CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Comparing Programming Languages in Google Code Jam

Master's thesis in Computer Science, Algorithms, Languages and Logic

ALEXANDRA BACK

EMMA WESTMAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Comparing Programming Languages in Google Code Jam

ALEXANDRA BACK
EMMA WESTMAN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Comparing Programming Languages in Google Code Jam
ALEXANDRA BACK
EMMA WESTMAN

Supervisor: Carlo A. Furia, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Comparing Programming Languages in Google Code Jam
ALEXANDRA BACK
EMMA WESTMAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

When developing software there are different requirements for what the software needs to deliver. In some cases raw speed performance is the most important factor, while in other cases achieving reusable design is more important. Regardless of the different implementation approaches that can be used, the programming language is likely to affect whether the requirements are met. Thus, knowing how to choose the optimal programming languages for a specific software project is an important task. The goal of this study is to identify weakness and strengths of some popular programming languages based on how they are used in practice by professional programming contestants.

This study empirically examines five programming languages: C, C#, C++, Python and Java. The characteristics of the languages are studied using data developed independently of this study, namely programs submitted as entries in the programming competition Google Code Jam (GCJ). Entries to the contest were downloaded, compiled and executed to measure features of interest such as: lines of code, size of executable file, run time and memory consumption, as well as each entry's final rank in the competition. Furthermore how self contained the languages are is studied using error messages received execution.

The study found no language that is superior in all features. C and C++ give great raw speed performance and use memory most efficient. C# and Java have slower performance and larger footprint compared to C and C++, but provide small executables. Python emerges as reasonable alternative to C and C++.

Keywords: Repository Mining, Programming Languages, Empirical Research

Acknowledgements

We would like to thank our supervisor Carlo A. Furia for his encouragements during this thesis. Carlo trusted us to work independently and was always a great source of advise when difficulties were encountered. This study would not be possible without his help.

Alexandra Back & Emma Westman Gothenburg, June 2017

Contents

List of Figures

List of Tables

Acronyms

CSV Comma Separated Values

GCJ Google Code Jam

LOC Lines Of Code

Pip Python Index Package Repository

1 Introduction

Since the invention of the modern computer in the 1940s various programming languages have been developed. A natural question that arises is which language is to be preferred in the abundance of programming languages. This question is highly debated among computer scientist and programmers. Similarity to other commonly used languages, the availability of open source libraries and the programmer's experience are just some of aspects that influence the choice of programming language [1]. Hence the answer depends on whom you ask and what parameters are taken into account.

Knowing beforehand which programming language to use in a specific context to solve a problem in the most optimal way is a difficult task. Therefore observations from *empirical comparisons* can provide significant guidance both when designing a new programming language and when choosing existing programming language for a project.

Empirically comparing programming languages is an ambitious endeavor since the success of such comparison is affected by several factors. For example, the programmers who wrote the programs under analysis must have similar experience in the language of preference and the problem set must be varied in order to achieve generalizability. Thus solutions have to be written by programmers of comparable skills, solving the same unambiguously specified and sufficiently varied problem set.

In this thesis, we utilize the Google Code Jam repository to compare features of five popular programming languages. Google Code Jam, *GCJ*, is known as one of the most challenging and prestigious programming competitions in the world [2]. The contest attracts up to 20,000 different programmers each year [2] and the GCJ repository collects solutions written in up to 70 different programming languages [2]. The large diversity of problems and programming languages used to solve these as well as the high number of contestants make this repository suitable for a empirical comparison such as this one.

1.1 Problem Definition

The study presented in this thesis examines the characteristics of five programming languages: *C*, *C#*,*C++*, *Java* and *Python*. The aim of this work is to identify, based on empirical grounds, weaknesses and strengths of these languages in relation to one another, and to decide which languages are better given specific parameters. The approach used, in order to derive such findings, is statistical analysis of data originating from the GCJ repository.

1.2 Motivation

Choosing the optimal programming language for a project can save developers, researchers and companies both time and money. There are several possible benefits of choosing the optimal language for a specific task. Reducing the number of bugs, improving the performance of a program and even reducing the time spent on development are just few examples. Thus the cost of training developers to use a more suitable language may in the long term be much cheaper than using the current programming language.

Previous studies [3, 4] investigating language features such as memory consumption, conciseness of code, and so on, looked at a small problem set or used a data set containing solutions to well known computers science problems. This thesis makes use of a larger set of problems for which there exist no obvious standard solutions.

1.3 Research Questions

In order to determine which programming languages are optimal given a set of parameters, the following research questions were investigated during the thesis.

- **RQ1:** Which programming languages make for the top rank in Google Code Jam?
- **RQ2:** Which programming languages make for more concise code?
- **RQ3:** Which programming languages compile the smallest executables?
- **RQ4:** Which programming languages have better running time performance?
- **RQ5:** Which programming languages use memory most efficiently?
- **RQ6:** Which programming languages are more self contained, i.e. which programming languages produce less errors when run on other devices.

1.4 Summary

To give a concrete answer to which programming language is the best is hard. From the examined parameters in this study we conclude that no language is superior with respect to all investigated parameters. According to our results C and C++ perform well when analyzing the majority of the examined features. C and C++ provide the fastest performance, use little memory and tend to provide fairly concise code base; on the other hand C and C++ have large executable files. In contrast, C# and Java have small executable files but longer runtimes, use more memory and are more verbose compared to C and C++. Python tends to place itself in the middle when analyzing most features. However for conciseness of code and small size of the executable Python is the best option. When considering memory consumption of Python, we found that Python ranks better than C# and Java and worse that C and C++; but is closer to the latter. The run time of Python solutions is the greatest weakness of the language we found in this study. Finally, to give an indication of which programming languages are more self-contained, we investigated the different encountered errors. By investigating the encountered errors and study the corresponding solution, we try to determine how easy it is to move a solution from one machine to another machine and make the solution execute without errors. For this question availability of third party libraries weights in as well as how descriptive error messages are in the different languages. In this study Python had the smallest percentage of execution errors and C# the highest percentage. Concluding the results our findings align, in general, with the findings of previous studies [3][4]

1.5 Reading Guidance

This master thesis report is divided into six chapters. The first chapter, i.e. current chapter, gives an introduction and motivation to this study. Chapter 2 gives the reader a general background on empirical research and summarized related research papers. Chapter 3 describes the experimental design, including motivation to the design choices. Chapter 4 describes the implementation and the development tools as well approach for encountered difficulties. Chapter 5 presents the results of this study and answers each research question individually. Reflections on the results, the design choices and suggest ideas for extending the work in the future are presented in chapter 6. The final chapter, chapter 7, concludes our findings.

2
Background

Empirical research is a cornerstone in many areas of scientific inquiry and it lends itself for use in software engineering as well. Section 2.1 provides an overview of how to construct empirical research in software engineering; section 2.2 summarizes empirical studies about topics similar to ours.

2.1 Empirical Research in Software Engineering

The information technology revolution has resulted in software being a vital part of an increasing amount of products. Software engineering is the process of developing new software and involves complexities such as large amount of developers, complexity of the software itself and the long development time [5].

Empirical research has historically been common in social, behavioral and physical sciences since it provides a way of evaluate human-based activities using experiments. However, empirical research can be applied to areas within software engineering as well, for example comparing new tools and languages to existing ones. Applying empirical research to software engineering is suitable since software engineering is a human activity based on creativity and studies ingenuity of developers.

Empirical studies in software engineering involve; setting up formal experiments, studying real projects in industry (e.g. a case study), performing surveys and interviews [5]. This master thesis is a case study and therefore the focus will be on describing this kind of empirical research.

A case study examines a real life context and can be used as a comparative research strategy. When conducting a case study it is important to minimize the confounding factors, in other words factors that have not been accounted for [5]. These factors may lead to biased or misleading results. Planning a case study is rather simple in comparison to other research strategies and the projects can be scaled easily. There are however, some potential problems, for example the difficulty to generalize results. Results may differ depending on the scale of the study and confounding factors may potentially make it difficult to pin point the variables leading to the results [5].

2.2 Related Work

The previously conducted studies have been roughly divided into two categories: *controlled experiments* and analyzing programs in *public repositories*. In controlled experiments a group of human subjects solves small programming tasks in various programming languages under a limited time span. While studies that analyze programs in public repositories investigate code in projects that has evolved under months or even years. The main drawback of controlled experiments is in most cases the use of students as human subjects, due to the limited programming knowledge among them. On the other hand the main drawback when analyzing public repositories is the diversity of the solved problems.

2.2.1 Rosetta Code Study

Nanz and Furia explored the middle ground between controlled experiments and analyzing public repositories using the Rosetta Code Repository [3]. The Rosetta Code Repository collects implementations of over 700 programming tasks, written in hundreds of different languages. Furthermore, the solutions have been revised and improved by other programmers, hence they are close to optimal with regard to conciseness, running time, failure proneness and memory usage.

The study revealed differences regarding some of the most frequently discussed language features such as conciseness, performance and failure proneness. The study concluded that functional and scripting languages are more concise than procedural and object-oriented languages, while C gives the best raw speed performance given large inputs. However, the tasks are relatively small and have well known solutions, thus not suitable for real-life programming challenges.

2.2.2 Controlled Study Using One Given Problem

Another study, similar to Nanz and Furia's, was done by Prechelet in 2000 [4]. A comparison between different languages was performed, investigating similar parameters of interest (i.e. conciseness, programming effort, runtime efficiency, memory consumption, and reliability). However, the environment was controlled and the set of investigated languages was : C, C++, Java, Perl, Python, Rexx and Tcl. In the experiment, several programmers solved a given problem, using one of above mentioned programming languages.

The study revealed that computations in C and C++ ran faster than in other languages, most notably twice as fast as solutions written in Java. C is also superior when dealing with memory usage. However, the scripting languages: Perl, Python and Tcl, were reasonable alternatives to C and C++, even for tasks that require fair amounts of computation and data. The results of this study however, can not be generalized due to the limited amount of tasks that were solved, i.e. only one programming problem.

2.2.3 Code Quality in Github

A study [6] analysing programs in public repositories was conducted in 2014 and focused on code quality in Github repositories. The study investigated 729 projects in 17 languages, in order to probe the effect of programming languages on software quality. Furthermore, the study investigated if some languages are more prone to failure than others. When such relationships was found, the authors looked for what language properties that were related to those defects and however defect proneness depend on domain.

The study concluded that some languages are more often associated with defects than other languages. Although the effect is small it has a significant relationship with the language class. It was found that functional languages contained less defects that procedural and scripting languages. With respect to domain no significant relationship was found.

3
Experimental Design

This chapter describes the work flow of this thesis and includes motivation for our choice of methodology. Sections 3.1 to 3.6 outline the work flow for each part of the project.

3.1 Data Collection: Google Code Jam

The data used in this study originates from submissions to the programming competition Google Code Jam, *GCJ*. Every annual edition of GCJ consist of several rounds to rank the contestants and announce a winner. The first round is the qualification round, followed by rounds 1A-1C, round 2, round 3. Finally, the winner is announced in the World Finals. All submissions to the competition are collected in a public repository, thus accessible to anyone. The availability, the large amount of data and the competence among the programmers provide a good base for a study such as this one.

In each round a set of problems is given. Typically this set contains four problems to be solved using a programming language of choice. Two inputs are provided, in general, i.e. one small and one large input, to test the correctness of the program. The first line of the input file specifies number of test cases, followed by the actual test cases. From the time the input was downloaded, a contestant has four minutes to upload output for the provided input. This to avoid solutions that merely do pattern matching on the given input. In addition to the output file, a contestant has to upload the source code solving the problem; but only the output file is used to verify the solution.

The server gives one of three responses for the submitted output:

1. Accepted, the contestants output was correct.
2. Rejected, the file was rejected due to reason unrelated to whether the output is correct or not, e.g. the contestant might have submitted the source code as output file.
3. Incorrect, the produced output is incorrect. However without revealing whether the output contained some correct cases or not.

If the submission was not accepted, it is up to the contestant to debug his/hers code within the time limit. If the time limit is exceeded, a new set of test cases becomes available to download and the timer is reset. Furthermore if the contestant manages to solve the small input, the large input becomes available. The process for submitting the large input set is the same, however the contestant is given eight minutes to submit [7].

The submission procedure entails that all solutions available in the GCJ repository have somewhat been verified. However, it is not the code itself that has been verified just the output it have produced. We will consider this as the code have been verified to an acceptable extent. Since it is hard to fake correct output without writing a correct program, we consider all solutions in the repository fulfill the requirements to be used as data in this study.

3.1.1 Selection of Competitions

Apart from solutions to the annual edition of GCJ, the GCJ repository also stores solutions to spin offs and different tracks such as: Distributed Google Code Jam and EuroPython. To identify which competitions to include in this study all competitions were examined carefully. Competitions that focused on specific skills such as distribution, and competitions that were limited to a specific group of people or area were disregarded. All competitions in which a sample input was unavailable have been disregarded as well, since the solutions are not executable without a provided input.

Thus this study is focused on the original version of the competition, for the years 2012-2016. We have chosen to include the Qualification round, round 1A, round 1B and round 1C; since the amount of solutions naturally decreases when the final round approaches and performing statistical analysis on too small a data set is prone to be inconclusive. Because the amount of contestants who solved all problems for a round drastically decreased, we decided to only include the first 300 pages for each contest of interest.

3.2 Exploratory Study

An exploratory phase was conducted to investigate factors that could have negative impact on the process of downloading and compiling solutions automatically. In the exploratory phase a small set of solutions was downloaded and compiled manually. Then for each solution in the set that did not compile correctly, an error message was written to a log file. The findings from the exploratory study are presented in section 4.2 and laid the groundwork for the implementation phase of the project, described in chapter 4.

3.3 Language Selection

When deciding which programming languages to analyze in the study two parameters were taken into account: number of solutions submitted to GCJ written in the language and TIOBE index. The first parameter was considered since performing statistical analysis on a small data set is a threat to validity for this study. TIOBE index, the second parameter, is an indicator of how popular a programming language is at the moment based on the number of skilled engineers, courses and third party vendors [8]. The top five programming languages in January 2017 according to the TIOBE index [8] were Java, C, C++, C# and Python, in that order. These five languages are also by far the most represented languages in GCJ each year [9]. Therefore, these five languages were included in the study.

3.4 Task selection

The specific features that were analyzed were determined in the exploratory phase of the project. In order to pinpoint such features, the first step was to investigate if there exist major differences between the solutions when manually downloading them. For example if the solutions vary much with regards to length, the size of the execution file is an interesting feature to analyze. This feature is measurable and therefore a suitable aspect to analyze. Other features, such as how comprehensive the code is to another programmer who was not involved in the development, could have been an interesting parameter to analyze. However, this is not a very suitable parameter to analyze since it can not be measured objectively nor did this project contain data that could measure this aspect. Thereby the features to be analyzed must be suitable for objective measurement. Having this in mind, the following features were selected to be investigated:

- Correlation between rank and language
- Lines of code
- Size of executable file
- Runtime performance
- Maximum memory footprint
- More self contained

3.5 Scripts and Storage of Results

To analyze the features of interest for comparison of language features scripts were written in Python. The scripts were used to compile and execute the solutions while measuring the determined featured. The output, produced by the scripts, was stored in comma-separated values, *CSV*, format and analyzed with statistical methods. A CSV file is a plain text file with tabular data. Each line represents a record where each field is separated by comma or another reserved separator token. The CSV format is not standardized, thus it was tailored to fit this projects' specific needs. Beside CSV being a customizable format, it is also one of the most efficient ways to store data with these characteristics.

3.6 Statistical Analysis

The measured data was analyzed using the statistical methods and tools described in section 3.6.1 to 3.6.3. The results from the statistics were used to draw conclusions answering the research questions and were the underlying material for the discussion. The results were also visualized using tables and diagrams to facilitate the interpretation.

3.6.1 Python Library Pandas

To analyze the output from our measurements we used a Python library, `Pandas`. Pandas allowed for reading a CSV-file to a data frame and perform database like operations on the data frame to select and filter fields. Pandas also provides several graph plotting functions, which were used to plot graphs and diagrams with.

Box plots were used to compare a specific feature among the five investigated languages. Box plot graphs use the median, first quartile, third quartile, minimum and maximum to represent the data. A box plot graph is, as the name indicates, represented by a box. The top edge of the box represent the third quartile, the bottom edge the first quartile and the horizontal line in the middle of the box plot represents the median. The first quartile is the middle value between the minimum and the median, the third quartile is similarly the middle value between the median and the maximum value. The box is also extended with a vertical line from the top edge of the box up to the maximum and another line from the bottom edge to the minimum value. Abnormal values are represented by a small x and are placed on the y-axis accordingly to it's value. However, these abnormal values were not representative and made the graphs unintelligible and therefore were omitted.

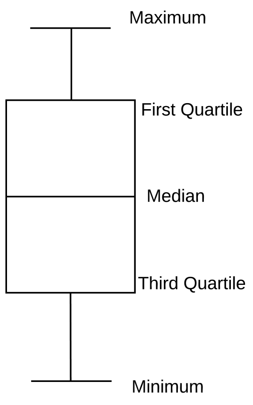


Figure 3.1: Simple example of a box plot

3.6.2 Kendall's τ

For measuring the relationship between two sets of ranked data we used a statistical method called *Kendall's* τ. In Kendall's τ, 1 means a perfect correlation and 0 no correlation. The algorithm to calculate the Kendall's τ value was implemented in Python.

3.6.3 Vargha and Delaney Effect Size Measurement

We used the Vargha-Delaney effect size statistics to decide which of the investigated languages is superior using the observed data from our measurements. The algorithm to compute the effect size takes two vectors as input, A_x and B_x, where x denotes the feature of interest and A, B denote the programming languages used, and returns a value between 0 and 1. A returned value of 0.5 indicates that the programming languages are stochastically equivalent. A returned value closer to 1 indicates that language A performs better than language B for the measurement x. A value closer to 0, on the other hand, indicates that B is superior.

4 Implementation

This chapter will guide the reader through the implementation phase of the project. The first section, section 4.1, will explain the environment in which the experiments were conducted. Secondly in section 4.2 the exploratory phase will be discussed, followed by section 4.3 describing how data was retrieved and processed, using our scripts. In section 4.4 we will describe how various compilation and execution errors were handled. Finally, information about how the tasks examination and statistical analysis were conducted will be presented in section 4.5. The challenges encountered whilst the implementation phase will be portrayed throughout this chapter at their specific occurrence.

4.1 Environment

The study was conducted on the virtual machine running the operating system *Debian-8.7.1-amd64* hosted on virtual machine monitor *Qemu*. The examined solutions were placed on the virtual machine, where compilation and execution were performed using the compilers listed in Table 4.1.

The scripts for running the solutions automatically were written in Python and executed with Python 2.7.3. Further *Cloc*, a script for counting lines of code and used for the solution files, was installed in the virtual environment together with a Python package manager *Pip*. This manager was used to download and to install module from Python Index Packge Reposity. Python libraries *Pandas* and *Matplotlib* were used to plot graphs for the statistical analysis.

Table 4.1: Compilers

Language	**Compiler Version**
C	gcc 4.9.2
C#	mono 3.2.8
C++	gcc 4.9.2
Java	Java SE 1.8.0_131
Python	2.7.9 or 3.4.2

Git was used as the version control tool for this project. There are three repositories for this project; one for the data collection, containing all the solution files and input files. Another one for the CSV files, where the results from the statistical analysis were stored and finally a repository for the scripts executing solutions automatically.

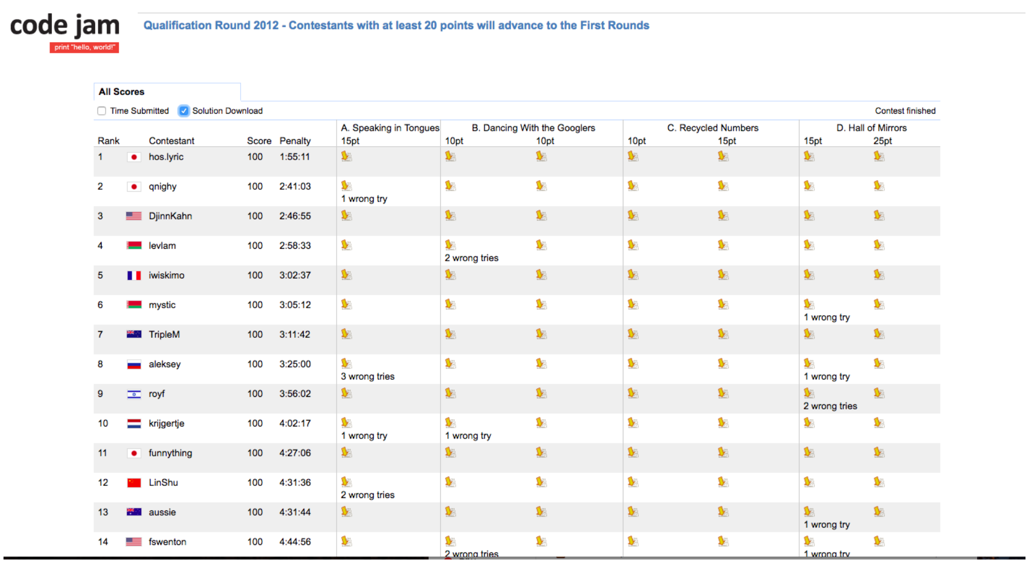
4.2 Exploratory Study

In the exploratory study we examined how the download process could be automated and if there existed challenges related to automatically compile and execute the solution files. Our course of action when exploring how to download solutions is described in section 4.2.1 and our approach to finding compilation and executing challenges is described in section 4.2.2.

4.2.1 Description of the Google Code Jam Repository

The GCJ repository collects solutions to previous competitions categorized by the year of the competition. Each competition consists of several rounds where each round has an unique id. Each round, in turn, consist of several problems also identified by an unique id. For each problem two input sets are given, one smaller and one larger. Thus a contestant must upload two solutions, one solving the smaller case and another solving the larger case. However, in most cases, contestants have submitted the same program twice.

When clicking on a specific round, a scoreboard of that round is displayed. The scoreboard is presented as a table, displaying 30 entries on each page. Number of pages for a round varied depending on the round; an qualification round consisted of up to 27000 pages, while the final round contained not even 30 entries. An entry, in turn, contains: alias of the contestant, rank, score, and time spent to solve all problems and finally links to zip files containing the solutions to the specific problem, see image 4.1.



Qualification Round 2012 - Contestants with at least 20 points will advance to the First Rounds

All Scores

☐ Time Submitted ☑ Solution Download

Contest finished

				A. Speaking in Tongues	B. Dancing With the Googlers		C. Recycled Numbers		D. Hall of Mirrors	
Rank	Contestant	Score	Penalty	15pt	10pt	10pt	10pt	15pt	15pt	25pt
1	hos.lyric	100	1:55:11	⬇	⬇	⬇	⬇	⬇	⬇	⬇
2	qnighy	100	2:41:03	⬇ 1 wrong try	⬇	⬇	⬇	⬇	⬇	⬇
3	DjinnKahn	100	2:46:55	⬇	⬇	⬇	⬇	⬇	⬇	⬇
4	levlam	100	2:58:33	⬇	⬇ 2 wrong tries	⬇	⬇	⬇	⬇	⬇
5	iwiskimo	100	3:02:37	⬇	⬇	⬇	⬇	⬇	⬇	⬇
6	mystic	100	3:05:12	⬇	⬇	⬇	⬇	⬇	⬇ 1 wrong try	⬇
7	TripleM	100	3:11:42	⬇	⬇	⬇	⬇	⬇	⬇	⬇
8	aleksey	100	3:25:00	⬇ 3 wrong tries	⬇	⬇	⬇	⬇	⬇ 1 wrong try	⬇
9	royf	100	3:56:02	⬇	⬇	⬇	⬇	⬇	⬇ 2 wrong tries	⬇
10	krijgertje	100	4:02:17	⬇ 1 wrong try	⬇ 1 wrong try	⬇	⬇	⬇	⬇	⬇
11	funnything	100	4:27:06	⬇	⬇	⬇	⬇	⬇	⬇	⬇
12	LinShu	100	4:31:36	⬇ 2 wrong tries	⬇	⬇	⬇	⬇	⬇	⬇
13	aussie	100	4:31:44	⬇	⬇	⬇	⬇	⬇	⬇ 1 wrong try	⬇
14	fswenton	100	4:44:56	⬇	⬇ 2 wrong tries	⬇	⬇	⬇	⬇ 1 wrong try	⬇

Figure 4.1: Screen shot of scoreboard for Google Code Jam Qualification Round 2012. *Taken: 2017-04-04*

4.2.2 Identified Challenges when Compiling and Executing Solution Files

During the exploratory study it was found that most of contestants using C#, Java, Python (and in some C++ programs) had hard coded the path to the input file. This issue together with compiler version used and dependencies on external libraries were the most common problems found when trying to execute the solution files manually. Thus, when executing such a program an exception is raised, containing information about the missing file. Furthermore, an error found mostly in C#, was the absence of the main method which naturally made the solution file non executable. We also noticed that not all errors could be solved automatically, for example we encountered solutions that got stuck in infinite loops or programs that raised `Index Out Of Bound Exceptions`.

4.3 Using Scripts For Downloading, Compiling and Executing Solutions

In section 4.3.1 we describe how the download links were reversed engineered and how the script uses information from the GCJ repository to download solutions. This followed by section 4.3.2 describing the process of partitioning data by language to facilitate compilation and execution with the correct commands; compilation and execution are described in section 4.3.3. Finally, the section 4.3.4 portrays the process of measuring and storing features of interest.

4.3.1 Implementation of Downloading Solutions

To download solutions automatically, a download link for each zip file was built. The format of the URL is not specified officially, so we reversed engineered it by scraping the GCJ web page. By using Chrome's tool for inspecting a web page, we found that each link contains: alias of a contestant, id of a problem, a flag referring to input size and an id of the contest round. To extract an id for each contest, the page listing all contests was scraped. Then by using the id of a contest, each individual contest page could be accessed, i.e. each contest scoreboard. From each contest page both the contestants' aliases and problem ids for the contest round were obtained. After retrieving this information the downloading of zip files could start using the scraped contest id, problem id, flag and alias used to build the download link. Additionally, to retrieve information for each contestant about rank, score, and time spent to solve the given problems, data from the scoreboard was downloaded from the REST API as a JSON message.

To speed up the downloading process the downloading was done in parallel using multiple processes. We used Python's standard module `multiprocessing` to distribute the downloading of different contests to all available cores. The zip files were downloaded into a directory, which is available on GitHub [10]. By using a dedicated GitHub repository for storing solutions, we were able to revert all files to their original state. Reverting to original state facilitated the process of testing scripts for error handling.

4.3.2 Partitioning Solutions

To simplify compilation and execution, the solution files were partitioned by language. Hence all zip files were sorted while being unpacked. Before the unpacking started, five directories were created in the problem id directory. Each of the five directories represents one of the selected languages: C, C#, C++, Java or Python. Then for all zip files a directory with the same name as the zip file, i.e. the alias, was created inside the correct language directory. Finally the solution files were placed inside the alias directory. The partitioning process also took care of discarding solutions written in programming languages other then the five selected ones.

While sorting the zip files a CSV file for each problem id was created. Each CSV file was initialized with a tuple containing the contestant's alias and programming language used to the problem, where the name of contestant is used as key for each row to store additional data later on. Each row in a CSV file stores all data related to that contestant and that problem id. The data in the CSV files was analyzed in the statistical phase. All CSV files can be accessed on GitHub [11].

4.3.3 Compilation and Execution of Solutions

To automatically compile and execute the downloaded data a set of scripts in Python was written; one for each language. A master script written in Python was used to start the compilation and execution of the downloaded solutions. Module `subprocess` was used to launch a new process for compilation or execution in the host environment. We also passed the full path to the solution file and redirected the input file. Additionally, the usage of subprocess module allowed us to retrieve error message, output and exit code of the process. This information revealed the outcome of the command and hence the script can take action accordingly; either continue to the next solution or make an attempt to resolve the error and then try again. In the case of failure, the script tries to patch the error and re-run the solution. The last output is stored to the CSV file.

During execution, a time limit of 10 seconds was set. If a solution failed to finish within this limit it was forcefully terminated. However, it was found that a considerable amount of solutions failed to finish within this time limit. When studying such solutions manually it was found that they indeed worked, however some needed more than 10 seconds to finish. Unfortunately this was discovered late in the thesis, thus we did not manage to redo the execution of these files a second time. The goal was to execute the solutions that were caught in the timeout during the first run but using a higher time out of 3 minutes.

When compiling and executing the solutions on the virtual machine occasionally the virtual machine entered the `READ-ONLY` state. Instead of the master script executing all problems sequentially, a bash script was written. The bash script launched the Python script for compilation and execution of solutions, however for one problem and one language at time.

4.3.4 Measurement of Features

For each downloaded solution a script was run on the file to count: lines of code, comments and blanks. The output was appended to existing values for the row corresponding to the contestant who wrote the program and stored in a CSV file corresponding the problem id. The actual counting of lines in the solution files was performed by a script downloaded from GitHub named `cloc` [12].

Most part of the measurements were carried out under compilation and execution. During compilation the exit code of the compile command and compiler version were stored. The exit code is an integer, 0 representing success of the command, while any other number represents different errors which caused the compilation to fail. If compilation was successful the script also measures the size of the executable file produced. During the execution phase each execution command was run with the set of flags listed in table 4.2 for measuring the remaining properties of interest. In some cases the solution contained errors that the script tried to resolve. If patching of error(s) succeeded, features of interest were measured again and the resulting values were stored in the CSV file.

Table 4.2: Flags used under the execution and description of them

Flag	Description of Measurement
%x	exit code
%e	wall-clock time
%U	user time
%S	system time
%K	average of memory allocated to the process
%M	maximum of RAM usage (resident set size)
%t	average RAM usage (resident set size)
%F	number of page faults (major, that is requiring I/O)
%O	number of file system outputs
%I	number of file system inputs
%W	number of swapping out of main memory

4.4 Error Handling for Compilation and Execution

In this section we present how the errors were automatically fixed using the script. As described in the Exploratory Study, section 4.2, various reasons can cause compilation or execution to fail. In order to increase the success rate of executing solutions and get more data, the script tries to resolve the error. Depending on the error and language different error handling strategies was used.

Despite the fact that several languages encountered problems of similar characteristics, the error handling had to be tailored to the specific language. This due to the fact that different languages have different syntax and semantics. Thus, for each language we have dedicated a section describing errors encountered and strategies used to solve these. However, most of the errors encountered were related to the handling of the input and the output file. Some solutions expected input and/or output provided as an argument during execution. While other solutions used hard coded paths causing `FileNotFound` error to be raised. For `FileNotFound` we used the same idea for all languages and this technique is presented in section 4.4.1.

We tried to fix simple errors that did not required manual altering the solution file. However, several of the encountered errors could not be automatically resolved. Hence some solutions were left uncompiled or/and unexecuted. Fixing every error would simply take too much time and not all errors could be fixed automatically, e.g. missing non-standard libraries.

4.4.1 Handling File Not Found Exception

File not found was resolved automatically by the script modifying the source code. The hard coded paths to input and output were found using regular expressions and replaced with the path to input and output on the virtual machine. Capturing groups were used to only replace paths and not whole statements. In most cases these tactics could successfully change the paths. However there are many ways to read input and write output in each language and therefore, close to impossible to patch all cases.

4.4.2 C Specific Error Handling

When compiling C solutions mainly two errors were encountered: *undefined reference* to standard math library and the need of specifying the *compiler version*. C programs that use math functions require math library to be linked with the source code during compilation. This is achieved by adding `-lm` flag to the compilation command. Some files also used version specific syntax of the `std C11` version of the compiler, thus these solutions were run with the flag `std=c11`. Iostream errors were solved by compiling and then executing the file using the C++ compiler. However other errors encountered in C were not correctable since these involved imports of self written `.h` files that were not include in the submitted solution or errors of similar characteristics. A flow chart illustrating how the script deals with all errors in C is available in appendix A, figure A.1.

4.4.3 C# Specific Error Handling

A program written in C# usually is developed using Visual Studios. Since the contestants did not include the whole project, several of the C# solutions did not include a main method when submitting their code. It is therefore the most common issue found in C# was the absence of the main method.

For those solutions that did not include a main method, a main class was created automatically by the script, i.e. `TestMain.cs`. The main class has a static format where only the name of the namespace and the method call is changed depending on the solution. An example of a main class created by the script is shown in figure 4.1, here the submitted file is named `Solution.cs`. The name of the namespace has to be the same as the namespace in the solution file and can be extracted from the solution file using regular expressions. In the example the namespace in the file `Solution.cs` is `GoogleCodeJam.CountingSheep` thus the script gives the namespace in the main class the same name.

To find the method to call from the main method is however more challenging. We saw no other possibility than finding the executing method by brute force search. However, we limited the search to static methods, thus the name and signature of all static methods are collected. Then all static methods are called one by one from within the main method. If the program runs successfully, the selected method is retained and the measured values are stored, otherwise the search continues by trying next static method. If no appropriate method was found, the solution file is disregarded.

Listing 4.1: Automatically Generated Main Method

```
//TestMain.cpp
namespace GoogleCodeJam.CountingSheep{
   class TestMain
   {
      static void main()
      {
         Solution.SolveProblem();
      }
   }
}
```

Another error that was fixed for C# was dynamically-linking standard libraries under compilation. If an error message for absence of a specific library is received, the the missing library is added to the command and the process of compilation and execution is started again. A flow chart illustrating how the script deals with all errors in C# is available in appendix A, figure A.2.

4.4.4 C++ Specific Error Handling

Most C++ programs included a file named `stdc++.h`. Since this file were used by a significant amount of contestants, it was downloaded from github [13] and put in the directory `/usr/local/include/bits`. The benefit of including this file is that it imports several commonly used libraries at once.

Several C++ programs also required to be run with a flag,`-std=c++0x`, specifying which version of the compiler to use. Since this flag does not affect programs that are meant to run with lower versions of the compiler, all programs are executed with this flag to reduce the number of cases in the script and reduce compile time. Another issue we found in some C++ files was related to the signature of the main function. The correct signature of the main function in C++ is `int main()` when C++ programs are developed in Visual Studio the compiler for C++ accepts programs with main signatures of simply `main()` or `void main()`. When these errors were encountered the scripts tried to replace the signature to `int main()` using matching on regular expressions and then recompile the file. A flow chart illustrating how the script deals with C++ errors is available in appendix A, figure A.3.

4.4.5 Java Specific Error Handling

Some of the solution files written in Java files had a package name declared at the top of the source code. This package name caused the execution of the program to fail. The solution to this problem was to remove package name and re-run the file. A flow chart illustrating how the script deals with Java errors is available in appendix A, figure A.4.

4.4.6 Python Specific Error Handling

The most common error found in Python was related to the two versions of the Python environment. Some contestants have used the 2.x branch and others the 3.x. Since these versions are not compatible with each other, a program must be syntactically correct for that version or a generic syntax error will be thrown. Thus, the correct branch needed to be specified when interpreting. Furthermore there is no way to check beforehand which version is required for the interpretation and therefore the script first tries to execute the solution using Python 2.x. If a syntax error was thrown, the script tried to execute the solution using Python 3.x. However, not all syntax errors are related to the version of the environment; syntax errors encountered were also typos, which we were not able to correct automatically.

Another common error found among Python solutions was the absence of an imported module used in the solution file. This caused an `Import error` to be thrown. The script solved this problem by parsing the error message for the library name. The script then tried to install the missing library, if it is available, using the package manager Python Index Package, Pip. Finally, the script tried to execute the program again. However in most cases the missing packages were not published and hence could not be downloaded. A flow chart illustrating how the script deals with Python errors is available in appendix A, figure A.5.

4.5 Statistical Analysis

We used the Pandas library to read the CSV files, containing the observed data, and merge them into one single data frame. From this data frame feature specific values could be selected and grouped by language for comparing. Box plots and tables were used as an aid for comparison when answering the research questions stated in section 1.3.

We implemented Kendall's τ and the Vargha and Delaney effect size measurement using Python. The Kendall's τ function was used to investigate if there is a correlation between language and rank. To be able to apply this method, we calculated the mean rank for each language in each competition and ordered them according to mean rank. We then applied pairwise Kendall's τ on the ordered lists, to get a value indicating if there exists a similarity of the rank for those two competitions.

Vargha and Delaney was used in all research questions to analyze which of the two languages was superior given an investigated feature. Using Pandas we were able select the feature column and feed the data to the Vargha and Delaney algorithm. The output from the algorithm was visualized with tables.

5
Results

In this chapter, we present the outcome of this study. Firstly, we present the amount of data that we were able to successfully make use of in this study. This is followed by section 5.2 presenting the findings of the statistical analysis that lay the foundations for answering the research questions put forth in this thesis.

5.1 Successful Compilations and Executions

In total 236 428 solutions were downloaded, whereof 18 744 were written in programming languages other than those selected for in this thesis. The majority of the analyzed solutions were written in C++, followed by Python and Java which had approximately as many solutions. The amount of solutions written in C# and in C were significantly fewer compared to the other three languages.

Figure 5.1 illustrates, for each of the investigated languages, the percentage of the downloaded solutions that we manged to compile and execute. As described from the figure C, C++ and Python have the highest percentage of executable files. Furthermore, table 5.18 summarizes the number of downloaded, compiled and executable solutions of each language.

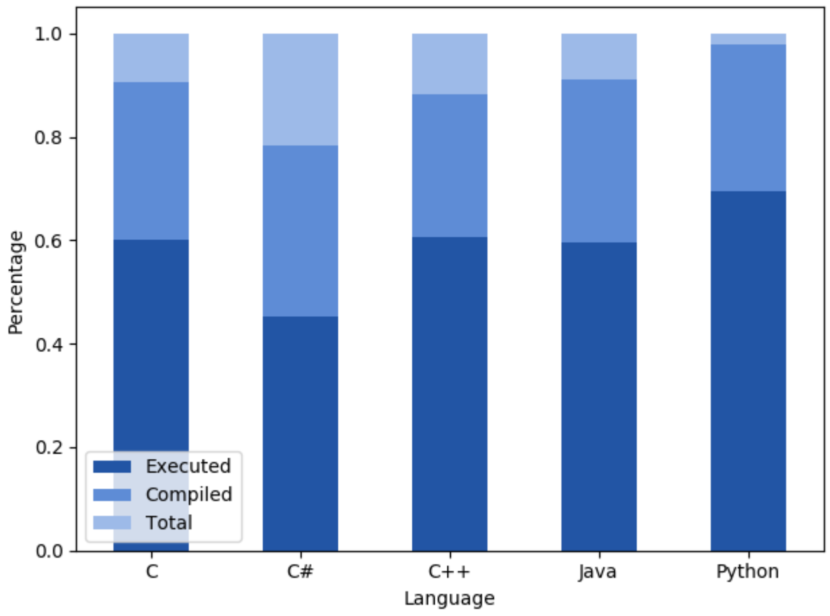


Figure 5.1: The percentage of successfully compiled and ran solutions among total.

Table 5.1: Number of Solutions

Language	Number of Solutions	Compiled	Executed
C	4685	4290	2816
C#	6823	5417	3084
C++	136085	120928	82572
Java	33187	30524	19749
Python	36904	36497	25597
Total	217684	197656	133818

5.2 Answers to the Research Questions

In this section we present our findings based on the statistical analysis that was made on the solutions that we succeeded to compile and execute. These findings are used to provide an answer to the research questions stated in this thesis.

RQ1. Which programming languages make for the highest rank?

We investigated if there is a correlation between the programming language used in the competition and the contestants rank. The languages used in GCJ vary from year to year. However C, C#, C++, Java and Python have been the most used languages during the recent years, where C++ is the most used language in the competition and therefore accounts for the highest amount of solutions.

A contestant's rank in a round is based on the combined performance in all problems of that round. For this analysis contestants that used the same language for all problems in a round were included and the languages were ordered based on their mean rank. It would have been interesting to investigate if a combination of programming language could make for the highest rank, however this question is outside the scope for this thesis.

Table 5.2 lists all rounds and languages that were included in the study; for each round the left most language had the lowest mean rank, i.e. performed better, and the rightmost the highest mean rank. We found that the most common ordering is: C++, Java, Python, C# and C. This ordering occurs three times, in qualification round 2015, 2013 and round 1C 2012. Looking at which position each language ranks most often we found the same pattern among the languages. C++ is ranked first most often, 17 times. Java most often ranks in second place, 8 times. Python most often rank as the third language, 6 times. C# most often rank as language number four, 6 times and C most often ranks as the fifth and final language, 9 times. Such ranking is not surprising because solutions written in C++, as mentioned above, account for the highest proportion of downloaded data.

We used Kendall's τ statistics to compare rounds pairwise in order to get a value on how equal the ordering of the mean rank is. The complete table, table B.1, displaying the resulting values can be found in appendix B. From this table

Table 5.2: Consistency mean rank, left most language has lowest mean rank

Contest	**Mean rank ordering**				
Qualification Round 2016	C++	C	C#	Java	Python
Round 1A 2016	Python	C++	C#	Java	C
Round 1B 2016	C++	Java	C#	C	Python
Round 1C 2016	C++	Java	C	C#	Python
Qualification Round 2015	C++	Java	Python	C#	C
Round 1A 2015	C#	Python	Java	C++	C
Round 1B 2015	C++	Python	Java	C	C#
Round 1C 2015	C++	C#	Java	C	Python
Qualification Round 2014	C++	Python	Java	C	C#
Round 1A 2014	C++	C#	Python	Java	C
Round 1B 2014	C++	Java	C#	C	Python
Round 1C 2014	C++	C	Java	Python	C#
Qualification Round 2013	C++	Java	Python	C#	C
Round 1A 2013	C++	Java	Python	C	C#
Round 1B 2013	C++	Python	C	Java	C#
Round 1C 2013	C++	C#	Python	Java	C
Qualification Round 2012	C++	Java	C#	Python	C
Round 1A 2012	C++	Python	Java	C#	C
Round 1B 2012	Python	C++	C	C#	Java
Round 1C 2012	C++	Java	Python	C#	C

we conclude that in the worst cases there are no similarity between two competitions. In the best cases, in contrast, there are two competitions with exactly the same ordering. We found that the order the languages rank in are consistent with approximately 60% overall.

To get an indication for which language make for the highest rank, we used Vargha and Delaney effect size statistic to compare languages pairwise based on rank. Table 5.3 shows the resulted values of the comparison, where languages on the y-axis are compared to languages on the x-axis. This statistics indicates that C++ is the superior language, i.e. makes for the highest rank in the competition.

Table 5.3: Vargha and Delaney for Contestants Rank

Language	C	C#	C++	Java
C#	0.54			
C++	0.62	0.58		
Java	0.55	0.51	0.42	
Python	0.55	0.51	0.43	0.51

We manually inspected around 100 of the downloaded solutions to find advantages for using C++ instead of other investigated languages. However no such traits could be discovered. It is therefore hard for us to pinpoint the features that favour

C++ in the competition.

RQ2. Which programming languages make for more concise code?

To compare the overall conciseness of solutions between languages, lines of code, *LOC*, were counted for each solution. Given that the repository contained inadequate solutions, we only measured LOC for solutions that executed successfully. Furthermore, we manually investigated solutions that had fewer than 20 lines of code for correctness. In this set we identified solutions that clearly did not solve the given problem and these solutions were disregarded from the analysis.

Table 5.4 shows the minimum, the maximum, mean, median and sum of LOC for each language. The largest gap of 47 LOC, when analyzing the median values, can be found between Python and C#. Since the analyzed programs are relatively short in general, this is a significant difference. When analyzing the difference between the medians of procedural languages, C and C++, and the object oriented languages, Java and C#, the largest gap of 32 LOC can be found between C# and C.

Table 5.4: Lines of code (LOC) for successful execution

Language	Min	Median	Mean	Max	Sum
C	12	46.00	55.34	510	155454
C#	16	77.50	99.81	1395	307619
C++	11	55.00	63.97	3087	5273389
Java	16	65.00	78.46	3193	1548896
Python	1	30.00	38.01	483	973026
Overall	1	55.00	67.12	3193	8258384

The box plot, figure 5.2, shows that Python tends to provide the lowest mean and the smallest variance of LOC. The median value of Python is between 1.5 and 2.6 times shorter compared to the other languages. C#, on the other hand, tends to be the most verbose and with the highest variance. In the comparison between procedural and object oriented languages, it can be noticed that the procedural languages tend to be slightly more concise than the object oriented languages.

Table 5.5 displays the Vargha and Delaney statistics applied to the measured LOC. The output from this statistic suggests that Python has an advantage having effect size up to 0.8 compared to other investigated languages. Furthermore, table 5.6 shows the rank of the languages ordered by median, mean and the Vargha and Delaney statistics. As can be noticed from the table the ordering of the values from the Vargha and Delaney statistic align with the ordering of the median and then mean values derived from the table 5.4. These results are consistent with the study done by Nanz and Furia [3], as well as findings by Prechelet [4]

.

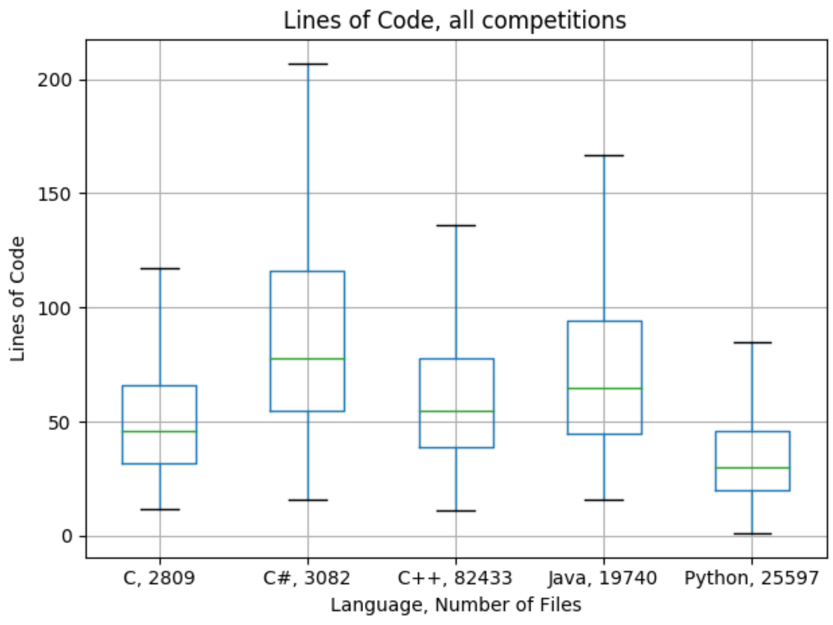


Figure 5.2: Box plot for lines of code (LOC)

Table 5.5: Vargha and Delaney results for lines of code (LOC)

Language	C	C#	C++	Java
C#	0.24			
C++	0.41	0.69		
Java	0.33	0.61	0.41	
Python	0.70	0.88	0.77	0.82

Table 5.6: Rank of languages for lines of code (LOC)

Median	Python	C	C++	Java	C#
Mean	Python	C	C++	Java	C#
Box Median	Python	C	C++	Java	C#
VD	Python	C	C++	Java	C#

RQ3. Which programming languages compile into smaller executables?

Knowing which programming language compiles in to the smallest executable is useful when code has to be run on a device where memory storage is a limitation. We measure size of executable to give an answer to this question and take into account only solutions that execute without errors or time out and have at least 10 lines of code, apart from Python where a working solution was found having one line of code. This study was carried out without usage of optimization flags for making the compiler attempt to improve code size and performance. Our findings are shown in table 5.7 and in the corresponding graph 5.3.

Table 5.7: Smaller executables, in bytes

Language	Min	Median	Mean	Max	Sum
C	6552	7864.00	79424.76	100008312	223660136
C#	3072	4608.00	5317.23	26112	16398336
C++	1811	12336.00	22852.63	168009808	1886576214
Java	247	2110.50	2345.31	63330	46305708
Python	334	1354.00	1663.26	113268	40302501
Overall	247	4608.00	22320.64	168009808	2213242895

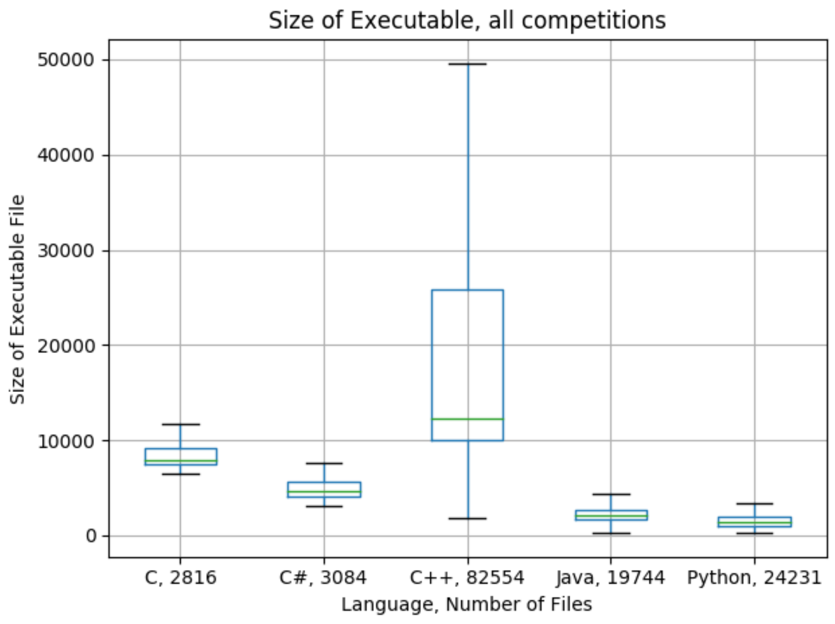


Figure 5.3: Box plot for size of executable file, measured in bytes.

C#, Java and Python compile into bytecode, which is interpreted by a virtual machine under execution. C and C++, on the other hand, are complied into assembly code that is translated to machine code. From table 5.7 it can be seen that among languages that compile to bytecode Python accounts for the smallest executable; having executable size around 30%, when analyzing the medians, of the size produced by C#. However, the smallest executable was found in Java. Among languages that compile to machine code C produces the smallest executable, when considering mean and median size of executable. When comparing these two groups, it can be found that languages which compile into bytecode produce smaller executables than languages that compile to machine code. The greatest difference is found between Python and C++, where C++ produces, when median is considered, an executable 9 times bigger. The differences found between these two groups agree with the study done by Nanz and Furia [3].

The Vargha and Delaney statistics, whose results are displayed in table 5.8, shows that solutions written in Python tend to have the smallest executables, followed by Java and C#. On average, C and C++ tend to produce largest executa-

Table 5.8: Vargha and Delaney results for size of executables

Language	C	C#	C++	Java
C#	0.94			
C++	0.06	0.02		
Java	1.00	0.97	1.00	
Python	1.00	0.98	1.00	0.75

bles, up to 13 times larger that executables produced by Python. These findings are coherent with the box plot.

Table 5.9: Rank of languages for size of executable

Median	Python	Java	C#	C	C++
Mean	Python	Java	C#	C	C++
Box Median	Python	Java	C#	C	C++
VD	Python	Java	C#	C	C++

To summarize the findings of the size of the executable file for the investigated languages we have ordered them in table 5.9 with the language with the smallest executable to the leftmost side for each statistics. As the table illustrates, Python and Java conduct the smallest executable. However there is some variance among solutions written in C++. The large variance in size of executables produced by C++ increased complexity in the comparison of executable size, in relation to other languages .

RQ4. Which programming languages have better running time performance?

Running time of software can be crucial and save a significant amount of time when computing large amount of data. Therefore time performance is an important aspect to consider when choosing a programming language for implementation. To analyze which programming languages have better time performance we executed the downloaded solutions with the corresponding input file downloaded from GCJ repository. When executing the solutions three measurements related to time were measured :

1. *Wall clock time:* time from start to finish of process call, includes time used by other process and time when the process is blocked, e.g. waiting for IO to complete.
2. *User time:* time spent in user mode (outside kernel) within the process, i.e. the actual CPU time used executing the process.
3. *System time:* time spent in CPU inside kernel within the process, e.g. allocating memory, accessing disks and network card.

Solutions which successfully executed and had more than 10 lines of code, were the only ones included for this research question. To determine which languages have better running time performance we investigated user time and system time, however problems emerged as follows. For user time it cannot be assumed that all contestants have solved the given problem with the same time complexity. Thus comparing the user time among the five investigated languages will give an indication of the time complexities of the solutions and not the speed of the languages. System time, in turn, is affected by number of read and write operations to a file and which standard methods are used to achieve this. Since a considerable amount of solutions read and/or wrote to a file using different methods, this metric is not useful when investigating which programming languages have better time performance. Thus, for time performance we analyzed wall-clock time, which measures the overall time elapsed in practice for a user's point of view. Resulting values are shown in table 5.10 and in the corresponding graph 5.4.

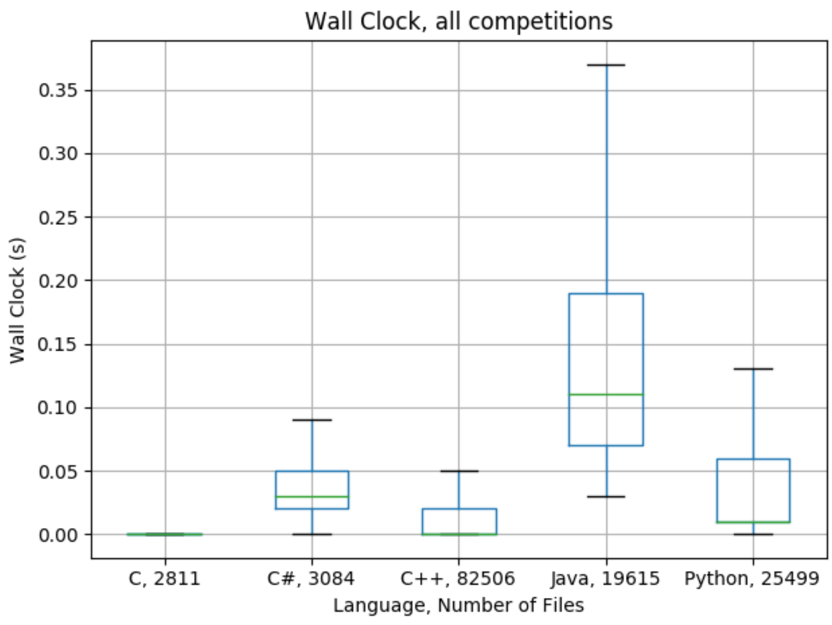


Figure 5.4: Box plot for wall clock time, in seconds

Table 5.10: Wall clock time, in seconds

Language	Min	Median	Mean	Max	Sum
C	0.00	0.00	0.15	9.95	421.92
C#	0.00	0.03	0.28	9.89	854.29
C++	0.00	0.00	0.26	9.98	21743.92
Java	0.03	0.11	0.35	9.95	6919.95
Python	0.00	0.01	0.30	9.97	7731.96
Overall	0.00	0.01	0.27	9.98	37672.04

Graph 5.4 shows that C and C++ have the fastest median wall-clock time. However C++ has an additional variance, compared to C. This difference can originate

from the fact that C only accounted for a small percentage of all downloaded solutions in contrast to C++ that accounted for the higest percentage. Solutions written in Java run, on average, up to 2.3 times slower than solutions written in C. Python and C# perform almost equally good when looking at the graph, through Python having slightly better median wall-clock time. Further Python's median time is not significantly longer than that for C and C++. The fastest solutions in Python runs as fast as those written in C and C++. If we aggregate programming languages according to programming paradigms, procedural languages run approximately 1.5 times faster, on average, than object oriented and twice as fast as Python. In the table 5.10 the resulting median values are coherent with the values displayed in the graph, however when taking into account the mean values Python emerges as the slowest option.

Table 5.11: Vargha and Delaney results for wall clock time

Language	C	C#	C++	Java
C#	0.16			
C++	0.45	0.79		
Java	0.11	0.17	0.14	
Python	0.23	0.67	0.29	0.81

The results using the Vargha and Delaney statistic seen in table 5.11 agree with the results displayed in the box plot. C, indeed, tends to have the wall-clock time among the languages. Java, in contrast, appears as the slowest option, having statistical effect size less than 0.2.

After further considerations we also decided to investigate the user time additionally, since it may given an indication for which language it is easier to implement algorithms with low time complexity. The results for the user time measurements are displayed in table 5.12 and figure 5.5. As can be seen in figure 5.5 C and C++ which compiles into assembly code runs notably faster than C#, Java and Python which compiles into bytecode. Bytecode has to be interpreted during runtime, thus acquire longer time for execution. Among the interpreted languages, Python runs the fastest and Java the slowest. C# has the smallest variance for the user time, this might be a result of having significantly fewer solutions in C# compared to Java and Python.

As can be observed when comparing the results form the wall clock time and user times, the languages preform the same when considering how they rank against each other. Correspondingly observing the time, the user time is naturally slightly lower than the wall clock time.

Table 5.13 shows the results of when analyzing the Vargha and Delaney for the user time. The results indicates the same rank of the languages as for the Vargha and Delaney for the wall clock time. However the values in table 5.13 are closer to 0 or 1 than the corresponding values in table 5.11. The effect size values being closer to 0 or 1 for the user time compared to the wall clock time shows that the Vargha and Delaney results are more significant for the user time than the wall clock time. Going back to the box plots and studying the variance the box plots illustrates that

Table 5.12: User time, in seconds

Language	Min	Median	Mean	Max	Sum
C	0.00	0.00	0.11	9.46	318.28
C#	0.00	0.02	0.26	9.87	790.32
C++	0.00	0.00	0.18	9.98	14986.14
Java	0.01	0.09	0.33	11.00	6520.22
Python	0.00	0.01	0.29	9.95	7390.48
Overall	0.00	0.01	0.23	11.00	30005.44

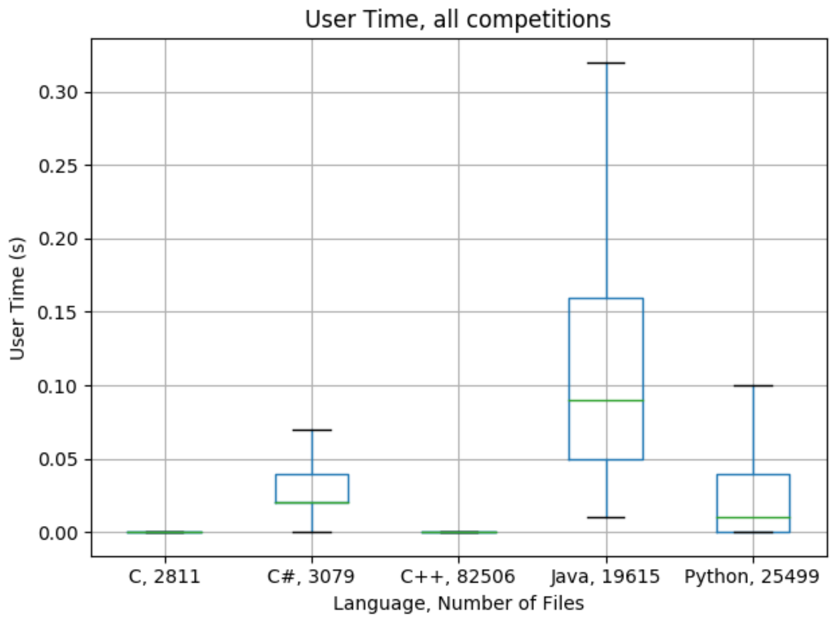


Figure 5.5: Box plot for user time, in seconds

Table 5.13: Vargha and Delaney results for user time

Language	C	C#	C++	Java
C#	0.15			
C++	0.47	0.81		
Java	0.10	0.17	0.13	
Python	0.35	0.70	0.38	0.81

the variance is larger for the wall clock time than the user time. Thus the Vargha and Delaney for user time is more convincing than for the wall clock time.

Further to investigate the run time, the format of GCJ allowed for analysis of how user time scales with respect to input size, since close to all problems were provided with two input set, one small and one large. Analyzing how the user time scales can indicate the complexity of the algorithms used for each language, but to determine the complexity more input sizes are needed.

In GCJ there were two possible variations of how the small and large input could differ, either the number of test cases differed or the length of each case differed. However calculating the difference in user time for the small an large input can serve as an indication on how well the languages scale with respect to time consumption. The results for this analysis are displayed in figure 5.6 and in table 5.14.

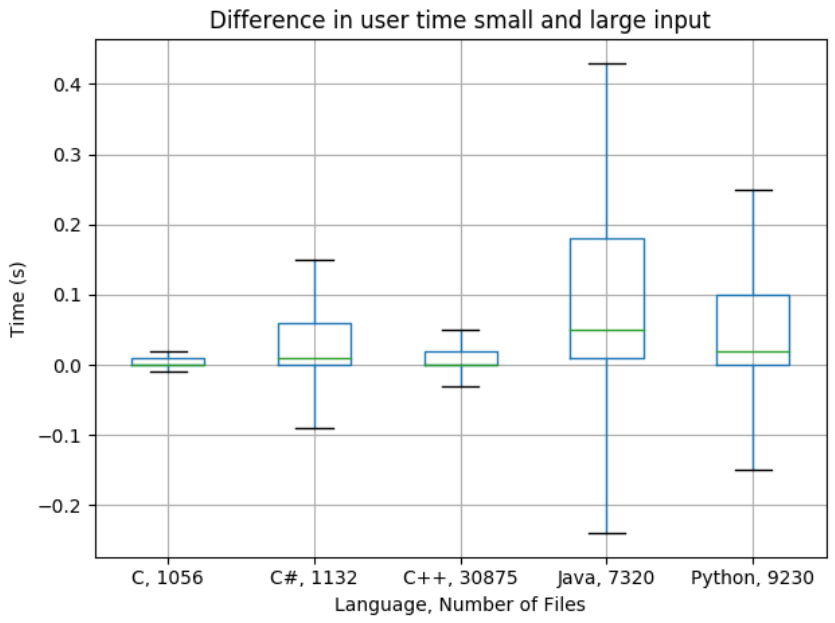


Figure 5.6: Box plot for difference in user time between small and large input, measured in seconds

Figure 5.6 visualize with a box plot the difference in user time between the small and large input. The difference has been calculated by comparing user time (user time large input - user time small input) for all user whom got accepted for both the small and large input in a problem with the same language. As can be seen the variance stretches to negative time for all languages, i.e. in each language there were some cases where the large input ran faster than the smaller. This originates form the natural fluctuation that is present in all machines. To get more accurate values for runtime, the execution time should be measured several times so extreme values could be disregarded and then use the median runtime. The difference in user time for small and large input is similar to the user time. C and C++ have more concise values and Java still has the largest variance. Due to the large variance the results might not be a great indication of how well the languages scale with respect to input. However, C, C#, C++ and Python have medians significantly closer to 0 than compared to Java which have a median of 0.1 seconds.

Table 5.14 summarizes the rank of the languages for both wall-clock time and user time. As shown by the table the rank of the languages are the same regardless if the wall clock time or the user time is considered. In general the results for the different approaches align, however when calculating the mean time the ordering of the languages differ compared tot he other approaches. This is most likely due to the edge cases being very different from the majority of the data.

Table 5.14: Rank of languages for time, a * detonates a tie

Wall-clock time					
Median	C*	C++*	Python	C#	Java
Mean	C	C++	C#	Python	Java
Box Median	C*	C++*	Python	C#	Java
VD	C	C++	Python	C#	Java
User time					
Median	C*	C++*	Python	C#	Java
Mean	C	C++	C#	Python	Java
Box Median	C*	C++*	Python	C#	Java
VD	C	C++	Python	C#	Java

RQ5. Which programming languages use memory most efficiently?

To answer the question which language uses memory most efficiently we measured the maximum used RAM memory for the solutions that only execute without errors or timeout. The results for this analysis are displayed in figure 5.7 and in table 5.15.

C and C++ have manual memory allocation and have therefore small memory footprints. The variance between solutions in these languages are also small. On the other hand, for the languages having a garbage collector responsible for the memory handling, i.e. C#, Java and Python, the memory footprint is larger.

There is a clear difference between C#, Java and Python languages. Solutions written in Python have a small variance compared to C# and Java and the smallest median. Thus Python being the best option with automatic memory allocation. A possible explanation for why C# and Java have a larger memory footprint is the initialization of objects needed in those languages. It is possible to create objects Python as well but our impression is that it is not necessary, in most cases, given the characteristics of the problems in GCJ. Another explanation for the differences among theses three languages are the different implementations of the underlying garbage collector, e.g. which strategy used. Furthermore, Java uses significantly more memory than C#, a possible implication of the additional memory allocated by the JVM during start up.

The Vargha and Delaney statistics, whose output is presented in table 5.16, shows that solutions written in C have the smallest memory footprint. Java, in contrast, provides the largest memory footprint. These results align with the results from calculating the median and mean which can be seen in table 5.17.

Besides comparing the languages against each other, we compared the size of the memory footprint between the small and large input. The larger input differs from the smaller input by having either more test cases or larger magnitude for each of the provided test cases.

Beside the space complexity of the algorithm, the memory consumption, in this study, is affected by two factors. The first factor is how a contestants read the input. More precisely if a contestant is reading the entire file into memory or reading the file

Table 5.15: Maximum Memory Footprint, in bytes

Language	Min	Median	Mean	Max	Sum
C	1256	1412.00	2766.57	548688	7776828
C#	8824	11764.00	15672.81	945948	48225240
C++	1932	2556.00	4398.43	2066600	362897008
Java	21276	27784.00	54658.29	1418332	1072122408
Python	6228	6764.00	9492.29	2104756	242043872
Overall	1256	6764.00	17397.68	2104756	1733065356

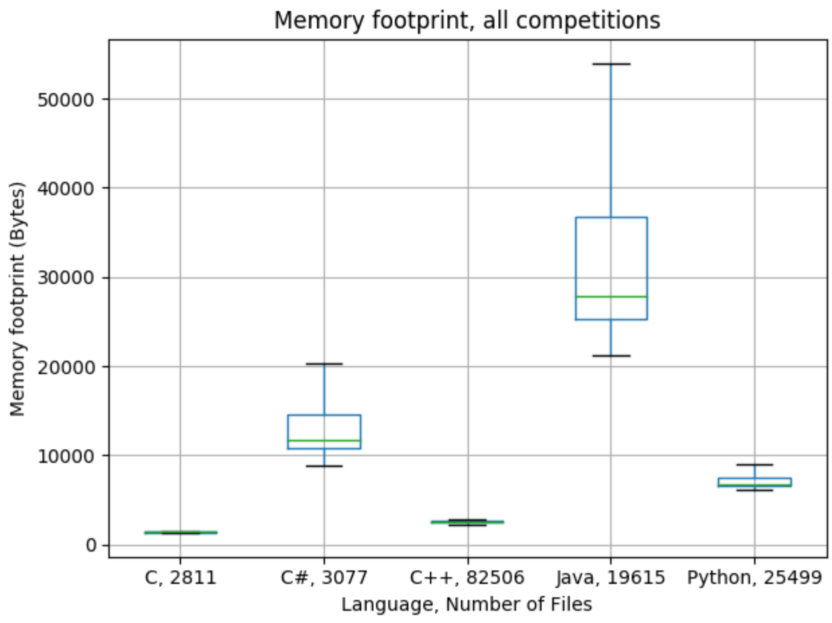


Figure 5.7: Box plot for memory footprint, in bytes

Table 5.16: Vargha and Delaney results for memory footprint

Language	C	C#	C++	Java
C#	0.01			
C++	0.04	0.98		
Java	0.01	0.04	0.01	
Python	0.02	0.93	0.03	0.98

line by line. The second factor is the magnitude of each test case. When the whole input file is read at once the amount of used memory depends on the size of the file. Large input requires more memory than the small input both when considering the magnitude and the number of test cases. In the case where the magnitude of each test case for the larger input is bigger, one could expect the larger input to use more memory since it may need more intermediate steps to solve the problem or require more memory for storing each test case.

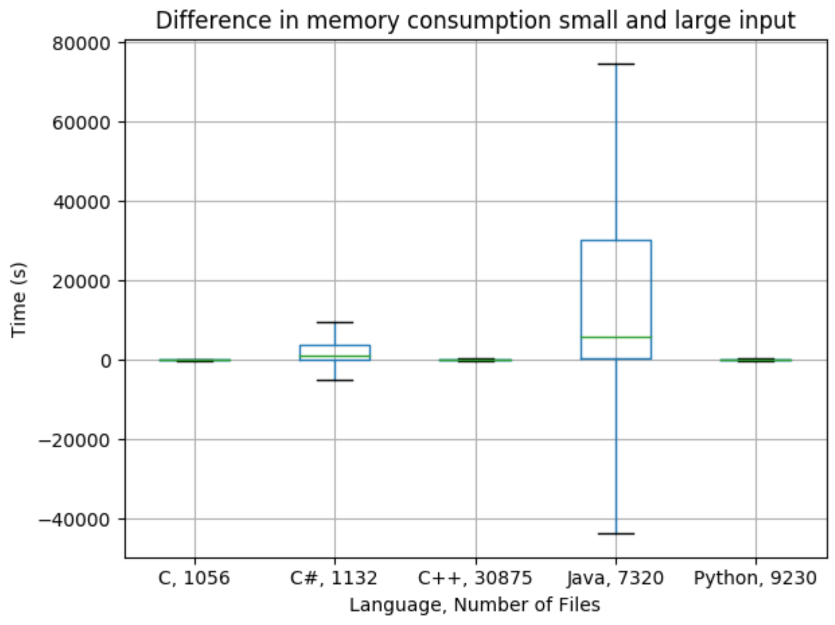


Figure 5.8: Box plot for difference in memory footprint between small and large input, measured in bytes

Figure 5.8 illustrates the difference in memory consumption between a contestants small and large solution. C, C++ and Python have no difference in size of memory footprint between the small and large input. For C# and Java solutions there exists solutions that use more memory for the large input as well as solutions that surprisingly uses less memory for the large input compared to the small one.

.

Table 5.17: Rank of languages for memory footprint

Median	C	C++	Python	C#	Java
Mean	C	C++	Python	C#	Java
Box Median	C	C++	Python	C#	Java
VD	C	C++	Python	C#	Java

Summarizing the findings for which languages uses memory most efficient, we see that all used statistical methods points towards the same result, see table 5.17. The is quite a clear difference between the languages, where C is the best option and Java the worst when RAM is limited. In general, memory footprint for each language has a small variance. Therefore different implementations of a solution in a language does not make a huge impact on the memory footprint of the program. Our findings for most efficient use of memory aligns with the results by Nanz and Furia [3] as well as the results from Prechelet's [4] study.

RQ6. Which programming languages are more self contained?

Besides analyzing performance features of the languages we also consider which of the languages are more self contained. For this analysis we excluded solutions that ran successfully. We identify which errors are most common for the investigated languages and what is the reason behind this. This is achieved by analyzing the error messages obtained under compilation or execution and the corresponding solution file. We also examine solutions that timed out to evaluate the set time limit, i.e. the time limit of 10 seconds.

Table 5.18: Number of solutions that compile and/or execute with errors

	Compile Errors		**Runtime Errors**	
Language	Nbr	%	Nbr	%
C	395	8	1869	39
C#	1406	20	3739	54
C++	15157	11	53513	39
Java	2663	8	13438	40
Python	406	1	11307	30
Total	20027	9	83866	38

Table 5.18 shows that Python tends to be the easiest language to compile. Such result is not surprising since Python is a dynamic language and therefore most of the errors occur at runtime. However, Python tends to have lowest percentage of runtime errors, suggesting that Python is the more self contained language among the investigated languages.

For solutions written in Python, Java and C# the most common error was exiting the program with status 1, having 8149 cases in Python, 10110 in Java and 2181 C#. Exit code 1 is used as catchall for general errors, e.g Syntax Error, File Not Found Error and Import Error. File Not Found Error was the most common error and accounted for 48% of all cases where exit code was 1 for these languages. This result is not surprising since most of solutions written in C and C++ expected the input provided as an argument or redirected and therefore contain less of such errors. Furthermore, this indicates that more development time should be spent on writing regular expressions to replace the hard coded input and/or output files. Other common errors with exit status 1 found in Java were `ArrayIndexOutOfBoundsException` and `NullPointerException` accounting for 1171 and 1260 cases respectively. While in Python the second and the third common errors, i.e. ValueError and Import Error, accounted for 1654 and 675 cases.

To give some concrete examples of cases that were not covered we examined a subset of solutions where automatic path replacement failed. The general pattern for Python is that we missed to patter match for '-' in the input file names, e.g `"ProblemA-large.in"`. Other cases involve adding an file ending to the input argument, which already contains the file ending, e.g `sys.argv[1] + '.in'` and abnormal paths, e.g `Users + // Downloads + // + input + '.in'`. Patterns found in Java are quite similar to those that were found in Python, e.g `Ç:\\\GCJ\\Large-a.in`. In addition to these patterns, we did not take into account cases as follows: `new File ("A-large(1).in"), new File(A.class.getName() + ".in")`. Many of the File not Found exceptions, in Java, were raised because of the missing output file since we did not spend as much time for writing regular expressions for replacing these paths. This was not a problem for Python since a file that is being written to is created automatically if it does not exists. To achieve automatic file creation in Java, in contrast, a developer must create a new File object; this was not the common procedure among the downloaded solutions. C# has the same issues as Java, i.e. in several cases the output path did not exist and thus caused the program to trow an exception. As a result of the following we succeeded to replace more paths in Python in contrast to Java and C#.

12844 of the 217684 downloaded solutions were forcefully terminated by our script because they ran longer than 10 seconds. We manually investigated a sample of 380 solutions to measure the actual running time for these languages; the output from this analysis is shown in table 5.19. The conclusion that can be drawn from the manual investigation is that time limit of 10 seconds was set too low for some problems. This is especially true for problems running with larger input; we found working solutions finishing after 6 minutes. Therefore the time limit should have been set to at least to 3 minutes to catch some of the slowest solutions. Furthermore, 207 of the examined solutions executed under 10 second when ran manually. These results suggests that the implemented time out, achieved by using threads, did not work as intended. Another possible explanation for this is that the time out followed from blocks by other process active on the virtual machine.

Table 5.19: Manual investigation of solutions that timed out where # >3 denotes number of solutions that took longer than 3 minutes to finish, # <3 denotes number of solutions that ran longer than 10 seconds but faster than 3 minutes. Other Errors denotes errors encountered other that File Not Found Errors

Language	# >3 minutes	# < 3 minutes	# Other Errors	# < 10 seconds
C	4	11	7	39
C#	3	9	3	10
C++	4	19	13	64
Java	4	27	2	37
Python	16	44	7	57
Total	31	110	32	207

6
Discussion

This chapter reflects on the quality of the downloaded data. We also evaluate design choices and implementations made in this empirical study and discuss potential improvements. Finally we present suggestions for future work.

6.1 Quality of the Downloaded Data

The data used in this study had more flaws than expected. In the following section we recognize some of the difficulties encountered with regard to the quality of the data and discuss how such difficulties may have influenced the result.

A considerable amount of the downloaded solutions were incomplete. These files contained only comments or an empty main method, just to mention some of the encountered problems. Problematically there was no simple way for us to filter out all solutions with incomplete code. Thus we choose to only include solutions which had successfully executed in the analysis to decrease the amount of inadequate solutions. However, this does not guarantee that all of the executed solutions solved the given problem or any problem at all for that matter and we recognize this issue as a potential threat to validity for our study. The simplest solution to this problem, from our point of view, would be that Google verified that the correct output is produced using the submitted code and not only verifying the submitted output file.

The aspect that data originating from the GCJ repository or analogous competitions is partly inadequate should be kept in mind for similar future studies. Clearly, a contestant that got accepted yet submitted inadequate source code must have solved the problem; however did not submit the actual code. Therefore when using data that have been developed independently from the study it is important to study the specifications for submission of the developed code. Specifications to consider are for example: handling of input and output, allowance of third party libraries and complexity of implemented algorithm.

Given the structure of the problems in GCJ, it is hard to generalize our findings to real world projects, since all problems in GCJ have similar characteristics and size. All problems test knowledge of algorithms and data structures, therefore resulting in fairly short programs. In real world projects similar code can be found in logical components of the software. However the size of these components are, in general, larger since they serve as logic for more complex problems.

6.2 Design Choices and Implementation

The environment used to perform all executions was the virtual machine manager Qemu. This was the environment provided by our university which made Qemu a natural choice for us to use. The reliability of Qemu was low. The virtual machine entered the `READ-ONLY` state consistently, was suddenly killed or lost the Internet connection. As a result of this the downloading, compilation and execution phase took a significantly longer time than planned. We have no explanation for the situations where the virtual machine lost the Internet connection or was suddenly killed. However for the first issue that is the `READ-ONLY` error we have two possible explanations. The first one being related to the number of threads. When executing solutions via Python's `subprocess` module, new threads are started. Then the executed program generally starts additional threads itself. When several threads run simultaneously and are not terminated correctly, the RAM is eventually exceeded causing a corrupt file system which finally puts the virtual machine in the `READ-ONLY` state. The other theory is that hardware errors occasionally occurs which the virtual machine cannot recover from and thus enters the `READ-ONLY` state.

Python was used for the implementation of the analysis scripts. This choice of language facilitated the process of automatically compiling and executing solutions. Python has several convenient libraries that we made use of, such as `Pandas`, `subprocess`, `multiprocessor`, `re` and `urllib2`. Furthermore, it would have been advantageously to use the Python 3.x branch because of the provided timeout parameter that could be set when launching a new process, instead of writing our own implementation. However when too large sets of data were to be compiled the we had difficulties with our environment Qemu, as mentioned above. We consider Python a good choice of language when doing a study such as this one, assuming another choice of virtual machine.

We wanted as many downloaded solutions to execute as possible. We fixed simple errors described in section 4.4, where most errors were regarding handling of files for input and output. The different ways contestants handled input and output affected the study in several ways. Firstly it influenced how many solutions we were able to compile and execute, since not all issues regarding input and output handling could be patched. Secondly it influenced the time performance. The fact that contestants have handled their input and output using different approaches was discovered early in the exploratory phase. However, that it would affect our measurements significantly was something we only realized towards the very end of the study. Therefore our findings with regard to time performance should be taken with a grain of salt. Furthermore, when performing time measurements each solution was only executed once. To get more reliable measurements each solution should have been executed a repeated number of times and the average of these measurements should be used instead. This especially affects languages that compile into bytecode. The first execution of these languages comes with one-time overhead due to the need for the virtual machine to load from the disk.

Another parameter that influenced the number of solutions that we were able to execute was the timeout. Our strategy for choosing the time limit before a solution got forcefully terminated was based on our findings during the exploratory study. Studying the execution times of the programs downloaded in the exploratory study, we found the slowest programs do not use more then 4 seconds, in general, to terminate thus 10 seconds is in most cases a reasonable time limit. To increase the amount of solutions that execute correctly a possible option would be to set the timeout to 4 and 8 minutes for the small and large input respectively. Thus having the same time limits used in GCJ. Using a timeout of 4 and 8 minutes was not an option due to the resulting excessive time requirements.

Considering the languages chosen to evaluate during this study we are satisfied with our choice. These languages are both the most popular in the competition and in real life software projects. It would have been interesting to include additional languages in this comparative study, for example JavaScript and Ruby. The amount of solutions in these languages however is just a fraction of the five considered languages, and therefore lead to the findings being hard to generalize.

6.3 Future Work

The classification of which paradigm a language belongs to has become more vague in recent years. The reason for this is the new features that are being added each year. Because of the size of the GCJ repository it is suitable to use for evaluating how well the new features are used in practice. An example of such study could be to compare versions of Java, more precisely comparing Java 1.7 to Java 1.8. Another suggestion is to compare solutions written in Python 2.x and Python 3.x.

Considering optimization in runtime, memory and size of executable investigating how flags and different compilers influence these parameters could be an interesting topic for the future. Using a new compiler and adding flags to the compilation would benefit programmers, thus there would be no changes to the existing code base and performance would increase if such evidence could be found.

7 Conclusions

To give a concrete answer to which programming languages are superior is a difficult task. The answer depends on the angle the question is asked from and which parameters are considered the most important. Generalizing and combining our findings from our research questions there are no language which is superior to other languages taking into account the investigated features. To summarize our findings, one has to choose between speed and size of executable.

C, C++ and Python are the top contenders for best languages according to our results, while Java and C# performed worse. Programs in C have a small memory footprint, run fast and can be written concisely. On the other hand, programs written in C have large size of the executable file. C++ has similar characteristics as C but tends to be less concise. Programs written in Python can be written very concisely and have small executable files. With respect to time performance and memory consumption, Python is the middle language. However, this does not mean that Python solutions ran slow. Python has the highest percentage of successful executions thus more solutions were included and there were a larger variance in run time for Python solutions.

For applications where time and memory consumption are the most important aspects to consider, C or C++ is the better choice. However, for programming completions, such as GCJ, Python emerges as a reasonable alternative to these languages. Java and C# produce smaller executables and therefore can be used with benefit on storage with limited space. The results from this study can not be applied to real world projects, though can provide some guidness when choosing a language for a software project.

Bibliography

[1] R. Meyerovich, "Emperical Analysis of Programming Language Adoption," *OOPSLA*, pp. 1–18, 2013.

[2] "Google Code Jam History," https://code.google.com/codejam/archive.html, accessed: 2016-12-10.

[3] S. Nanz and C. A. Furia, "A comparative study of programming languages in Rosetta Code," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 2015, pp. 778–788.

[4] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[5] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[6] "A large scale study of programming languages and code quality in."

[7] "Quick-start Guide," https://code.google.com/codejam/resources/quickstart-guide, accessed: 2017-04-14.

[8] "Tiobe Index for January 2017," http://www.tiobe.com/tiobe-index/, accessed: 2017-01-23.

[9] "Language statistics," https://www.go-hero.net/jam/16/languages, accessed: 2017-01-23.

[10] "alexandraback/datacollection," https://github.com/alexandraback/datacollection, accessed: 2017-03-16.

[11] "emmawestman/gcj-backup," https://github.com/emmawestman/GCJ-backup, accessed: 2017-03-16.

[12] "Aldanial/cloc," https://github.com/AlDanial/cloc, accessed: 2017-03-16.

[13] "stdc++.h," https://gist.github.com/eduarc/6022859, accessed: 2017-03-16.

A
Flow Charts Describing Error Handling

Understanding how the scripts deals with errors and how they tries to fix them has been challenging to explain in a simple and clear way. Therefore we have drawn flow charts to illustrate the sequence of actions the scrips takes when encountering a compilation or execution error. In this Appendix A, flow charts describing the error handling for C#, C++, Java and Python can be found.

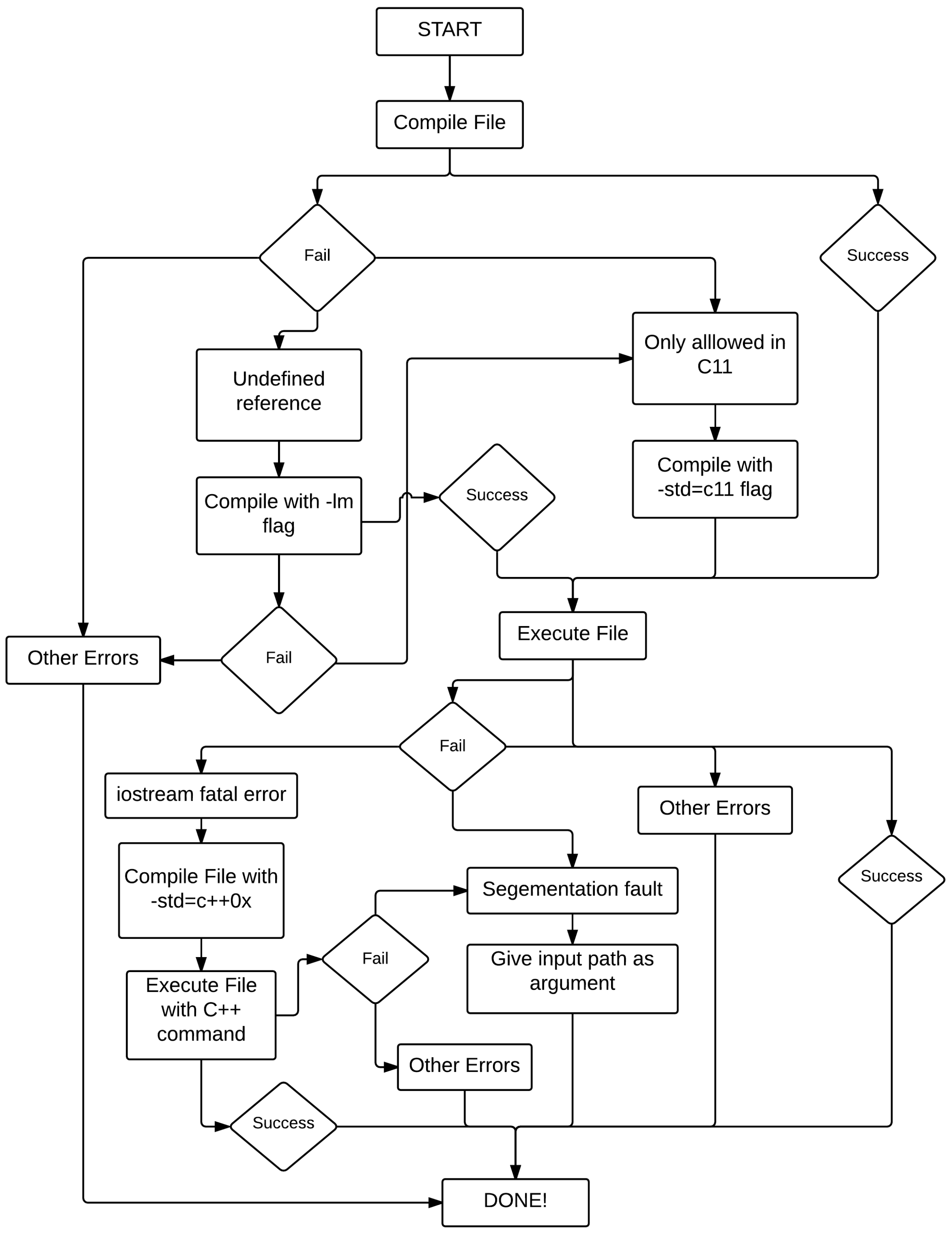


Figure A.1: Flow chart describing error handling for C solutions.

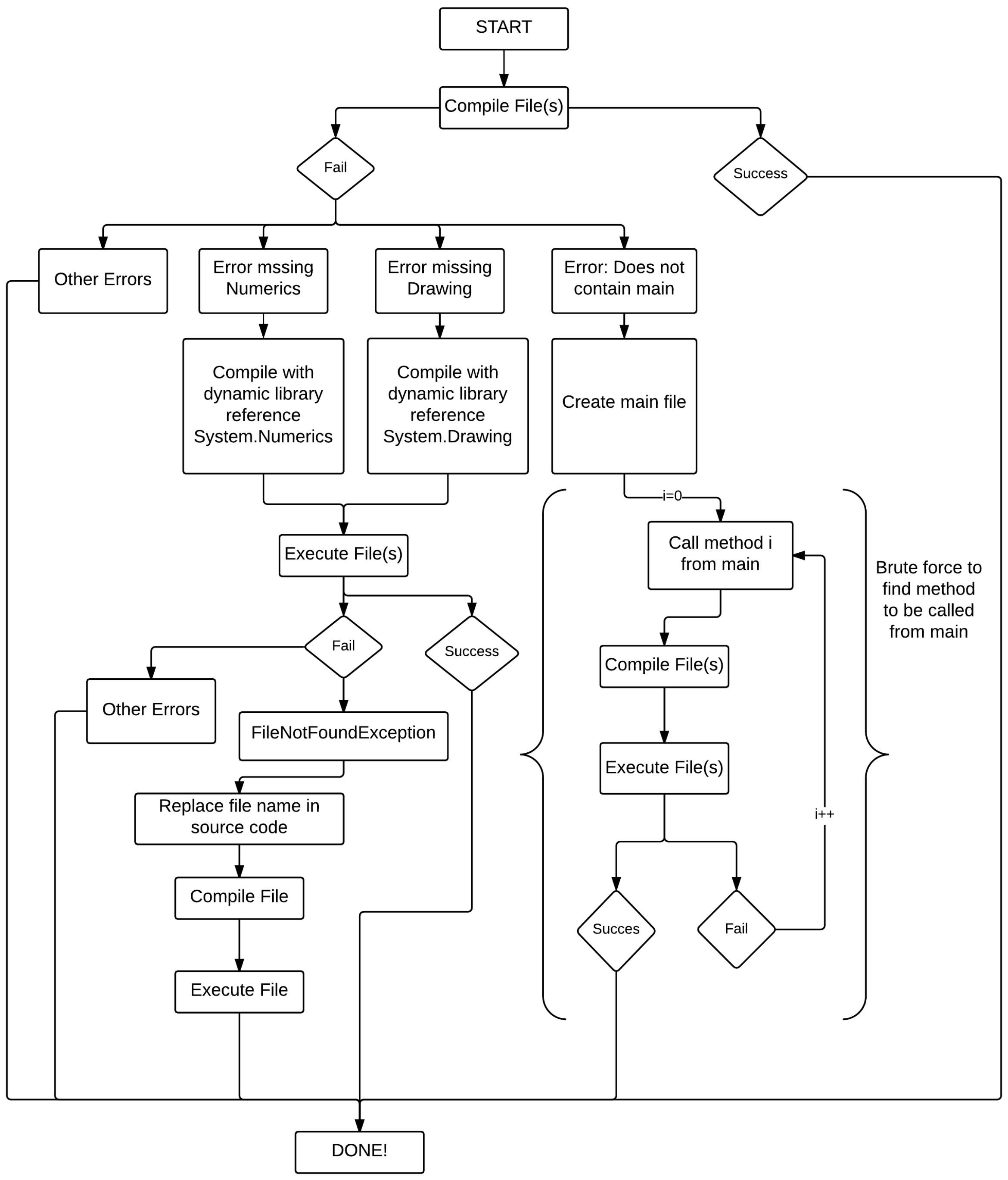


Figure A.2: Flow chart describing error handling for C# solutions.

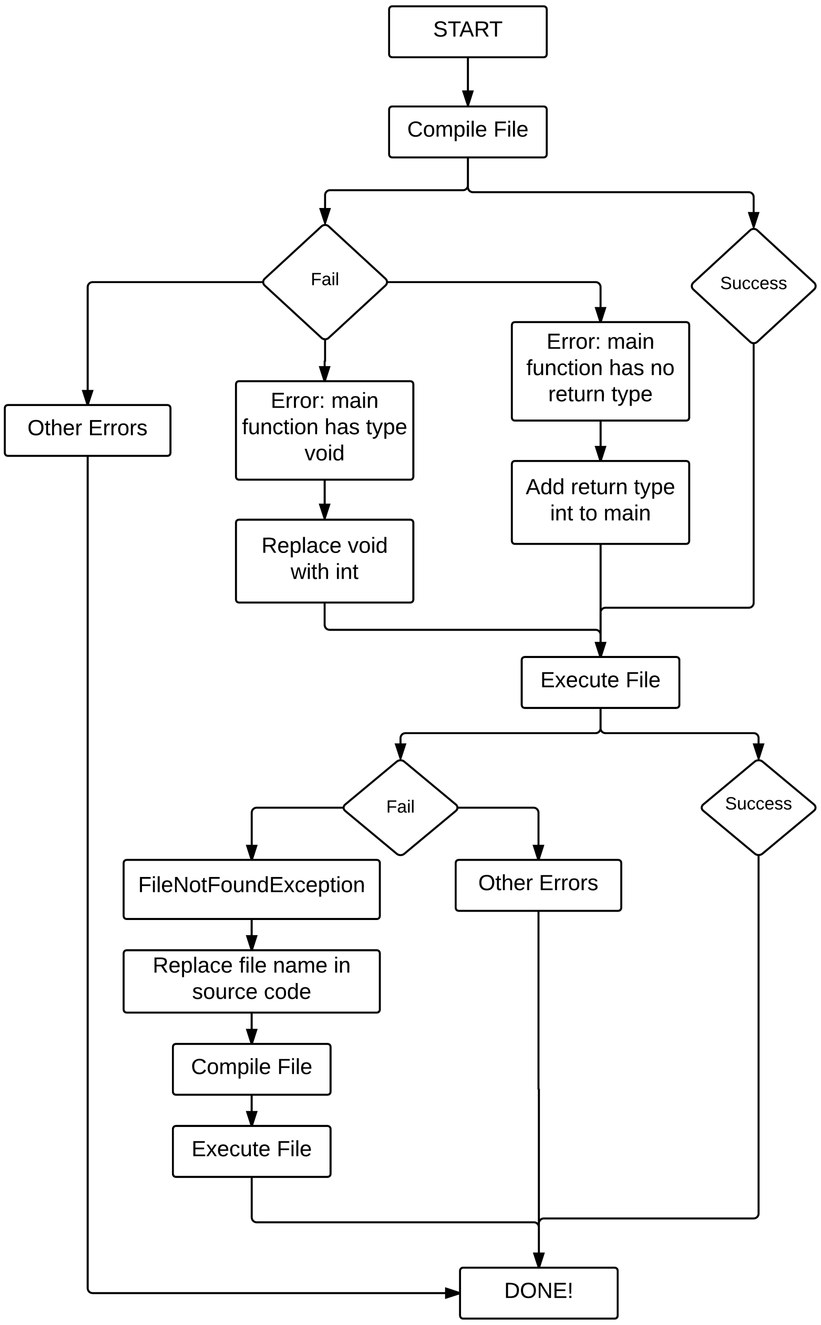


Figure A.3: Flow chart describing error handling for C++ solutions.

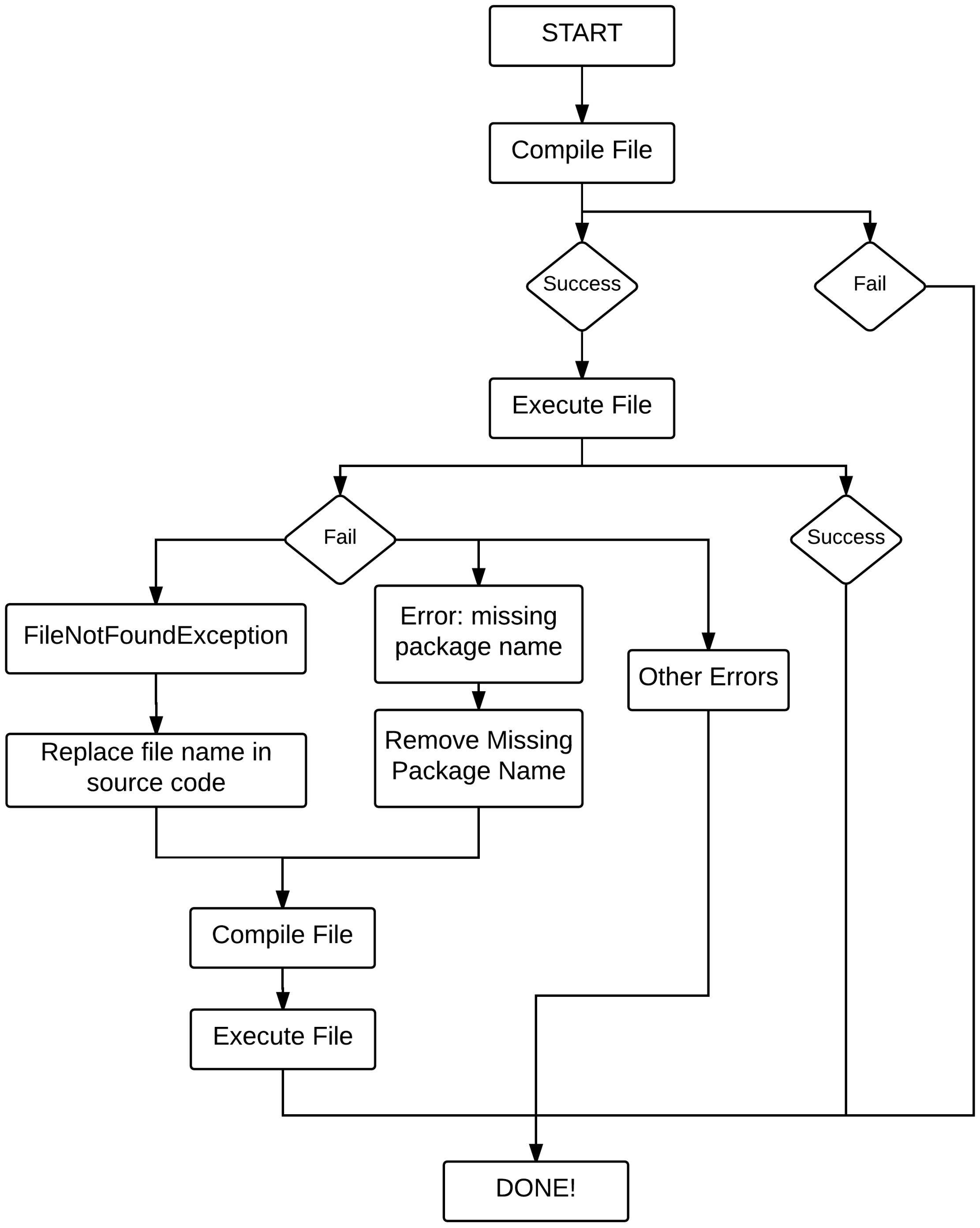


Figure A.4: Flow chart describing error handling for Java solutions.

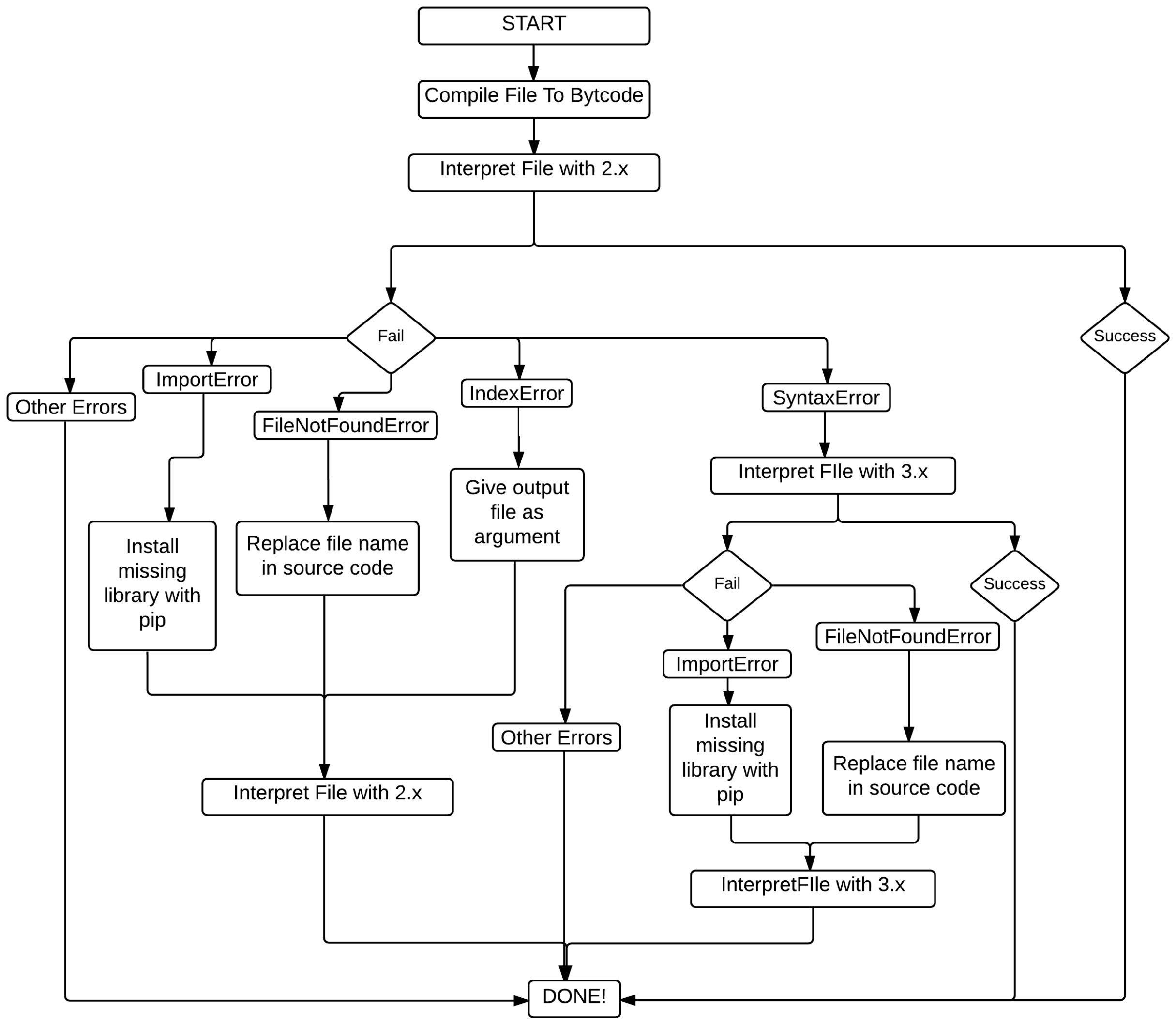


Figure A.5: Flow chart describing error handling for Python solutions.

B
Additional Statistics

Table B.1: Matrix of Consistency of Mean Rank

Contest Name	Qualification Round 2016	Round 1A 2016	Round 1B 2016	Round 1C 2016	Qualification Round 2015	Round 1A 2015	Round 1B 2015	Round 1C 2015	Qualification Round 2014	Round 1A 2014	Round 1B 2014	Round 1C 2014	Qualification Round 2013	Round 1A 2013	Round 1B 2013	Round 1C 2013	Qualification Round 2012	Round 1A 2012	Round 1B 2012	Round 1C 2012
Qualification Round 2016	1.00	0.60	0.80	0.75	0.43	0.25	0.56	0.60	0.56	0.60	0.80	0.60	0.43	0.43	0.71	0.60	0.56	0.56	-1.00	0.43
Round 1A 2016		1.00	0.33	0.14	0.50	0.50	0.40	0.20	0.40	0.60	0.33	0.33	0.50	0.25	0.60	0.60	0.60	0.60	0.75	0.50
Round 1B 2016			1.00	0.80	0.75	0.00	0.71	0.80	0.71	0.50	1.00	0.50	0.75	0.80	0.56	0.50	0.80	0.60	0.20	0.75
Round 1C 2016				1.00	0.80	-0.33	0.60	0.75	0.60	0.43	0.80	0.60	0.80	0.75	0.82	0.43	0.60	0.71	0.60	0.80
Qualification Round 2015					1.00	0.56	0.60	0.50	0.60	0.80	0.60	0.60	1.00	0.80	0.50	0.80	0.80	0.80	0.50	1.00
Round 1A 2015						1.00	0.50	0.33	0.50	0.60	0.25	0.56	0.56	0.25	0.33	0.60	0.56	0.78	0.14	0.56
Round 1B 2015							1.00	0.78	1.00	0.60	0.60	0.80	0.60	0.80	0.80	0.60	0.50	0.80	0.40	0.60
Round 1C 2015								1.00	0.78	0.75	0.80	0.56	0.50	0.60	0.56	0.75	0.60	0.82	0.20	0.50
Qualification Round 2014									1.00	0.60	0.60	0.80	0.60	0.80	0.80	0.60	0.50	0.80	0.40	0.60
Round 1A 2014										1.00	0.50	0.56	0.80	0.56	0.60	1.00	0.75	0.60	0.33	0.80
Round 1B 2014											1.00	0.50	0.75	0.80	0.56	0.50	0.80	0.60	0.20	0.75
Round 1C 2014												1.00	0.40	0.60	0.75	0.56	0.60	0.56	0.20	0.40
Qualification Round 2013													1.00	0.80	0.50	0.80	0.80	0.80	0.50	1.00
Round 1A 2013														1.00	0.60	0.82	0.75	0.60	0.25	0.80
Round 1B 2013															1.00	0.71	0.43	0.60	0.60	0.50
Round 1C 2013																1.00	0.75	0.60	0.33	0.80
Qualification Round 2012																	1.00	0.75	0.00	0.80
Round 1A 2012																		1.00	0.60	0.80
Round 1B 2012																			1.00	0.43
Round 1C 2012																				1.00