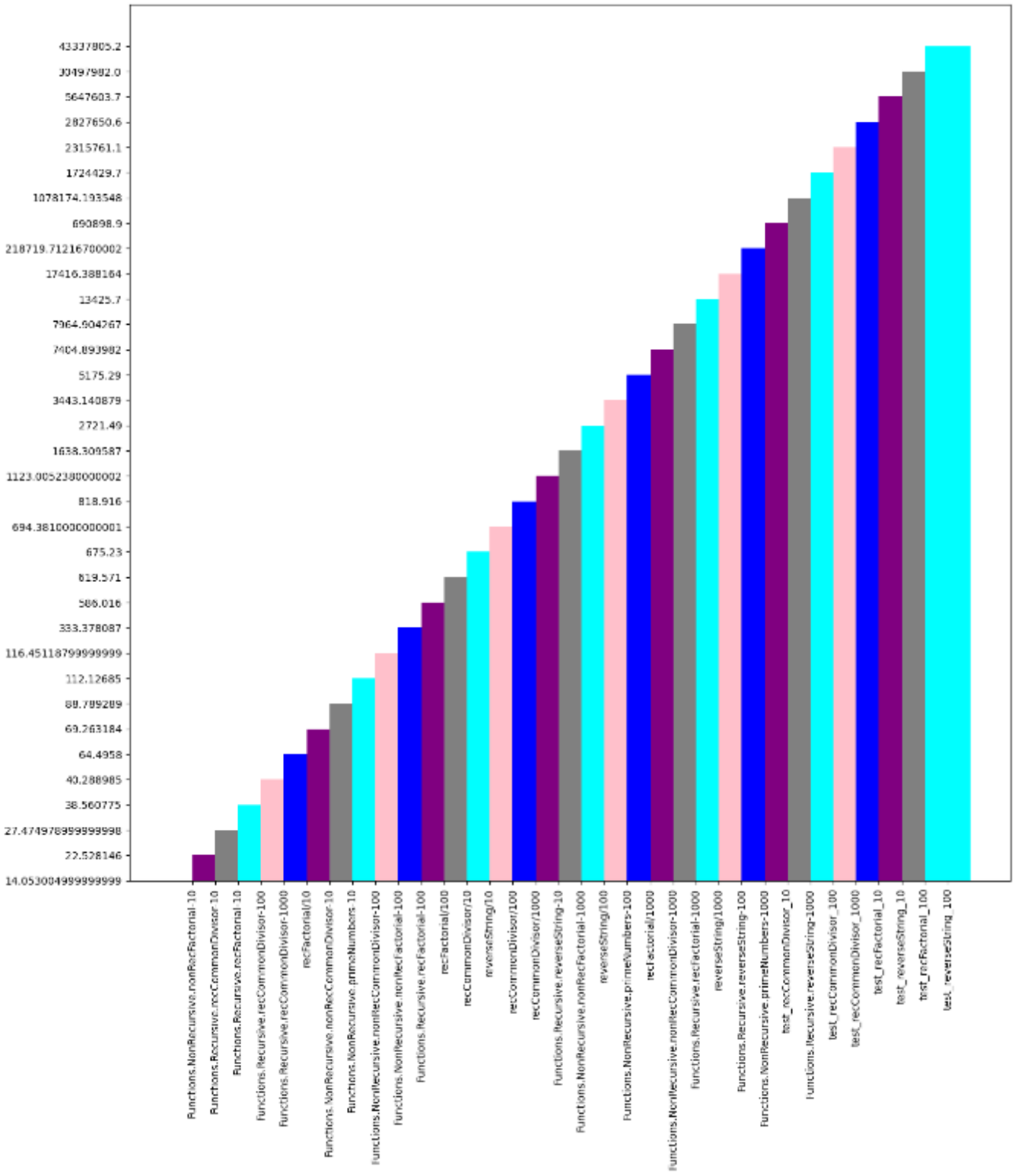

2331995.3
857609.3
832640.0
819580.8
809197.280335
701312.0
671131.8
633823.3
578642.2
557894.3
542303.0
464298.510029
99213.429731
60803.8
10067.2
9926.044568000001
8525.79
4095.61
3074.84
2592.59
2418.64
2123.64
2030.684327
1889.52
1780.8
1774.0338160000001
1515.6
1496.505782
1463.87
1425.73
1349.94
1329.21
1255.992141
1247.301886
1228.11
1191.38
1060.090498
975.895
893.0250779999999
881.6736400000001
861.337
858.541
856.711111
829.311
603.787879
453.74348
346.043165
336.78956400000004
295.26099300000004
246.06267999999997
220.64503100000002
217.282091
210.465871
192.786392
149.922057
92.20776
92.151363
90.64818299999999
76.99266899999999
76.795908
64.620977
DataStructures.Insertion.treeSetAddDuplicate-10
DataStructures.Insertion.arrayAddAtIndex-10
DataStructures.Insertion.LinkedListAddAtEnd-10
DataStructures.Insertion.LinkedListAddAtIndex-10
DataStructures.Insertion.hashSetAddDuplicate-10
DataStructures.Insertion.treeSetAdd-10
DataStructures.Insertion.hashSetAdd-10
DataStructures.Insertion.LinkedListAddAtEnd-100
DataStructures.Insertion.arrayAddAtEnd-10
DataStructures.Insertion.arrayListAddAtEnd-100
DataStructures.Insertion.hashSetAddDuplicate-100
DataStructures.Insertion.treeSetAdd-100
DataStructures.Insertion.treeSetAddDuplicate-100
DataStructures.Insertion.arrayAddAtIndex-100
DataStructures.Insertion.hashSetAdd-100
DataStructures.Insertion.arrayAddAtEnd-100
DataStructures.Insertion.LinkedListAddAtEnd-1000
arrayLastInsertion/10
DataStructures.Insertion.treeSetAdd-1000
vectorInsertion/10
arrayIndexInsertion/10
DataStructures.Insertion.hashSetAddDuplicate-1000
DataStructures.Insertion.hashSetAdd-1000
stackInsertion/10
DataStructures.Insertion.treeSetAddDuplicate-1000
arrayLastInsertion/100
queueInsertion/100
DataStructures.Insertion.arrayListAddAtEnd-10
DataStructures.Insertion.arrayListAddAtEnd-1000
arrayIndexInsertion/100
queueInsertion/10
vectorInsertion/100
priorityQueueInsertion/10
DataStructures.Insertion.LinkedListAddAtIndex-100
stackInsertion/100
DataStructures.Insertion.arrayAddAtEnd-1000
vectorInsertion/1000
priorityQueueInsertion/100
DataStructures.Insertion.arrayAddAtIndex-1000
arrayLastInsertion/1000
listInsertion/10
queueInsertion/1000
stackInsertion/1000
arrayIndexInsertion/1000
priorityQueueInsertion/1000
DataStructures.Insertion.LinkedListAddAtIndex-1000
listInsertion/100
listInsertion/1000
DataStructures.Insertion.arrayListAddAtIndex-10
DataStructures.Insertion.arrayListAddAtIndex-100
test_listInsertEnd_100
test_listInsertEnd_1000
test_listInsertEnd_10
test_setInsert_1000
test_setInsert_100
test_setInsert_10
DataStructures.Insertion.arrayListAddAtIndex-1000
test_dictInsert_10
test_dictInsert_100
test_dictInsert_1000
test_listInsertIndex_10

6.2.2. Functions

- NonRecursive

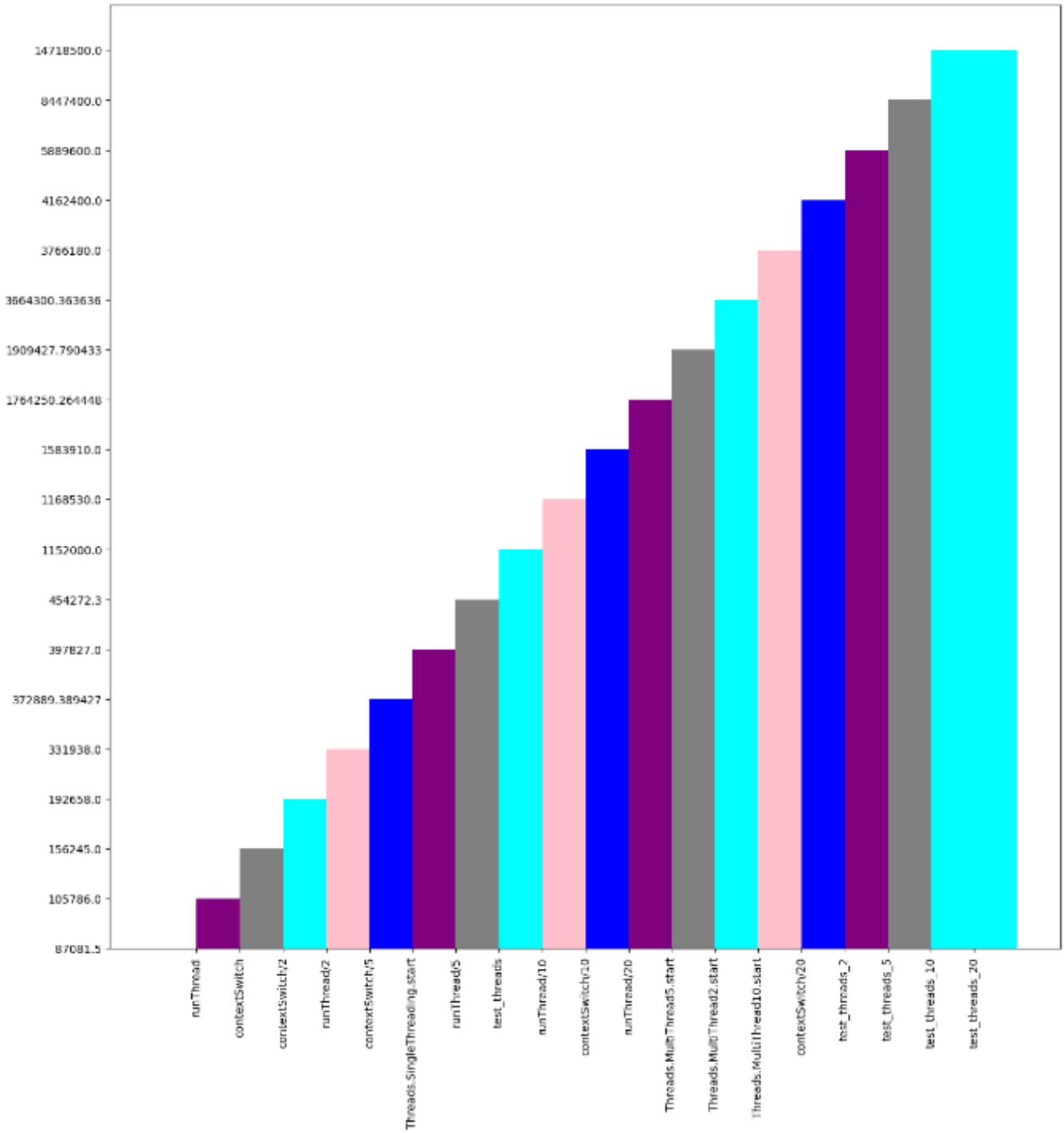


- Recursive


43337805.2
30497982.0
5647603.7
2827650.6
2315761.1
1724429.7
1078174.193548
690898.9
218719.71216700002
17416.388164
13425.7
7964.904267
7404.893982
5175.29
3443.140879
2721.49
1638.309587
1123.0052380000002
818.916
694.3810030000001
675.23
619.571
586.016
333.378087
116.45118799999999
112.12685
89.789289
69.263184
64.4958
40.288985
38.560775
27.474978999999998
22.528146
14.053004999999999
Functions.NonRecursive.nonRecFactorial_10
Functions.Recursive.recCommanDivisor_10
Functions.Recursive.recFactorial_10
Functions.Recursive.recCommonDivisor_100
Functions.Recursive.recCommonDivisor_1000
recFactorial_10
Functions.NonRecursive.nonRecCommanDivisor_10
Functions.NonRecursive.primeNumbers_10
Functions.NonRecursive.nonRecCommonDivisor_100
Functions.NonRecursive.nonRecFactorial_100
Functions.Recursive.recFactorial_100
recFactorial_100
recCommonDivisor_10
reverseString_10
recCommonDivisor_100
recCommonDivisor_1000
Functions.Recursive.reverseString_10
Functions.NonRecursive.nonRecFactorial_1000
reverseString_100
Functions.NonRecursive.primeNumbers_100
recFactorial_1000
Functions.NonRecursive.nonRecCommonDivisor_1000
Functions.Recursive.recFactorial_1000
reverseString_1000
Functions.Recursive.reverseString_100
Functions.NonRecursive.primeNumbers_1000
test_recCommanDivisor_10
Functions.Recursive.reverseString_1000
test_recCommonDivisor_100
test_recCommonDivisor_1000
test_recFactorial_10
test_reverseString_10
test_recFactorial_100
test_reverseString_100

6.2.3. Threads

C++ was expected to have the lowest execution times, since it is the lowest level language of the three of them and the thread control is very detailed. In terms of python multithreading, it can be seen that, as the number of threads increases, the required time does as well. That is because the Threadpool library is simply a way to attempt to simulate multithreading, but Python, as it is, does not allow it.



7. Conclusions

In conclusion, the generated tests gave, for the most part, the expected results. That is, that C++, being the most low level language of the three, tends to either be the most efficient or have a similar efficiency to Java. There is also the option of Java optimization, which can happen despite attempting to avoid it, which can push the Java tests to have a better result. The errors, are however, not that visible and are negligible.

Despite the errors, Python had the worst scores in terms of time, which was to be expected since the operations happen at runtime and the language encapsulates the C compiler.

As further development, it would be possible to create an interactive application for the project, in order to be able to view the graphs all at once.

8. Bibliography

1. Benchmark (computing). [Online] https://en.wikipedia.org/wiki/Benchmark_(computing).

2. Memory Management - Python documentation. [Online] https://docs.python.org/3/c-api/memory.html.

3. Memory Management in Python. [Online] https://towardsdatascience.com/memory-management-in-python-6bea0c8aecc9.

4. Memory Management in Java. [Online] https://www.javatpoint.com/memory-management-in-java.

5. Understanding Memory Management. [Online] https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html.

6. Dynamic Memory Allocation in C++. [Online] https://www.studytonight.com/cpp/memory-management-in-cpp.php.

7. Thread (computing). [Online] https://en.wikipedia.org/wiki/Thread_(computing).

8. JMH - Java Microbenchmark Harness. [Online] http://tutorials.jenkov.com/java-performance/jmh.html#dead-code-elimination.

9. Microbenchmarking with Java. [Online] https://www.baeldung.com/java-microbenchmark-harness.

10. timeit - Measure execution time of small code snippets. [Online] https://docs.python.org/3/library/timeit.html.

11. The Python Profilers. [Online] https://docs.python.org/3/library/profile.html.

12. Parallelism in One Line. [Online] https://chriskiehl.com/article/parallelism-in-one-line.

13. Python 101: An Intro To Benchmarking Your Code. [Online] https://dzone.com/articles/python-101-an-intro-to-benchmarking-your-code.

Extra: Project plan

In order to successfully design and implement the program and meet all the requirements, the following brief plan will be followed:

- Become more familiar with the benchmarking tools for each language by testing them on smaller pieces of code, i.e. initializing data structures / analyzing memory allocation and access performance
    - Python – timeit module
    - Java – JMH Java Microbenchmark Harness
    - C++ - Google benchmark library
- Design the necessary functions to study all the required fields in such a way that they either highlight the similarities or the differences between them, i.e. initializing data structures / objects / simple variables in a static / dynamic way, if possible (e.g. C++)
- Test the performance of each programming languages on simple algorithms to study the efficiency / performance
- Analyze the thread creation process on various number of threads, as well as thread manipulation and synchronization
- Record the results of each of the performed tests
- Contrast and compare the interpreted tests in order to examine the performance differences between the three programming languages

Note: The plan is to implement the benchmark programs in parallel for each programming language, i.e. perform the memory allocation tests for all the languages, then memory access and so on. The plan is to create somewhat of a visual representation of the results for all of them and contrast and compare them in detail within the documentation.