Time Complexity Analysis of a Simple Code: Using Graph Plotting and Pattern Based Approach

by

Lamia Nurtaj
23141057
Md. Tanvirul Hasan Rafi
20301009
Al Zabeir Bin Younus
20301119

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
November 2024

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Lamia Nurtaj

23141057

Md. Tanvirul Hasan Rafi

20301009

Al Zabeir Bin Younus

20301119

Approval

The thesis/project titled "Time Complexity Analysis of a Simple Code: Using Graph Plotting and Pattern Based Approach" submitted by

1. Lamia Nurtaj (23141057)
2. Md. Tanvirul Hasan Rafi (20301009)
3. Al Zabeir Bin Younus (20301119)

Of Summer, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on November 13, 2024.

Examining Committee:

Supervisor:
(Member)

Md Imran Bin Azad

Senior Lecturer
Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD

Associate Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi, PhD

Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Abstract

Computation is a core topic in computer science. We code in Java, python, c++ or any other programming language for the backend and frontend of computers. To accomplish a task, a software system applies a collection of interconnected algorithms[6]. Most of the time, we measure the run time of a program using only our visual calculations. Our research shows that this visualization could be represented graphically and compute time complexity using a pattern based approach. Our research can find an infinite loop in some cases. Calculation can be represented in a complexity graph and analyze the time complexity. This study will help to precalculate before program execution, save running time, reduce computation, and create a new viewpoint in coding. The immediate goal is to visualize complexity in a programming language and is this possible to detect infinite loops from code. Another challenge for programmers often arises is to determine and detect infinite loops. Moreover, we can also calculate the complexity of recursive calls. However, in this challenging segment, there are two things that need to be addressed, i.e, space and time complexity. This is dependent on depth, number of calls and many other factors.

Keywords: Computation; Programming; Control statement; Control variable; Time complexity; Infinite loops; Equation; Main scope;; LoopVariable; FunctionVariable; Function call; complexity measuring tool;

Table of Contents

Chapter 1

Introduction

"""
dear yyy,
I was doing a programme after an hour of unsaved work, and the programme ran for a long time. My programme is totally frozen and after three working days I am unable to find the cause. Could you help me with this matter??
Yours truely,
xxx
"""

As the above email illustrates, some specific errors can make an application unusable for its users. This programme clash can include loss of work, loss of time and energy, or being unable to use the application for the intended purpose. One solution can be to drop the application into a debugger, find the error , and enable the application to continue its productive execution. Unfortunately, not everyone has that much technical expertise to find errors and use solutions properly, and it is also time consuming. Even if one does , finding the clash and encountering the programme execution is an annoying process.

1.1 Motivation

The effectiveness of a code lies under the aspect of assessment of time complexity concept. However, computational operations or any program of computer is bypassed as conventionally these kind of computations are done by hand. In our investigation we have identified how gauge report is complicated when a user provided code is given. However, the expectation is the user will provide their code in a form of string. Therefore, the conduct for interpretation of the given string is the basis of our research. The profound objective of this research is rigorous analysis of the given code that was initially set forth. Determining the loop and recursive function thoroughly and systematically how coordinated code can be executed. Moreover, our working consists of giving automated instructions that will eventually optimise the code to improve the efficiency. From the insights obtained from time complexity that was applied, we targeted on to optimisation that can peak the performance. Moreover, we worked on to reduce the gap which occurs in the context of time complexity analysis and execution of code. Our desired result is to create an efficient automated code which will enable effectuality and accuracy by rigorous examination

and analysis. Advancement of software engineering is the purpose of this research through deploying effective tools and approach for improving the effectiveness and efficacy of their program.

1.2 Problem Statement

Code analysis is very important when we have to do a project or any purpose. The things that we analyze in the code are given below,

1. Running time of loops.
2. Loop measurement.
3. Minimize the complexity of loops.
4. Extra-allocating memory location.
5. Infinite execution in a program.

Our motivation is to create this code in an equation that will help us visualize more effectively.

1.3 Research Objective

To conduct our research, we will first examine programming code to create a general form and then evaluate the equation derived from this form. Subsequently, we will examine differences in graphs formed for various equation types to gain insight. We will also investigate the relevance of our research topic to learning methods, assessing whether it enhances understanding. Additionally, we will explore how an infinite loop might relate to detecting our research question. Our research will identify limitations, and we will attempt to resolve them to improve outcomes. Furthermore, we will analyze equations to develop a custom dataset for our methodology, aiming to establish the order of priority among equations.

Chapter 2

Related Work

We studied the existing body of literature and research that is currently available that is directly related to the topic of our study in this section. The objectives of this thorough analysis are to set the scene, define the current level of knowledge, and highlight significant discoveries and developments in the subject. We provide a synthesis of previous work that informs and analyses our research objectives through a study of relevant research papers and scientific contributions. We establish the groundwork for our examination and pinpoint areas that require more research in the field by critically analyzing earlier works.

Firstly, we reviewed the paper authored by Chen et al.[3] focuses on theoretical studies of Evolutionary Algorithms (EAs), particularly their time complexity. It draws attention to how crucial it is to comprehend time complexity in EAs and points out that there isn't a universally applicable method for evaluating population-based EAs on unimodal situations. The paper discusses several earlier methods for researching EAs, such as Chernoff bounds, analytical Markov chains, and drift analysis. The study notes previous research on the use of EAs in unimodal situations, citing several sources and emphasizing that the majority of these studies are case-specific and use various techniques. The contribution of the paper is put in motion by the literature review, which outlines an unusual general method for analysing the time complexity of population-based EAs in unimodal situations. It highlights the shortcomings of previous research, which frequently concentrates on particular situations or employs methodologies that aren't generally applicable, and explains why such an approach is necessary. The goal of the paper is to provide a thorough examination of EAs in solving unimodal problems by generalizing the idea of takeover time in EAs and applying it in conjunction with Markov chain models and supermartingale models. This method is positioned as a major development in the theoretical understanding of EAs, especially with regard to how well-suited they are to tackle particular kinds of problems. Then, We reviewed the paper written by Pietro S. Oliveto and Carsten Witt [9] focuses on the time complexity analysis of Genetic Algorithms (GAs), specifically the Simple Genetic Algorithm (SGA). It considers the difficulties and developments in utilizing crossover and mutation in conjunction with random selection to analyze the time complexity of GAs. The review highlights the development of theoretical ideas and procedures by citing important prior publications in this field. The limitations of past research, particularly about the population size and variety within these algorithms, are among the important top-

ics covered. The previous investigations, according to the scientists, were limited to smaller population sizes and necessitated restrictions on the population's bandwidth. Additionally, they discuss the use of certain mathematical methods, such as fitness levels for non-elitist populations and negative drift theorems, highlighting their benefits and drawbacks when it comes to crossover-based algorithms. The paper also goes into detail on how approaches for runtime analysis of population-based evolutionary algorithms have evolved over time, emphasizing the necessity for more thorough and adaptable methods. This requirement results from the limits of elitist selection operators frequently employed in previous studies as well as the complexity added by crossover operators. The paper's contribution, an enhanced runtime analysis of the SGA, is predicated on the literature review. To get around the restrictions of earlier research, the authors suggest a novel strategy that does not rely on the bandwidth limit and permits greater population sizes and a more complex comprehension of population variety. This development is considered a major step towards understanding the influence of crossover and selection mechanisms in these algorithms, as well as towards more systematic and generalizable evaluations of GAs. Overall, the review does an excellent job of placing the work in the larger framework of evolutionary algorithm research while emphasizing the difficulties that still remain and the demand for improved analytical methods in this area.

We reviewed the paper written by Z. Nopiah, M. Khairir, S. Abdullah, M. Baharin, and A. Arifin,[4] which provides a comprehensive overview of genetic algorithms (GAs) and their applications, particularly in clustering methods. It describes the common features of GAs, including chromosomal populations, fitness-based selection, crossover, and mutation. The review also covers the representation of solutions in GAs, fitness-based selection, and other selection techniques such as rank, roulette wheel, and tournament selection. It also discusses the crossover and mutation processes in GAs, how they produce new solutions, and when to end the GA process. The review lays the groundwork for the paper's analysis of GA's temporal complexity in clustering, with a particular emphasis on how population constraints affect the algorithm's performance. The literature review of the paper written by T. Dobravec [10] delves into the challenges of accurately determining the practical time complexity of algorithm implementations. It focuses on the drawbacks of theoretical time complexity analysis, which frequently depends on presumptions that don't accurately represent actual circumstances. The impact of memory management, particularly cache use, and environmental factors on algorithm speed are some of the concerns that are highlighted in the study. The paper also discusses the challenge of setting up a consistent and equitable framework for evaluating algorithm performance in real-world scenarios. Going beyond conventional theoretical models, the author presents a novel method for estimating algorithm complexity: counting Java bytecode instructions. By examining real bytecode instruction usage, this approach seeks to provide a more accurate picture of temporal complexity while acknowledging the impact of low-level operations on algorithm performance. The literature review includes the research paper authored by R. Michael Tanner[1] and published in the IEEE Transactions on Information Theory, this paper delves into the realm of low-complexity codes, a topic of significant importance in the fields of information theory and communications. In this study, a new method for creating low-complexity codes with bipartite graphs and recursive techniques is presented.

An overview of low-complexity codes and their importance in information theory is given at the outset of the study. After that, it discusses creating new codes using bipartite graphs and subcodes, offering an original viewpoint on the topic. In addition, a thorough examination of these codes' computational characteristics is included in the paper, emphasizing their usefulness. It goes on to talk about how to employ soft decision information effectively and how these codes' low computational complexity and parallelism can help them decode big blocks quickly. This paper illustrates many coding and graph theories using a variety of mathematical expressions and graphics. Furthermore, it offers a thorough explanation of the probability boundaries and the Salzberg Correction Procedure. The last section of the paper discusses decoding complexity and how a decoding process can be broken down into several subcode operations that only need local information. This paper offers insightful information about low-complexity codes and presents a distinctive method for developing and utilizing them. It contributes significantly to the understanding of coding theory and has potential implications for the development of more efficient communication systems.

The paper authored by Y. S. Patel, N. K. Singh, and L. K. Vashishtha [8] shows that informations are arranged in special orders like numerical and lexicographical orders in this modern age. Also here, the paper emphasizes many parts like computational complexity, number of swaps, stability, recursive nature etc. Measuring computational complexity as worst, average and best using some standard. For the worst case scenario, the sorting algorithm here is $O(n^2)$. For the average case, the sorting algorithm time complexity is $O(n \log n)$ and for the best case, the algorithm is $O(n)$. However, the ideal sorting algorithm in all the cases takes $O(nlogn)$ time. If any algorithm is comparison based then it is using any comparison operator to compare two elements. It is called non-comparison based otherwise. For in-place algorithms such as Quick sort number of swaps is used in order to compare and to sort in a single array. To check the stability we see the output. Such that the relative order of the same elements is in the correct order of sorted elements. In some algorithms we see non-recursive nature, and in some case, we see recursive nature. The sorting procedure of any sorting algorithm is usually measured for the space taken by the sorting algorithm. For ideal sorting algorithm it is $O(1)$ or for any other size of problems it is linear spaced. However there are other sorting algorithms like insertion sort, quick sort, merge sort, bubble sort, heap sort. Firstly let us understand Divide and Conquer approach. It is simply dividing the main problem into two or more smaller parts as subproblems till it cannot be divided. Now solving all the problems separately. Now combining the sub problems to get the solution of the original problems. It is used for mostly for the larger input size problems. Now for the proposed Fuse Sort Algorithm it is based on Divide and Conquer approach. It divides the n number of problems in $\sqrt{n}$ parts. Also every part has $\sqrt{n}$ size. This process continues until the sub-part minimum size is 2. After dividing the problems into $\sqrt{n}$ halves having size $\sqrt{n}$. It helps to solve the problem with the recursion method. Lastly, all the subparts are connected to form the solution of the whole. For the efficiency, it outperforms all other approaches. Fuse sort will not only perform computations in less time but will also consume linear space to save excess memory location.

From the paper written by J. N. Knight and M.Lunacek[2] proposes a modification of CMA-ES which aid to reduce the time and space complexity. Therefore, by using m<n we can increase computational efficiency which ultimately leads to a lesser runtime. However, complexity reduction ultimately decreases the data efficiency. Computational complexity and data efficiency can be controlled by alterable parameters. However, the overall complexity of the algorithm can be improved for the parameters that are problem dependent. Even so, there is yet to calculate the optimal setting of the parameter. However, values of m and n can be good when we use the functions with strong coupling between parameters. On the contrary, functions like Rosenbrock's function efficiency and performance can be developed as they are weakly coupled. When m=n the algorithm becomes the CMA-ES formulation. For m=0 the algorithm is step-size adaptation. However, L-CMA-ES helps to load the void between MVA-ES and CMA-ES as the number to alteration variety is from one to n. Then we reviewed the paper authored by Wang Xiang [7] focuses on the significance and efficiency of the Quick Sort algorithm in data processing systems. It begins by outlining Quick Sort's great efficiency and quick speed, which make it popular across a range of systems. It begins by outlining Quick Sort's great efficiency and quick speed, which make it popular across a range of systems. The review addresses the algorithm's basic structure and scientific design, highlighting the need for further investigation, especially in light of its temporal complexity. The paper describes various sorting algorithms, including Quick Sort, which is among the fastest, that are commonly used in computer encryption, company finance, data processing, and information searches. The partition operation and the choice of a pivot element—two fundamental Quick Sort concepts—are described. In comparison to alternative sorting techniques, including bubble sorting, selection sorting, and insertion sorting, the review also looks at the fundamental factors that contribute to Quick Sort's quick performance. The review also discusses how to calculate Quick Sort's temporal complexity. It goes into detail about the partitioning procedure and how pivot selection affects the algorithm's performance. The worst-case algorithmic scenario—where the pivot pick is always the smallest or largest element—resulting in an $O(n^2)$ time complexity is highlighted in the study, as is the best-case scenario, which has an $O(nlogn)$ time complexity. Additionally, the literature review places the Quick Sort algorithm in the broader context of algorithm analysis and highlights its importance for comprehending how sorting algorithms work in real-world scenarios. It prepares the reader for the paper's theoretical examination of Quick Sort's temporal complexity, which attempts to offer more precise estimates and insights into the system's effectiveness. The paper " Badger: Complexity Analysis with Fuzzing and Symbolic Execution" - gave a promising result in achieving high code coverage, error control, and getting out of vulnerabilities. Here Badger has been discussed which is a new approach for complexity analysis in the case of time and space.[12] It is when the worst-case complexity is significantly higher for a code compared to average cases. It uses diverse ways of input, which allows the coverage to increase as resource-related costs are associated with each path. However, fuzzing might fails to execute deep programming as it has limited knowledge about the conditions which essentially influence the paths. Here symbolic execution has been used that is evenly customised for efficiency. However, symbolic execution is particularly good at generating inputs that satisfy various program conditions; nonetheless, it suffers self-exploitation. Thus, Badger uses fuzzing to benefit them-

selves and overcome the problem by leveraging its usefulness. Here, a Java program has been used. After observing the success of fuzz testing, an algorithmic analysis to determine the complexity has been discovered. Essentially, the algorithmic complexity of a program has some advantages as it has the power to enable reasoning, determine the lack of performance, and use the compiler for optimum utilization. Nonetheless, it can also help to expose the worst-case time complexity. This can occur when the time complexity is higher than the average ones. In this scenario, the denial of service can provide input for the worst-case scenario. Nonetheless, it can help detect shallow bugs that cause errors in the execution of the code. Then we reviewed the paper authored by Wang Xiang[11] focuses on the significance and efficiency of the Quick Sort algorithm in data processing systems. It begins by outlining Quick Sort's great efficiency and quick speed, which make it popular across a range of systems. The execution time of the code needed can be determined by giving a rigorous model of the theme so that the proof can be secured. However, this tidy process gives results regarding the complexity class of the execution model. ICC, referred to as Implicit Computational Complex has convenience for this purpose as it makes a correlation between complexity and programming language. Thus it does not have to rely on an execution model or any explicit execution steps. To prove that a function is computable in polynomial time Coq library can be a great tool. Quasi-Interpretation and completeness theorem is the main core of this library. To aid in proving hypotheses of the soundness theorem on a given program. This tool for writing programs and computation of polynomial-time computability can be used by providing various decision procedures. Coq proof is another aid to provide ways to find running time bounds also functional correctness. By applying a little but non-trivial program, In this scenario, we intensify the usability of this equipment. In this case, modular exponentiation is widely used as they are mostly used for cryptography. Here in this paper, Quasi-Interpretation has been widely discussed. The essence here is that the program is in order according to PPO. Also, how the interface module can be made has been discussed here. The Interface Module consists of two parts: Module type and functors.

Lastly, we reviewed the paper written by S. Gayathri Devi, K. Selvam, and Dr. S. P. Rajagopalan,[5] presented at the Second International Conference on Sustainable Energy and Intelligent System (SEISCON 2011), delves into the realm of algorithm efficiency. First, the time and space complexity of algorithms are measured with Big-O notation, a mathematical tool that is commonly used to assess algorithm efficiency. Setting the scene, the introduction defines Big-O notation and its formal mathematical interpretation, emphasizing its use in mathematics and computer science, especially in algorithm analysis. The mathematical foundation part goes into more detail on Big-O notation, highlighting its use in comparing the scalability of various algorithms and characterizing functions based on their growth rates. The study provides useful code samples to explain several complexity classes, such as $O(1), O(n), O(n^2), andO(2^n)$, and shows how these complexities appear in actual situations. It specifically looks at how Big-O notation affects algorithm performance, especially in worst-case situations, and how it affects the use of computational resources. The authors also explored a sophisticated use of Big-O notation in the optimization of an algorithm for facial and fingerprint image identification. The part on performance evaluation explores the analytical model developed for this

purpose, looking at the probability distributions and total number of implicants in logic functions. The importance of Big-O notation in software project management and algorithm performance testing is emphasized in the study. It emphasizes the importance of Big-O notation as a metric in software development and application automation, highlighting how comprehending it may help optimize algorithms for improved performance. The paper advances computer science and algorithm analysis by presenting Big-O notation as a vital tool for mathematicians and engineers to evaluate and optimize algorithm efficiency.

Chapter 3

Methodology

Computation is a core topic in computer science. We code in Java, python, c++ or any other programming language for the backend and frontend in the computer code editor. To accomplish a task, a software system applies a collection of interconnected algorithms. Most of the time, we measure the run time of a program using only our visual calculations. Our research shows that this visualization could be represented in a graphical manner and compute time complexity using a pattern based approach. An equation can be represented in a graph and analyze the time complexity of a code. Firstly, the objective of the initial part of our research is to determine the possibility of using any machine learning or rule-based approach so that we can discover any pattern in the user code and the ideal code. Now for the source code that is written in C++, the declarations, classes, and logics are stored in a .cpp file. In our deep observations, we figured out that around 60 to 70 percent of the code is fixed in gcc which supports C++. However, machine learning was one of our approaches where we were supposed to calculate time complexity using a learning based approach.Our goal is to create extensions in the compiler. However, we need to make a tool where code can be filtered, organized , tokenized and make it simple for calculation. Therefore, we represent a complexity analyzer for the research purpose which modified and simplified user code and measured time complexity . The complexity analyzer is to work around 70 to 80 percent code as pattern in whose only sole duty is to calculate the time complexity.

3.1 Flow Diagram

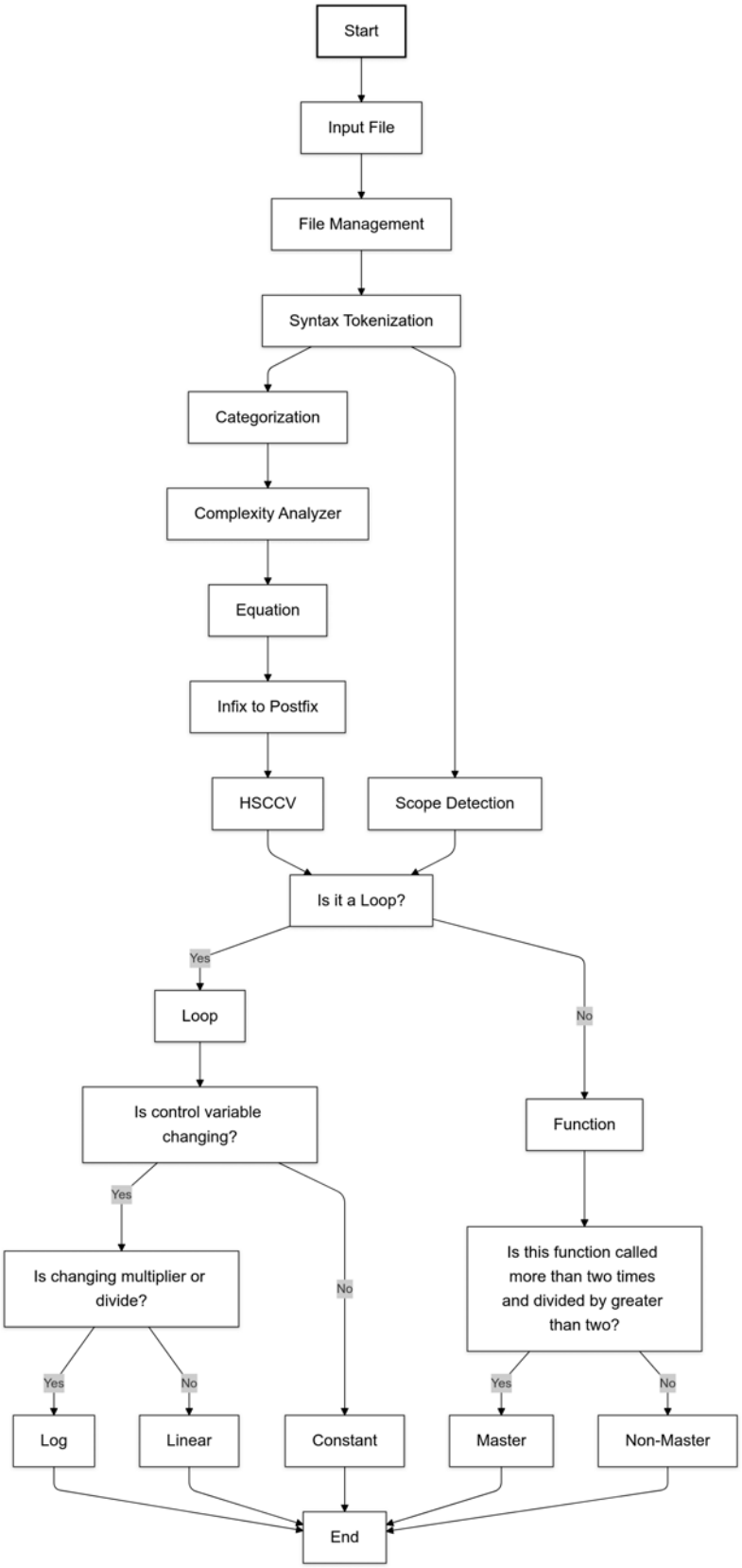


Figure 3.1: Working mechanism in flow diagram

3.2 Syntax Tokenization

Our research topic is only based on c++ code. It also can be used logically for java code. Following, filtration is critical in order to detect error, security and optimization. Through this process this tool can tokenize the code line at the initial state. Moreover, this tool can detect unnecessary lines which should not be considered for time complexity such as comments, header files and remove them at the initial state. This tool can proceed without any faulty input thus helps code correctness and ultimately, this tool can fix the errors which help developers primarily. Also, obliteration of unnecessary code and code elements the streamline produces and also it can create optimized machine code output. Moreover, by preventing erroneous inputs this tool can prevent unusual input. Thus, maintaining security. Proper filtering can help to correct code compilation and overall improvement in software quality can be ensured. However, the filtration system in our complexity analyzer helped us a lot. Essentially, filtration in our complexity analyzer functions for distinction of every line in a word which ultimately produces tokens. This tokenization implements a further pattern which is the third process. Moreover, filtering the code like this significantly assists in order to extract the file needed for consideration as well as obliteration of the files which is irrelevant. Thus, conclusively executing computation of time complexity.

Figure 3.2: Process of Syntax Tokenization

3.2.1 Syntax Tokenization module

```
def syn_tok(self, string):  # separating sign and word
    return_lst = []
    tmp_string = ""
    for each character in string:
        if ("a" <= chr <= "z"
          or "A" <= chr <= "Z"
          or "0" <= chr <= "9"
          or chr == "[" or chr == "]"
          or chr == '_'):
            # separating all word from code
        else:
            if (tmp_string includes , cin , cout):
                continue
            if (tmp_string != ""):
  # append word in return_lst stored in tmp_string
            tmp_string = "" #clear tmp_string for new char
    if (tmp_string != ""):
```

```
    # append remaining word in tmp_string to return_lst
    return return_lst
# Scoping: For scoping lines, find min and max boundary.
# In C++ or Java, most operations are scoped;
# are removed by loop iteration below.
```

User Input	Syn_tok Output

`#include <bits/stdc++.h>`
`using namespace std;`
`int binarySearch(int arr[], int l, int r, int x){`
`while (l <= r) {`
`int m = l + (r - l) / 2;`
`if (arr[m] == x) { return m; }`
`if (arr[m] < x) { l = m + 1; }`
`else { r = m - 1; }`
`}`
`return -1;`
`}`
	`['int', 'binarySearch', '(', 'int', 'arr', '[', ']', ',', 'int', 'l', ',', 'int', 'r', ',', 'int', 'x', ')', '{']`
	`['while', '(', 'l', '<', '=', 'r', ')', '{']`
	`['int', 'm', '=', 'l', '+', '(', 'r', '-', 'l', ')', '/', '2']`
	`['if', '(', 'arr', '[', 'm', ']', '=', '=', 'x', ')', '{']`
	`['return', 'm']`
	`['}']`
	`['if', '(', 'arr', '[', 'm', ']', '<', 'x', ')', '{']`
	`['l', '=', 'm', '+', '1']`
	`['}']`
	`['else', '{']`
	`['r', '=', 'm', '-', '1']`
	`['}']`
	`['}']`
	`['return', '-', '1']`
	`['}']`

Table 3.1: Example Code with Tokenized Output

3.3 Scope Detection

Second, boundaries or scoping are essential for locating specific code lines that need to be measured and for spotting nested loops. This is where the idea of scoping comes in very handy. Our findings emphasize how closely scoping is related to the execution flow of any program written in C++. Gaining an understanding of scope helps us select appropriate code parts for analysis, which makes runtime evaluations more efficient. For example

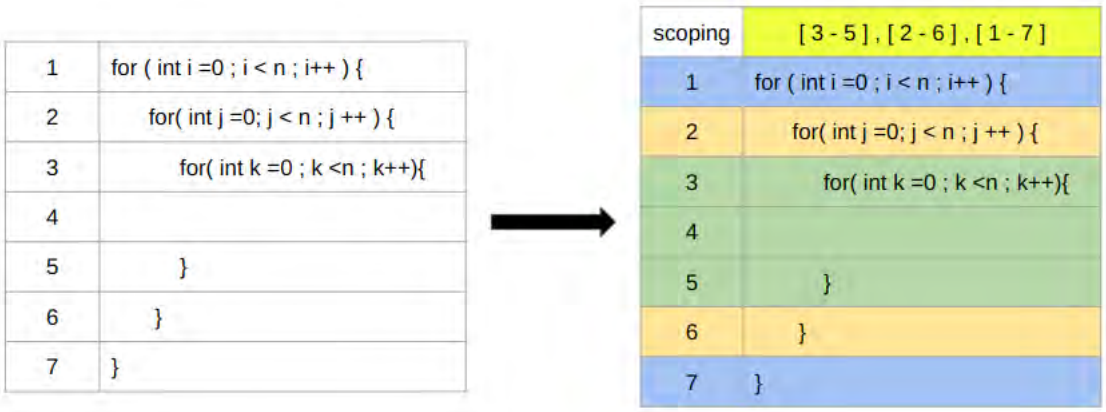


Figure 3.3: scoping of nested loop

Figure 3.4: Applying Scope Detection

In the above illustration, first this tool considers 3 to 5 lines first and marks them as visited. After that , 2 to 6 lines and 1 to 7 lines . Every time the closing bracket appears , memory will be cleared and multiplied with outer loop calculation. Additionally, boundaries assist in preventing a code line from being overridden. By applying scoping, which is saved in limits, those lines that are accessed only construct a checkpoint and prevent the wasteful computation. Additionally, this scoping will help with memory clearing following computation, which will become apparent later.

3.3.1 Scope Detection Module

```
def scope_detect(self.lst):
    stack = []
    tmp_lst=[]
    the number of line, count=0
    the maximum line of scoping, mx = 0
    for each line in of self.lst:
         if "{" in self.lst line:
            else: stack will append count
         if "}" in self.lst line:
            if stack is not empty:
     mn = find minimum line in mn and stack top to consider
                 stack.pop()
     mx = find minimum line in mx and stack top to consider
        count += 1
    boundaries=[]
    stack=[]
```

```
index=0

for each line from maximum to minimum:
    if "{" in self.lst line:
        if the length of self.lst is one
        :stack will append count -1
        else: stack will append count
    if "{" in self.lst line:
        if stack is not empty:
            store in boundaries stack top and index
            stack pop the top element
    index +=1
return tmp_lst,boundaries
```

Scope_detect Input	Scope_detect Output
['int', 'binarySearch', '(', 'int', 'arr', '[', ']', ',', 'int', 'l', ',', 'int', 'r', ',', 'int', 'x', ')', '{'] ['while', '(', 'l', '<', '=', 'r', ')', '{'] ['int', 'm', '=', 'l', '+', '(', 'r', '-', 'l', ')', '/', '2'] ['if', '(', 'arr', '[', 'm', ']', '=', '=', 'x', ')', '{'] ['return', 'm'] ['}'] ['if', '(', 'arr', '[', 'm', ']', '<', 'x', ')', '{'] ['l', '=', 'm', '+', '1'] ['}'] ['else', '{'] ['r', '=', 'm', '-', '1'] ['}'] ['}'] ['return', '-', '1'] ['}']	[3, 5][6, 8][9, 11][1, 12][0, 14] [17, 17][21, 22][15, 24]

Table 3.2: Example of Detecting Scopes

3.4 Categorization

Our objective was to find the meaning and motive of a code line to learn about what is the use of this line , will it be used in complexity analyzer , everything. To achieve this goal, we investigated methods based on natural language processing (NLP) and deep learning. But after more research, we realized that these methods went beyond what was necessary for our research because deep learning and natural language processing functions were blind to the structured nature of code. After realizing that code always follows established patterns, we shifted our attention to using regular expressions, or regex, to recognize patterns. Different jobs or functionalities are commonly represented by distinct lines in C++ programming. This tool successfully tokenized input code by utilizing regex pattern search techniques, which allowed us to retrieve valuable data and expedite our research process. This change in methodology not only made our process simpler, but it also improved how well our study identified and analyzed code.

3.4.1 Applying Regular expressions

A particular kind of character pattern known as regular expression is used to find a string or collection of strings by searching for the same pattern or similarities.It is a blueprint of fixed string pattern and by comparing a text with a specific pattern, it may determine whether the text is present or absent. If it is present in text ,it divides one or more substring.A collection of regular expression patterns in the dictionary as value and label as key , we named this dictionary in our research as regex.Here, this tool detect code lines and name them based on their regex pattern using regular expressions.When a matching string is found in the string, the search is terminated; if no match is found, it moves on to the next string and so on, going through all the patterns.

Figure 3.5: Applying Regular Expression

```
regex={
"while_loop": "^(while)\((.+)*\){*",
"for_loop": "^(for)\((.+)*\;(.+)*\;(.+)*\){*",
"for_loopE": "^(for)\((.+)*\:(.+)*\){*",
"condition": "^(if|elseif)\((.+)*\)",
"compare": "(.*)(==|=>|<=|>|<|!=)(.*)",
"assignE": "(.*)(\+=|\-=|\*=|\/=|\%=)(.+)*",
"assignBY_1": "(.*)(\+\+|\-\-)",
"define":"^(bool|int|string|char|float|double|void|
longlongint|unsignedlonglongint|unsignedint)(.+)*",
"assign":"(.*)(=)(.*)",
"else":"^(else){*",
"return":"^(return)(.+)",
"function":"(.+)\((.+)*\){?",
"break":"break"
}
```

Tokenized syntaxes	categorized syntaxes
['int', 'binarySearch', '(', 'int', 'arr', '[', ']', ',', 'int', 'l', ',', 'int', 'r', ',', 'int', 'x', ')', '{'] ['while', '(', 'l', '<', '=', 'r', ')', '{'] ['int', 'm', '=', 'l', '+', '(', 'r', '-', 'l', ')', '/', '2'] ['if', '(', 'arr', '[', 'm', ']', '=', '=', 'x', ')', '{'] ['return', 'm'] ['}'] ['if', '(', 'arr', '[', 'm', ']', '<', 'x', ')', '{'] ['l', '=', 'm', '+', '1'] ['}'] ['else', '{'] ['r', '=', 'm', '-', '1'] ['}'] ['}'] ['return', '-', '1'] ['}']	define [('int', 'binarySearch(intarr[],intl,intr,int x){')] while_loop [('while', 'l<=r')] compare [('l', '<=', 'r')] define [('int', 'm=l+(r-l)/2')] condition [('if', 'arr[m]==x')] compare [('arr[m]', '==', 'x')] return [('return', 'm')] condition [('if', 'arr[m]<x')] compare [('arr[m]', '<', 'x')] assign [('l', '=', 'm+1')] else ['else'] assign [('r', '=', 'm-1')] return [('return', '-1')]

Figure 3.6: Syntax Conversion Process

3.4.2 Subcategorization

After a pattern search of each line, this tool retrieves a list with tokens as the first word and regular expression subpatterns following. As this can see, certain tokens have the name "define" attached to them. Since 'define' can happen, there are two kinds of regex patterns, initial with function and initial with value. Since these are familiarized with the Java or C++ programming languages, they must define the data type before applying them. Only variables, arrays, and functions are taken into consideration when parsing here. Apart from that, each subcategory requires its own regular expression pattern. Programming languages frequently use the following data types: bool, int, string, char, float, double, void, long int, unsigned long long int, and unsigned int, among others. Once a 'define' token has been obtained, this tool must once more search the regex dictionary to identify this line function, value with variable, or value only. This tool assigns and functions regex patterns, and this tool just specifies search. Regex will assign, function, or initial without value if it detects a pattern. Here, this tool swaps $function\_with\_curly\_braces$ for the function token since there may be further patterns that match $(.+)(.+)*?$. For instance, built-in or recursive functions. This tool calls it `function_with_curly_braces` to differentiate it as initialization. Here, I am adding an illustration of this process.

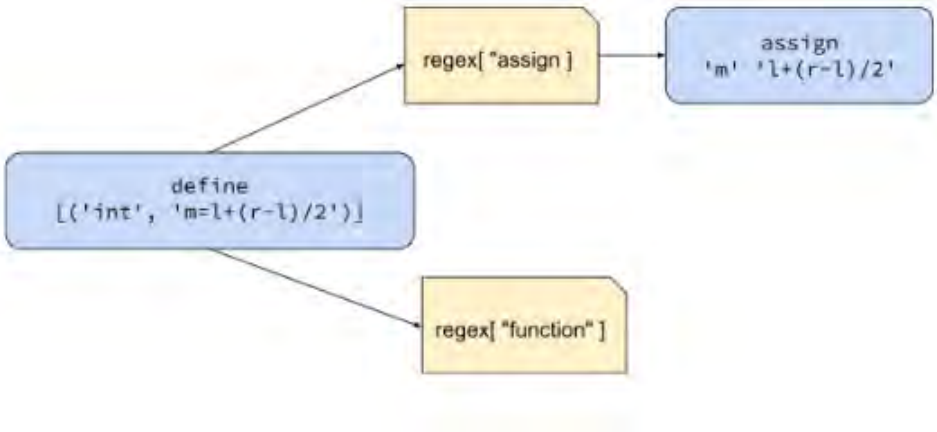


Figure 3.7: Process of Subcategorization

```
## here tmp has a token and subpattern list.

if(key=='define'):
   for ii in range(len(tm)):
       a = list(re.findall(dic["assign"], tm[ii]))
       b= list(re.findall(dic["function"], tm[ii]))
       if a:
           tm[ii]=list(a[0])
           return "assign",tm
       elif b:
           return "function_with_curly_braces",b
       else:return "initial",tmp
```

Before	After
define [('int', 'binarySearch(intarr[],intl,intr,intx){')] while_loop [('while', 'l<=r')] compare [('l', '<=', 'r')] define [('int', 'm=l+(r-l)/2')] condition [('if', 'arr[m]==x')] compare [('arr[m]', '==', 'x')] return [('return', 'm')] condition [('if', 'arr[m]<x')] compare [('arr[m]', '<', 'x')] assign [('l', '=', 'm+1')] else ['else'] assign [('r', '=', 'm-1')] return [('return', '-1')]	['function_with_curly_braces', [('binarySearch', 'intarr[],intl,intr,intx')]] ['while_loop', [('while', 'l<=r')]] ['assign', [['m', '=', 'l+(r-l)/2']]] ['condition', [('if', 'arr[m]==x')]] ['return', [('return', 'm')]] ['}'] ['condition', [('if', 'arr[m]<x')]] ['assign', [('l', '=', 'm+1')]] ['}'] ['else', ['else']] ['assign', [('r', '=', 'm-1')]] ['}'] ['}'] ['return', [('return', '-1')]] ['}']

Figure 3.8: Subcategorization

3.4.3 Expression tokenization

Currently the code have below pattern:

token	subpattern
for_loop	['for', 'inti=0,j=0', 'i<n&&j<n', 'i=i+12,j+=1']
while_loop	['while', 'i<10']
for_loopE	['for', 'inti', 'arr']
assignBY_1	['i', '++']
assignE	['i', '+=', '1+m']

Figure 3.9: Expression tokenization

We must make more modifications to user input before we can begin calculations.As we observe from the string list above, we may identify other patterns in each string. Future analyses will benefit from it since we won't have to deal with as many cases and can go right into the primary work. While the while loop has one portion that is in compare, the for loop can have three sections: initialization, comparison, and increment/decrement. Each subset has its own significance and programming language patterns. To identify subpatterns and provide more precise specifications, this tool once more scans the regex dictionary. Again, there are two special different types of increment and decrement for example i + + and i+= value. For the i++ case, this tool simply swap this statement to i = i +1 . Secondly the i+= value case, this tool just replaces this statement to i+=val to this i = i + val statement. Every case study will be handled by going through below pseudo code.

```
1  ## here tmp has a token and subpattern list . for example
       ['function_with_curly_braces', [('binarySearch', '
       intarr[],intl,intr,intx')]]
2
3  if(tmp[0]=="for_loop"
4  or tmp[0]=="while_loop"
5  or tmp[0]=="for_loopE"
6  or tmp[0]=="assignBY_1"
7  or tmp[0]=="assignE"):
8
9     for i in range(len(pt)):
10         dic = {}
11         if(len(pt[i])>0 and type(pt[i])==str):
12           # for specific type of pattern i+=1,i++,j+=m+q;
13             tp = pt[i].split(',')
14             for ii in tp:
15                 tm=self.pattern(ii) #find pattern from
                       text
16                 if(tm!=ii):
```

```
                    key,value=tm[0],list(tm[1][0])
                    if(key=="assignBY_1"):
                        ## It is a special kind of assign
                            that changes value linearly.
                           we do i++ to i=i+1
                        tm[0] = 'assign'
                        if(value[-1]=='++'):
                            value = [value[0], '=', value
                               [0], '+', '1']
                        if (value[-1] == '--'):
                            value = [value[0], '=', value
                               [0], '-', '1']

                    if (key == "assignE"):
                        ## It is a special kind of assign
                            that changes value with
                           special operator for example
                           +=, -= etc . we do i+=m to i=i
                           +m
                        tm[0]='assign'
                        if('+' in value[1]):
                            value=[value[0],'=', value
                               [0],'+',value[2]]
                        if ('-' in value[1]):
                            value = [value[0], '=', value
                               [0] , '-' , value[2]]
                        if ('*' in value[1]):
                            value = [value[0], '=', value
                               [0] , '*' , value[2]]
                        if ('/' in value[1]):
                            value = [value[0], '=', value
                               [0] , '/' , value[2]]
```

After handling all the cases our final string list will be:

token	subpattern
for_loop	['for', 'inti=0,j=0', 'i<n&&j<n', 'i=i+12,j+=1']
while_loop	['while', 'i<10']
for_loopE	['for', 'inti', 'arr']
assignBY_1	['i', '++']
assignE	['i', '+=', '1+m']

3.4.4 Simplification

The analyzer used a simplified token with a general name for grouping calculations in order to simplify the calculations. In this case, while and for loops would be tokens for "loops," which would be taken into account as grouped code in loop computation. In addition, the same task is assigned in a variable using three different types of tokens in the analyzer: "assign," "assignE," and "assignBY-1." To keep things simple, we substituted "assign" for "assignE," "assignBY-1," and "assign."

Before	After
for_loop	loop
while_loop	loop
assignBY_1	assign
assignE	assign

3.5 Complexity analyzer

After researching, we realized loop or recursion is dependent on variables that break the loop with some condition. For complexity analysis, this condition is the primary decision point where a loop or recursive call is either continued or terminated. We named these variables as control variables. Control variables involved in these control statements are crucial, as they dictate the flow of execution. These variables must be tracked and stored in appropriate memory spaces to facilitate accurate complexity calculations.

3.5.1 Tracking allocated memories

Before diving into the complexity calculations, This tool needs to establish six distinct memory types to manage different categories of variables and function executions. These memory types are essential for separating and organizing the various aspects of the program's structure and behavior. The six types of memory are:

3.5.1.1 Separating the scopes

Memories from main scope: This is where all global variables are stored. It constitutes the structures or variables that remain throughout the execution of a program. Those are frequently used over many functionalities or crosscutting concerns, contributing greatly to the complexity. It will store all assigned values and change in code. Another type of memory that can only be used for storing variables which have this particular scope, mainly inside some chunk of code or function block etc. These temporary variables only live while the block in question is being executed and are crucial for managing loops, conditionals or branches of recursion. It will store all assigned scope and clear after scope ends.

Function Variables: It will store function parameters and change parameters that occur with each function call. As functions are invoked, these variables hold the arguments passed to the function and may be updated throughout the function's execution. In recursive functions, the FunctionVariable memory is particularly important, as it handles a new set of parameters for each recursive depth.

```
void function_name_0 (int low,int mid,int high){
        function_name_0 (low,(low+high)/2, high);
 }
```

function_name_0	low	mid	high
	low	(low+high)/2	high

Figure 3.10: Function Variable

Loop Variables: This memory holds variables that control the execution of loops, such as loop counters, iterators, or conditions that terminate the loop. The performance of the loop, which can significantly impact time complexity, depends on how efficiently the Loop Variable is managed and updated. It will store control variable and initial variable and increment and clear when loop ends

```
for(int i=0,j=0;i<n && j<n; i=i+12,j+=1)
{
   i+=1+m;
   j=j+1;
   i++;
   m--;
 }
```

Loop Variable	i	j	n
	0+i+12+i+1+m+i+1	j+1+j+1	

Figure 3.11: Loop Variable

Function calls: This is a container for the function execution context, tracking function calls and their hierarchical relationships. Understanding how functions call each other (especially recursively) is key to determining the time complexity. It will store numbers and call each function. This memory type is dedicated to the

```
 void function_name_1 (){
}
 void function_name_0 (){
        function_name_0 ();
        function_name_1 ();
        function_name_1 ();
}
 void function_name_2 (){
}
```

function	function_name_0	function_name_1	function_name_2
	1	2	0

Figure 3.12: Function calls

actual storage of complexity metrics and formulas. As the program runs, complexity memory keeps track of the time and space resources consumed, allowing for real-time analysis and optimization suggestions based on current usage patterns. It will store the complexity of each function and loop. we will clear after traversing all lines of code.

3.5.1.2 Tracking process

This module in the user code example handles a variety of programming operations, including assignment, return statements, loops, and function declarations. It then arranges the information about these constructs into memory. This module offers an organized method for keeping track of variables, assignments, and function calls, which is helpful for further consideration. The parameter list 'par_lst' will be kept in both 'main scope' and 'ScopeMemory' if the 'token' contains an 'assign' or 'return' statement. According to this, variables and their values are stored in two separate scopes: 'main scope' appears to be a global memory or general storage. Local memory for a particular scope or context, like a function or loop, may be represented by 'ScopeMemory'.The method provides an area for loop variables in 'LoopVariable' if there is a comparison inside the loop.Every assigned value in the loop statement is saved in 'LoopVariable' if there is an assignment inside the loop. For instance, ''i': 'i+1'' will be stored in 'LoopVariable' in an assignment statement such as 'i = i + 1'. This maintains the status of every variable during the loop's execution by tracking all variable updates. For the function to process the arguments provided throughout calls and maintain state, 'FunctionVariable' keeps track of the parameters and their each call changing. Every function parameter is kept in 'FunctionVariable', where they are all configured with an empty list (for example, a function with parameters 'a', 'b', and 'c' would store 'a: [], b: [], c: []').In order to maintain track of the function's call count, the 'Function' count for the function name is increased by 1. Program management and tracking of function calls are made possible by this structure, which also keeps track of call counts and records parameters for each function application. This design pattern organizes program constructs, which may be useful in an interpreter to execute or simulate code execution while tracking scopes, loops, and functions.

```
def formation(self, par_lst):
   if token is assign:
       par_lst will store in Main scope
       par_lst will store in ScopeMemory

   if token is return :
       par_lst will store in Main scope
       par_lst will store in ScopeMemory

   if token is loop:
       if token is compare:
           for each variable will take space in
              LoopVariable:
               LoopVariable[variable] = []
       if token is assign:
           for each value in loop statement assigned will
               stored in LoopVariable:
                ## for example {assign:['i', '=', 'i+1']}
                    ,
                  in LoopVariable it will store in
                     loopVariable as {'i':'i+1'}
                value will stored in LoopVariable

   if token is function_with_curly_braces:
       Initial function name in function with zero as it
          initialization of function

       for each parameter function will stored in
          FunctionVariable:
            ## for example function ( a , b, c),
                  in parameter it will store in
                     FunctionVariable as {a:[], b:[] ,c
                     :[] }
                value will stored in FunctionVariable

   if token is function:
       separate every parameter from function call
       stored every parameter in MainMemroy according to
          function name
       if function_name in not in  function: Function[
          function_name] = 0
       Function[function_name] increment by 1
```

3.5.1.3 Control Variable:

Control variables are a very important and considerable thing for measurement. When a loop is broken depends on the comparison of variables , those variables

we named control variables. For example ,for (int i=0;i<n;i + +) this loop will break when the i variable is greater than n variable. So, here is the i variable and the n variable is the control variable. In order to increase the variable-i, it must be assigned consistently [table 1]. [table: 2] If another variable assigns the control variable, this must determine whether that variable is assigned within the loop's scope; if not, it will be regarded as a constant. Furthermore, [table;3] Assigning a control variable that is not increment or decrement can result in an endless loop and conflict with the entire program. This tool can identify and throw an endless loop error if a program crashes with this type of infinite loop.

`for (int i=0;i<n;i++){`
`  i=i+m`
`}`
	`Table :1`
	`i = i + 1 + i + m`

`for (int i=j+1;i<n;i++){`
`  i=i+m`
`}`
	`Table :2`
	`i = i + 1 + j [ constant] + m`

`for (int i=0;i<n;){`
 	
`}`
	`Table:3`
	`i =`

3.5.1.4 Return statement:

Return statements are essential for spotting recursive behavior in code analysis. Return statements can be divided into two main categories: return value, which only returns a value, and return a function call. A thorough examination of return function statements is necessary to accurately identify recursion. This can be achieved by storing the information in Main Memory and using a regex dictionary to determine whether the return involves a function call.There is more to the study of recursive complexity than simply figuring out which function calls itself. To comprehend the wider situation, including changes in variables and interactions with other functions, all lines of code must be taken into consideration. This involves evaluating every function call and parameter change that could affect the recursion. Following the state of variables and their transitions throughout the program is necessary to get a full picture of recursive complexity. Recursive complexity is thereby computed after each function has been examined in order to take into consideration each variable change, function call, and potential effects on memory usage and performance. Given their dependencies and interactions with other code elements, this comprehensive analysis confirms that the computational and memory requirements of recursive functions are precisely measured.

```
1 def return_statment(self,par_lst):
2         for each element in par_lst:
3             if element is function in return statement:
4                 increment in function memory
5                 split parameter according to ','
6                 store in Main Memory in  function key with
                      index.
```

3.5.1.5 Function call:

If we take a profound look on C++ this tool can encounter that all the C++ code is executed in function. This tool recognises them by seeing the curly braces that have been put and also by identifying the initialization. In addition, the name of the function and parameters that this tool needs to set are defined in FunctionVariable and Function. Conversely, when this tool has a function that has already been defined in FunctionVariable this tool can effortlessly call it a recursive call and thereby can change the parameters that are defined in FunctionVariable. Additionally, two types of return statements that are presented. return functions are typically used for returning vector, arr, number and string etc. As well as it can be used as a recursive call. However, when return statements are returning a function we can thereby assess this as recursion and thus store it in the FunctionVariable.

`int binary search(){`
`  binary search();`
`  sort()`
`}`
	`Function Variable:`
	`binary search : recursive`
	`sort : built-in function`

3.5.2 Evaluation

The complexity analyzer made code into an equation. In the main scope we have for example ‘ i ’ : [‘i+1’, ‘(m-1) / 2’, ‘n+2’] . For the calculation purpose , complexity analyzer add an assigned value with a plus sign such as ‘ i ’ : ‘ i + 1 + (m-1) / 2 + n+2 ’ or it would be like this i = i + 1 + (m-1) / 2 + n+2 . As plus is a very low precision , the result would not change. Calculation scans either from left to right or from right to left for calculator expression. The expression is scanned by the compiler either from right to left or from left to right. It is very inefficient for the repeated scanning. It is preferable to convert the expression infix to postfix form before doing evaluation because infix expressions are easier for humans to read and solve while computers have difficulty differentiating between operators and parentheses.

3.5.2.1 Infix to postfix:

After modification, we concluded that infix to postfix conversion would be the best and easiest for arithmetic calculations. This would be meaningful and easy for future calculations to retain precision. This tool transfers the code in an equation .Infix to postfix is the most practical method since it is the equation for calculation.It is the most useful tool for our research because it makes it simple to define precision and priority. Initially, we believed that the tree would be a useful calculation if we put the sign as parent and value as child. Thus, ’l’: ’lrl-2/+1’, ’r’: ’lrl-2/+1-’ would be saved. With large-scale, complex calculations, this won’t change. The outcome remains the same.For example, the pseudo code will appropriately handle the particulars of operator order and parenthetical grouping given an input of l+ (r - l) / 2+ 1, ultimately yielding a postfix output of l r l - 2 / 1 + +. There is no uncertainty in evaluating this postfix expression in a simple left-to-right manner without ambiguity.

3.5.2.2 Finding the Most Significant Change of the Control Variable (MSCCV):

Finally, for postfix evaluation, as we are doing with strings. Integers would not always be obtained. We enter 'n' as the response in subsequent applications after the initial procedure as the first answer if two variables one of is control variable otherwise is 'constant'. Gradually all calculations will be considered the same as this will change our control variables.In essence, the function evaluates a postfix expression by turning it into a list of infix-style expressions and using a stack for intermediate computations. It keeps track of every step while identifying and applying arithmetic operations to operands kept in the stack. The stack-based method makes it easier to handle postfix expressions and enables correct evaluation without relying on parentheses to determine precedence.

```
 def centralfunc(self, ab,key):
       list=[]
       for i in ab:
           if i not an arithmetic sign:
                 push in self.stack
           else:
               val1 = self.pop()
               val2 = self.pop()
               if i is  '/':
                   push n in self.stack
                   append val2+'/' + val1 in list
               elif i == '^':
                    push n in self.stack
                   append val2+'**' + val1 in list
               elif i == '-':
                    push n in self.stack
                   append val2+'-' + val1 in list
               elif i == '*':
                    push n in self.stack
                   append val2+'*' + val1 in list
               elif i == '+':
                    push n in self.stack
                   append val2+'+' + val1 in list
       return list

def linear_or_logarithmic(val1, val2,sign,par_dict,call):
   ans='n'
   if it a loop scoping:
       if (val2 not in calculation.ScopeMemory
               and val1 not in calculation.ScopeMemory
               and val2 not in par_dict
               and val1 not in par_dict
       ):
               ans='c'
       else: ans='n'
```

```
36      if(sign is '+' or sign is '-'):
37              return ans
38      else: return 'log'+ans
```

After passing through all the modules explained above, the user input code is converted to the following:

```
1  loopVariable:{'l': ['r-l', 'n/2', 'l+n', 'n+1'], 'r': ['r
       -l', 'n/2', 'l+n', 'n+1']}
2
        ------------------------------------------------------

3  functionVariable:  {'binarySearch': [['arr[]'], ['l'], ['
       r'], ['x']], 'main': [['void']]}
4
       ------------------------------------------------------

5  function:  {'binarySearch': [], 'main': []}
```

3.5.2.3 Result

In our research , we can successfully detect a loop, nested loop, function, recursion, built in function, infinite loop for some cases etc. For testing purposes we use ideal c++ code for example sorting algorithms etc can compute time complexity and can tell if a recursive is a master theorem or not. The process of calculation is described below.

3.5.2.3.1 Loop:

For illustration purpose , this tool will do user code to this form loopVariable:'l': ['n', 'logn', 'n', 'n'], 'r': ['n', 'logn', 'n', 'n']. For the loop We can say that they are changing logarithmically if they change when we divide and multiply in an iteration. Otherwise, they are changing linearly if it changes add or sub. The 'n' variable, that is, 'n', 'logn', or some other 'constant', must be connected to the control variable. Think O(n) whenever the tool detects + or - operations, such as i = i + 1 or i = i - 1.Consider O(log(n)) whenever you see * or / operations (i = i * 2, i = i / 2).Consider O(1) if there is no relationship with n or if the loop executes a predetermined number of times independent of n. There is a special kind of loop in c++. for(int i:vec), it is a special case where ' i ' denotes every element in a vector. So maximum run time is the size of the vector. So complexity is the size of a vector, which is O (n).

3.5.2.3.1 Nested loop:

This provided pseudocode checks for code scope nesting, with a special emphasis for identifying nested loops or nested code lines and calculating the total complexity. The main idea here is to use a dictionary par_dict, which contains information about these code parts, to analyze the structure of scopes like loops or functions.Finding out whether scopes are nested within one another and, if so, calculating the complexity by adding the complexity of the nested scopes is the aim. For instance, the

complexity may increase multiplicatively if you have nested loops. A loop inside another loop, for instance, could alter the complexity from O(n) to $O(n^2)$.A single for loop with complexity O(n) nested inside another for loop with complexity O(n) results in $O(n^2)$.In order to identify such scenarios, the pseudocode determines whether one scope (loop) is inside another and modifies complexity calculations appropriately.Nesting, which necessitates the multiplication of complexities, is indicated if one scope starts and ends inside another.When you want to predict the complexity of a function or code snippet by finding loops and recursive functions, this method is especially helpful for static code analysis.If there are more than one nested loop, the complexity of each loop is increased.Without nesting, scopes' complexities stay different and are added after the fact.

```
for ( int i = 0 ; i < n ; i++ ) {
      for ( int j =0 ; j < m ; j ++ ) {
       }
}
```

Figure 3.13: Nested loop

The nested function would in this example, Understand that the first loop for (int i = 0 ; i < n ; i++) contains the second loop for (int j = 0 ; j < m ; j ++). Consider the fact that the inner loop is completely contained within the outer loop, indicating that both loops are inclusive. Their complexities can be multiplied to obtain O(n) * O(m) = O(n*m).

```
1
2  def is_nested(par_dict):
3      tmp_lst=[]
4      prev1,prev2=0,0
5      prev_tuple=0
6      for key in par_dict:
7           append key in tmp_lst
8      for each string in tmp_lst:
9          if for the first scope:
10             previous start=scope starting index
11             previous end=scope last index
12             previous tuple= scope
13         else:
14             new start =  scope starting index
15             new end = scope last index
16             if  previous scope is inclusive in new scope :
17                 complexity is stored every scope
                       complexity.if it is inclusive it will
                       multiple such as new_complexity*
                       prev_complexity.this is how we detect
                       nested loops.
18             previous start= new start index
19             previous end= new end index
```

```
20              previous tuple= new scope
```

3.5.2.3.1 Function:

According to the theorem , if the function call is a times (a>2) and divides each time n/b [b is a constant where b>1], it will consider the master theorem , otherwise it will not consider it.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

In this case, if a ==1, it could be an increment, a decrement, or the master theorem, but there are still a lot of debts related to that. The situation and the code determine this. Every function call in function memory will be counted by this tool. This tool will be defined as a master theorem if a function is greater than two and divides by two each time. Function= 'function-name00':2,'function-name01':1,'sort':0 is an example. The tool will divide the number from the main memory in this case since function-name00 is called twice. The master theorem detects this in this way. 'sort' is another function that is called in code, but it is never called as recursive because its value is zero. It is a built-in function. According to the above complexity is below:

```
1  --------------Loop Analysis--------------
2  >>>> scope:  (1, 12)
3  >>>> The complexity is O(logn)
4  ['while', '(', 'l', '<', '=', 'r', ')', '{']
5  ['int', 'm', '=', 'l', '+', '(', 'r', '-', 'l', ')', '/',
       '2']
6  ['if', '(', 'arr', '[', 'm', ']', '=', '=', 'x', ')', '{'
     ]
7  ['return', 'm']
8  ['}']
9  ['if', '(', 'arr', '[', 'm', ']', '<', 'x', ')', '{']
10 ['l', '=', 'm', '+', '1']
11 ['}']
12 ['else', '{']
13 ['r', '=', 'm', '-', '1']
14 ['}']
15 ['}']
16
17 --------------Recursive Analysis--------------
18 binarySearch is not recursive
19 main is not recursive
20
21 --------------Number of function called--------------
22 main is Main function
```

Chapter 4

Result Analysis

According to our research, the most usable code complexity is O(n), O(1), O(log(n)), O(n!), $O(2^n)$, $O(n^a)$. O(n!) is only seen in next permutation which is a built-in function in c++. We need to find how to control variables changing in every iteration and also need to observe the parameter in every function call. The purpose of comprehending and managing time complexity is to strike a balance between an algorithm's efficiency and its resource intake.To make sure that code operates as efficiently as possible, particularly as the input size grows, it is essential to analyze variable changes, monitor recursive calls, and apply optimization techniques.our research is applicable for 70% c++ code and can compute properly. Complexity analyzer can detect loops, nested loops, recursive functions etc. Here 30% code will not be able to be calculated for syntax parsing. As our motive was to focus calculation, so our tool is applicable for fixed code snippets. This tool is only finite loops and recursion where for infinite loop detection needs more research in comparison.For recursive call, In this case, if a ==1 means function call is one in master theorem, it could be an increment, a decrement, or the master theorem, but there are still a lot of debts related to that. The situation and the code determine this. Moreover, Our research is based on scoping. In C++, a for loop or while loop can run without scoping. According to our research, without scoping, only one line is considered. Other lines are out of scoping. For example, `for (int i = 0; i < n; i++) cout « "HelloWorld";` again `for (int i = 0; i < n; i++) cout « "HelloWorld";` i++ ; here incrementation i++ will not consider.Variation for programming language and update syntax maybe change, The way of calculation and proposes would be same.

Chapter 5

Limitations

We found some logical limitations in our research. The first limitation of our research is the assumption that the user will always input error-free code. As we are creating a complexity measuring tool and our only motive is to calculate time complexity .This indicates that scenarios in which the code may have logical or syntactic mistakes or other problems that could prohibit it from operating successfully are not taken into consideration by our study. Rather, we have only examined situations in which the code runs error free but generates inaccurate output. This supposition severely limits the breadth of our investigation because real-world programming frequently entails handling different kinds of mistakes that may arise during code execution. Finally, the sole input we used for our investigation was C++ code. Although C++ is a strong and popular programming language, concentrating only on it limits our research to that programming language. Several programming languages are used in modern software development, each having its own special characteristics and difficulties. Our research does not cover the entire range of programming environments since it excludes certain languages, particularly Python and higher-level languages used in machine learning. Particularly in the fields of data science and machine learning, Python is renowned for its ease of use and broad application. Taking it out of the analysis leaves a big hole in our research because a lot of modern apps rely on Python and related languages. In our research, the most challenging issue is dealing with Python code and higher-level code utilized in machine learning, which is becoming more and more significant in software development. For contemporary applications, Python libraries and frameworks like PyTorch and TensorFlow are essential. Incorporating these domains into our research would augment its pertinence and yield more all-encompassing resolutions.

Chapter 6

Conclusion

In conclusion, our research gives a measurement in order to search through calculating the time complexity of c++ code by finding loops and mitigating the master theorem. By meticulous and thorough analysis the effective method can impact in order to detect and evade the infinite loops in some cases .It is a success that this research transfers code into equations and represents mathematical calculation. This way it helps in performance, lowering runtime, and effectiveness which makes the code more reliable and useful. The insights that can be gained may aid in adopting modern ways of performance to be optimized. Using modern techniques will serve into performance optimization, dependability and effectiveness in future.

Bibliography

[1] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, 1981. DOI: 10.1109/TIT.1981.1056404.

[2] J. N. Knight and M. Lunacek, "Reducing the space-time complexity of the cma-es," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07, London, England: Association for Computing Machinery, 2007, pp. 658–665, ISBN: 9781595936974. DOI: 10.1145/1276958.1277097. [Online]. Available: https://doi.org/10.1145/1276958.1277097.

[3] T. Chen, J. He, G. Sun, G. Chen, and X. Yao, "A new approach for analyzing average time complexity of population-based evolutionary algorithms on unimodal problems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 5, pp. 1092–1106, 2009. DOI: 10.1109/TSMCB.2008.2012167.

[4] Z. Nopiah, M. Khairir, S. Abdullah, M. Baharin, and A. Arifin, "Time complexity analysis of the genetic algorithm clustering method," in *Proceedings of the 9th WSEAS international conference on signal processing, robotics and automation, ISPRA*, vol. 10, 2010, pp. 171–176.

[5] S. G. Devi, K. Selvam, and S. Rajagopalan, "An abstract to calculate big o factors of time and space complexity of machine code," 2011.

[6] M. Mezini, *ECOOP 2011–Object-Oriented Programming: 25th European Conference. Lancaster, UK, July 25-29, 2011, Proceedings*. Springer, 2011, vol. 6813.

[7] W. Xiang, "Analysis of the time complexity of quick sort algorithm," in *2011 International Conference on Information Management, Innovation Management and Industrial Engineering*, vol. 1, 2011, pp. 408–410. DOI: 10.1109/ICIII.2011.104.

[8] Y. S. Patel, N. K. Singh, and L. K. Vashishtha, "Fuse sort algorithm a proposal of divide conquer based sorting approach with o(nloglogn) time and linear space complexity," in *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*, 2014, pp. 1–6. DOI: 10.1109/ICDMIC.2014.6954240.

[9] P. S. Oliveto and C. Witt, "Improved time complexity analysis of the simple genetic algorithm," *Theoretical Computer Science*, vol. 605, pp. 21–41, 2015, ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2015.01.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397515000298.

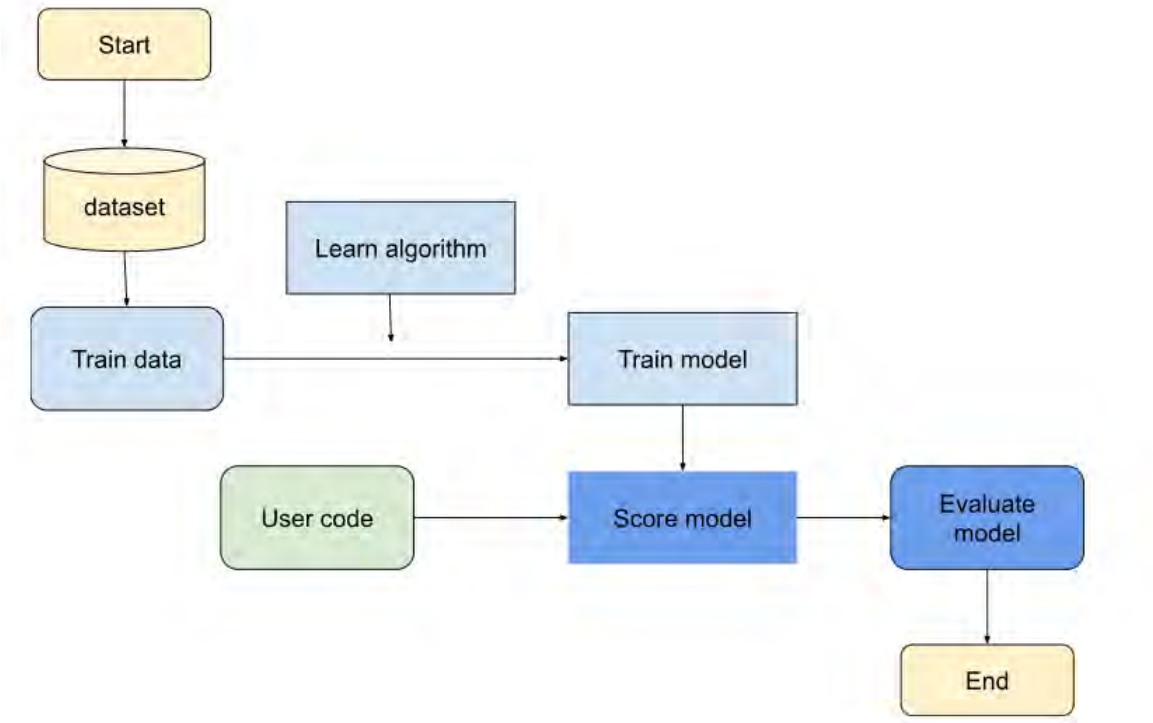
[10] T. Dobravec, "Estimating the time complexity of the algorithms by counting the java bytecode instructions," in *2017 IEEE 14th International Scientific Conference on Informatics*, 2017, pp. 74–79. DOI: 10.1109/INFORMATICS.2017.8327225.

[11] H. Férée, S. Hym, M. Mayero, J.-Y. Moyen, and D. Nowak, "Formal proof of polynomial-time complexity with quasi-interpretations," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018, Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 146–157, ISBN: 9781450355865. DOI: 10.1145/3167097. [Online]. Available: https://doi.org/10.1145/3167097.

[12] Y. Noller, R. Kersten, and C. S. Păsăreanu, "Badger: Complexity analysis with fuzzing and symbolic execution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '18, ACM, Jul. 2018. DOI: 10.1145/3213846.3213868. [Online]. Available: http://dx.doi.org/10.1145/3213846.3213868.

Appendix I: Other explored ideas

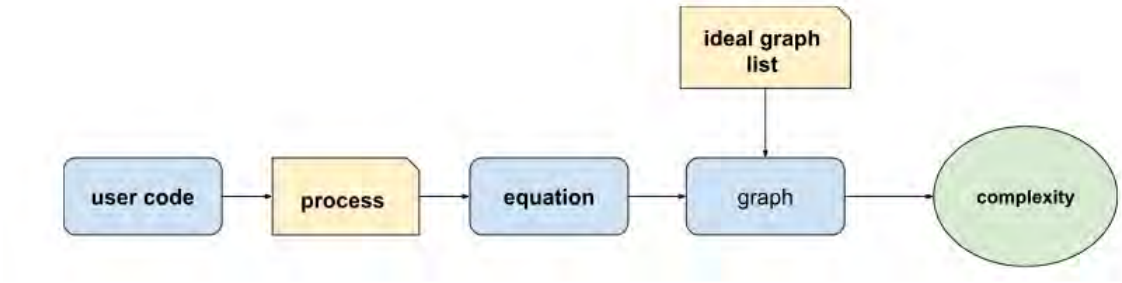
Our first research was to find time complexity using machine learning. Here we create a dataset where one column would be code and the second column would be time complexity. Then learn algorithms and train machine learning models for measurement. Next, the user input code which would classify the highest percentage of matching dataset code and user code, and the time complexity would be according to the dataset code.



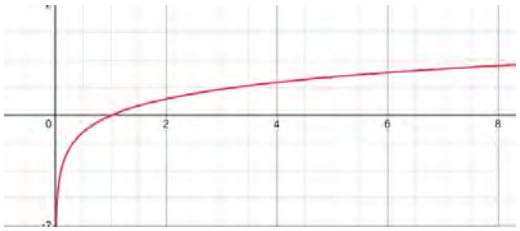
We did not process this idea as for time complexity measurement the train model always would not be able to find accurate time complexity. Maybe there will be different types of user code where machines can not find any score which may cause drawbacks in our research.

Approach 02

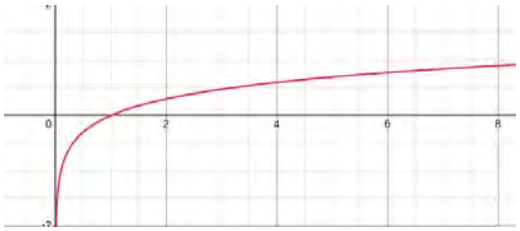
One of our research was it is possible to transfer a code to a mathematical equation and then we will plot the value according to the equation. After that, we will compare the ideal graph and declare complexity.
The primary cause of this study's failure was the difficulty of converting a code into an equation. Many variables might be constant during each iteration or should not be taken into consideration while making calculations. Once more, graph charting will be challenging if a loop or recursion depends on too many variables because each change requires a graph point. for example,f(x)=x+m*2+n/2 – 1.0 One of the other approaches was to swap all variables into one variable. for example, f(x)=x+x*2+x/2– 1.1 But if the equation is like this, f(x) = (r - l)/ 2+1 –1. 2 the



equation will be after swapping all variables in one variable, f(x) = (x - x)/ 2+1 –1.3 Here is the main catch where this research will not be able to calculate properly. This is the key point at which our research will be unable to perform accurate calculations. According to the 1.3 equation, an important portion will be removed for the same variable method. It will happen continuously, which is incorrect; log n or n/2 will be the true computation. Second, We plan to find a similarity in image processing between the conduct graph and the ideal graph. Image processing compares two image pixels to pixels. If comparing the same image returns 100 percent similarity(figure 3.5) and two different images return 92.1 percent similarity(figure 3.6). Two different images have a white background and the dissimilarity is the line-coloured curve. This line-colored curve represents the deviation in pixel values between the two images, highlighting where the differences occur. Despite the high similarity score, the curve signifies that even small variations can affect the overall comparison. If we put two photos, one with a white background and another with a black background, then the similarity would be 0 percent, as it counts only pixel to pixel. Our motive was to get two graph lines that were similar or not which we couldn't accomplish because it was only counting pixel to pixel. In complexity measurement, similar small changes in variables or execution paths do not create significant changes. Only checking two graphs is not enough for complexity measurement because complexity analysis involves more than just similarity between graph lines.

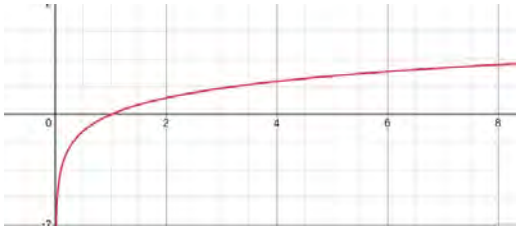


(a) White Background Image

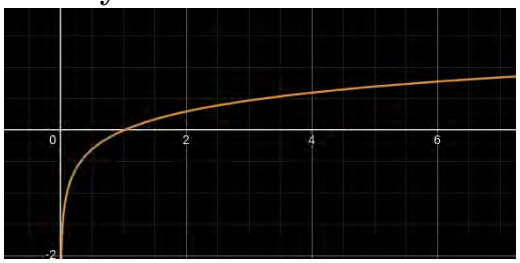


(b) White Background Image

100 percent similarity



(c) White Background Image



(d) Black Background Image

80 percent similarity

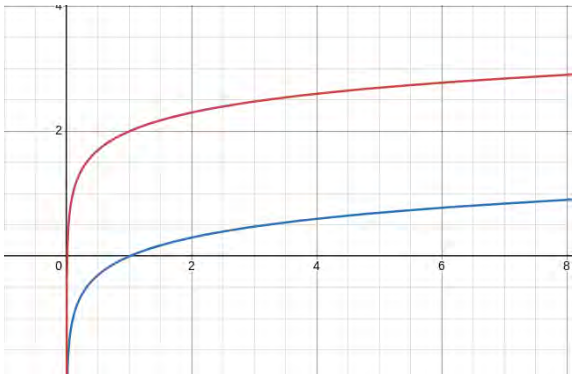


Figure 6.1: Graphs for comparison

Figure 6.2: Graphic Approach summary

Appendix II: Code Repository

Our research only supports this kind of code pattern. Otherwise, it will not be able to calculate time complexity for syntax parsing.

```
#include <bits/stdc++.h>
using namespace std;

int main(){
        for(int i=0;i<n;i++){
                i=i+1;
                i+=1;
                i++;
        }
        for (int i:arr){
                cout<<i;
        }
        int i=0;
        while(i<n){
                i+=1;
        }
        return 0;
}
```

Postfix Expression:

```
import calculation
class Postfix_Expression:
    def __init__(self):
        self.stack = []
        self.top = -1


    def pop(self):
        if self.top == -1:
            return
        else:
            self.top -= 1
            return self.stack.pop()


    def push(self, i):
        self.top += 1
        self.stack.append(i)


    def centralfunc(self, ab,key):
        lst=[]
```

```
23
24
25          for i in ab:
26
27
28              if (i not in calculation.arithmetic):
29                  self.push(i)
30              else:
31                  val1 = self.pop()
32                  val2 = self.pop()
33
34
35                  if i == '/':
36                      self.push('n')
37                      lst.append(val2+'/' + val1)
38                  elif i == '^':
39                      self.push('n')
40                      lst.append(val1+'**'+val1)
41
42
43                  elif i == '-':
44                      self.push('n')
45                      lst.append(val2+'-' + val1)
46                  elif i == '*':
47                      self.push('n')
48                      lst.append(val2+'*' + val1)
49
50
51                  elif i == '+':
52                      self.push('n')
53                      lst.append(val2+'+' + val1)
54
55
56          return lst
57
58
59
60
61  # #Driver code
62  # if __name__ == '__main__':
63  #     str = '100 200 100 - 2 / + 1 -'
64  #
65  #     # Splitting the given string to obtain
66  #     # integers and operators into a list
67  #     strconv = str.split(' ')
68  #     obj = evalpostfix()
69  #     print(obj.centralfunc(strconv))
```

Pattern:

```
from filteration import filteration
import re
class pattern(object):
    def __init__(self,lst):
        self.lst=lst


###################### Method ######################
    def pattern(self,string):
        dic={
        "while_loop": "^(while)\((.+)*\){*",
        "for_loop": "^(for)\((.+)*\;(.+)*\;(.+)*\){*",
        "for_loopE": "^(for)\((.+)*\:(.+)*\){*",
        "condition": "^(if|elseif)\((.+)*\)",
        "compare": "(.*)(==|=>|<=|>|<|!=)(.*)",
        "assignE": "(.*)(\+=|\-=|\*=|\/=|\%=)(.+)*",
        "assignBY_1": "(.*)(\+\+|\-\-)",
"define":"^(bool|int|string|char|float|double|void|
   longlongint|unsignedlonglongint|unsignedint)(.+)*",
        "assign":"(.*)(=)(.*)",
        "else":"^(else){*",
        "return":"^(return)(.+)",
        "funtion":"(.+)\((.+)*\){?",
        "break":"break"
        }

      for key,value in dic.items():
          tmp=re.findall(value, string)
          if tmp:
              if(key=='funtion'):
                 tmp[0]=list(tmp[0])
                 tmp[0][1]='('+tmp[0][1]+')'
                st=tmp[0][1]
                for pp in range(len(string)-1):
                  if(string[pp] in ['==','=>','<=','>','<
                     ','!='] and key=='compare'):
                      return 'compare',string
                 if (string[pp]+string[pp+1] in ['==', '
                    =>', '<=', '>', '<', '!='] and key=='
                    compare'):
                   return 'compare', string
              if(key=='define'):
                  tm=st.split(',')
                  for ii in range(len(tm)):
                      a = list(re.findall(dic["assign"
                         ], tm[ii]))
```

```
42                          b= list(re.findall(dic["funtion"
                               ], st))
43                          c = list(re.findall(dic["assignE"
                               ], tm[ii]))
44                          if a:
45                              tm[ii]=list(a[0])
46                              return "assign",tm
47                          elif b:
48                              return "
                                   function_with_curly_braces
                                   ",b
49                          elif c:
50                              tm[ii] = list(c[0])
51                              return "assignE",tm
52                          else:return "initial",tmp
53                  return key,tmp
54          return string
55
56 ###################### Method ######################
57
58      def pattern_match(self,string):
59          tmp=list(self.pattern(string))
60          output=[]
61          output_dict={}
62          if(len(tmp)>1):   ##to remove extra parenthesis
63              filteration([]).reemovNestings(tmp[1], output,
                       output_dict)
64              tmp[1]=output
65              pt = tmp[1]
66          #print("I_want_to_see",tmp[0])
67          if (tmp[0] == "assignBY_1" or tmp[0] == "assignE")
                :
68              string = ''
69              for i in pt: string += i
70              if (string != ''): pt = [string]
71              #print("pt",pt,'tmp',tmp)
72          #### i+=1,j=j+1,i++;
73          if(tmp[0]=="for_loop"
74          or tmp[0]=="while_loop"
75          or tmp[0]=="condition"
76          or tmp[0]=="for_loopE"
77          or tmp[0]=="assignBY_1"
78          or tmp[0]=="assignE"):
79              ##### this if-else making i+=1 = i=i+1 or i++
                       = i=i+1
80              if(tmp[0]=="assignBY_1" or tmp[0]=="assignE"):
81                  string=''
82                  for i in pt: string+=i
```

```
83                  if(string!=''): pt=[string]
84              #### this loop iteration is for i+=1,j=j+1,i
                   ++; -> assign i=i+1,....
85
86              for i in range(len(pt)):
87                   dic = {}
88                   if(len(pt[i])>0 and type(pt[i])==str):
89                       tp = pt[i].split(',')
90                       #print("I_want_to_see",tmp,tp)
91
92                       for ii in tp:
93                           tm=self.pattern(ii)
94                           if(tm!=ii):
95                               # tm=('assign', [('lamka', 'l
                                    +m')]) ii=intlamka=l+m
96                               tm=list(tm)
97
98                               key,value=tm[0],list(tm
                                    [1][0])
99                               if(key=="assignBY_1"):
100                                  tm[0] = 'assign'
101                                  if(value[-1]=='++'):
102                                      value = [value[0], '=
                                             ', value[0], '+',
                                             '1']
103                                  if (value[-1] == '--'):
104                                      value = [value[0], '=
                                             ', value[0], '-',
                                             '1']
105
106                              if (key == "assignE"):
107                                  tm[0]='assign'
108                                  if('+' in value[1]):
109                                      value=[value[0],'=',
                                             value[0],'+',value
                                             [2]]
110                                  if ('-' in value[1]):
111                                      value = [value[0], '=
                                             ', value[0] , '-'
                                             , value[2]]
112                                  if ('*' in value[1]):
113                                      value = [value[0], '=
                                             ', value[0] , '*'
                                             , value[2]]
114                                  if ('/' in value[1]):
115                                      value = [value[0], '=
                                             ', value[0] , '/'
                                             , value[2]]
```

```
116
117                                 if(key=='compare'): value=tm
                                       [1]
118
119                                 if(tm[0] not in dic): dic[tm
                                       [0]]=[]
120                                 dic[tm[0]].append(value)
121
122                                 #print(tm, ii)
123                                 #### tm=['assign', ['lamka',
                                       'l+m']] ii=intlamka = l +
                                       m
124
125                         if(len(dic)>0):pt[i]=dic
126
127
128
129             tmp[1]=pt
130         return tmp
131
132 ###################### Method ######################
133
134     def method(self):
135         tmp_lst=[]
136         for i in self.lst:
137             string = ""
138             output = []
139             output_dict={}
140             for j in i: string += j
141             #print("check",string)
142             lst00=self.pattern_match(string)
143             #print("check after", lst00)
144             #print("-----------------------------------")
145             filteration([]).reemovNestings(lst00,output,
                    output_dict)
146             if(len(output_dict)>0): output.append(
                    output_dict)
147
148             tmp_lst+= [lst00]
149         return tmp_lst
```

Modification:

```
1
2 from filteration import filteration
3 class modification(object):
4     def __init__(self,lst):
```

```
5           self.lst=lst
6       def separation(self,string):
7           keyword=['int','longlongint','unsignedlonglongint
               ','signedlonglongint'
8                    'string','float','double','longdouble','
                       char']
9           lst=[]
10          string00=''
11          for i in string:
12              string00+=i
13              if(string00 in keyword):
14                  lst.append(string00)
15                  string00=''
16          if(string00 =='while_loop' or string00=='for_loop
               '):
17              string00='loop'
18          # if (string00 == 'assignE' or string00 == '
               assignBy_1'):
19          #     string00 = 'assign'
20          if(len(string00)>0): lst.append(string00)
21          return lst
22
23      def method(self):
24          lst = []
25          for lst00 in self.lst:
26
27              output = []
28              output_dict={}
29
30              filteration([]).reemovNestings(lst00,output,
                   output_dict)
31              if (lst00[0] == "assignE" or lst00[0] == "
                   assignBY_1"):
32                  for key in output_dict:
33                      tmp_lst=output_dict[key][0]
34                      output[0]='assign'
35                      for ii in tmp_lst:
36                          output.append(ii)
37                  output_dict.clear()
38              #print("OUTPUT :",output)
39              #print("OUTPUT_dict:",output_dict)
40              lst01=[]
41              dict01={}
42              for ii in range(len(output)):
43                  return_lst=filteration([]).filt(output[ii
                       ])
44                  for il in return_lst:
45                      tmp=self.separation(il)
```

```
                    lst01+=tmp

            for ii in output_dict:
                for pp in output_dict[ii]:
                    lst02 = []
                    if(type(pp)==list):
                        tmp01=[]
                        for yy in pp:
                            return_lst=filteration([]).
                               filt(yy)
                            tmp00=[]
                            for il in return_lst:
                                tmp=self.separation(il)
                                tmp00+=tmp
                            tmp01+=tmp00

                        lst02+=tmp01
                    else:
                        lst02 += filteration([]).filt(pp)
                    if(ii not in dict01): dict01[ii]=[]
                    dict01[ii].append(lst02)
            #print("check",lst01,dict01)
            if(len(output_dict)>0): lst01.append(dict01)
            lst.append(lst01)
        return lst
```

Measurement:

```

import calculation
from filteration import filteration
def final_calculation_loop(par_lst):
    if(type(par_lst) == str): return par_lst

    comp='constant'
    for inner in par_lst:
        l1,l2=inner[0],inner[1]
        if(l2 == 'logn'):
            return l2
        if(l2=='n'):
            comp='n'
    return comp
def final_calculation_recursion(par_lst,key):
    if(type(par_lst) == str): return par_lst
    dict={}
    one_par=0
    for inner in par_lst:
```

```
20          if(len(inner)<=1):
21              one_par=one_par+1
22              continue
23          l1,l2=inner[0],inner[1]
24          if(l2 not in dict): dict[l2]=0
25          if(l2!='c'): dict[l2]=dict[l2]+1
26      if (len(par_lst)==one_par): return 'not recursive'
27      if('logn' in dict):
28          if(dict['logn']>=2 and calculation.Function[key
               ]>=2):
29              return "master theorm"
30      return 'not master theorm'
31
32  def is_nested(par_dict):
33      tmp_lst=[]
34      prev1,prev2=0,0
35      prev_tuple=0
36      for key in par_dict: tmp_lst.append(key)
37      for ii in range(len(tmp_lst)):
38          if(ii==0):
39              prev1,prev2=tmp_lst[ii][0],tmp_lst[ii][1]
40              prev_tuple=tmp_lst[ii]
41          else:
42              new1, new2 = tmp_lst[ii][0], tmp_lst[ii][1]
43              if(new1<prev1 and prev2<new2):
44                  comp=par_dict[prev_tuple]
45                  par_dict[tmp_lst[ii]]=par_dict[tmp_lst[ii
                       ]]+'*'+comp
46                  par_dict.pop(prev_tuple)
47              prev1, prev2 = tmp_lst[ii][0], tmp_lst[ii][1]
48              prev_tuple = tmp_lst[ii]
49  def choose_linear_or_logarithmic(par_lst):
50      if('logn' in par_lst):
51          return 'logn'
52      elif ( 'n' in par_lst):
53          return 'n'
54      else:
55          return 'constant'
56  def linear_or_logarithmic(val2,sign,val1,check,par_dict,
       call):
57      ans='n'
58      if(call=='loop'):
59          if (val2 not in calculation.ScopeMemory
60                  and val1 not in calculation.ScopeMemory
61                  and val2 not in par_dict
62                  and val1 not in par_dict
63          ):
64                  ans='c'
```

```
65                  check=True
66          else: ans='n'
67      if(sign=='+' or sign=='-'): return ans
68      else: return 'log'+ans
69
70  def loop(par_dict):
71
72      for key in par_dict:
73          value_of_n = ''
74          check = False
75          for ii in range(len(par_dict[key])):
76              tt = filteration([]).filt(par_dict[key][ii])
77              val2,sign,val1=tt[0],tt[1],tt[2]
78
79              if(ii==0):
80
81                  tm=linear_or_logarithmic(val1,sign,val2,
                        check,par_dict,'loop')
82                  par_dict[key][ii]=tm
83                  if (check): value_of_n='const'
84                  else: value_of_n = key
85              else:
86                  if(val2=='n'): val2=value_of_n
87                  if (val1 == 'n'): val1 = value_of_n
88                  tm = linear_or_logarithmic(val2, sign,
                        val1, check,par_dict,'loop')
89                  par_dict[key][ii] = tm
90          par_dict[key]=choose_linear_or_logarithmic(
              par_dict[key])
91      lst=[]
92      for key in par_dict:
93          lst.append([key,par_dict[key]])
94      return lst
95
96
97  def recursion(par_lst):
98
99      for inner in par_lst:
100         if(len(inner)<=1): continue
101         lst=inner[1]
102         if(type(lst)==str):
103             inner[1]='c'
104         else:
105
106             for ii in range(len(lst)):
107                 tt = filteration([]).filt(lst[ii])
108
109               val2,sign,val1=tt[0],tt[1],tt[2]
```

```
110             tm = linear_or_logarithmic(val1, sign, val2,
                    False,'recusion')
111                 lst[ii] = tm
112             inner[1]=choose_linear_or_logarithmic(lst)
113
114             #par_dict[key]=choose_linear_or_logarithmic(
                    par_dict[key])
```

Input file:

```
1  from file_control import file_control
2  print("Pls␣give␣your␣c++␣code:")
3
4  #filename=open("binary Search.txt")
5  #output=file_control(filename)
6  lines = []
7  with open('filename.txt', 'w') as gfg:
8      while True:
9          line = input()
10         if line == 'END':
11             break
12
13         gfg.write(line+'\n')
14 InputFile=open("filename.txt")
15 output=file_control(InputFile)
16 print(output.method())
```

Infix to Postfix:

```
1  def prec(c):
2      if c == '^':
3          return 3
4      elif c == '/' or c == '*':
5          return 2
6      elif c == '+' or c == '-':
7          return 1
8      else:
9          return -1
10
11 def associativity(c):
12     if c == '^':
13         return 'R'
14     return 'L'
15
16 def infix_to_postfix(s):
```

```
17      result = []
18      stack = []
19      string=''
20      tmp_str=''
21      i=0
22
23      while(i<len(s)):
24          c = s[i]
25          if(('a' <= c <= 'z') or ('A' <= c <= 'Z') or ('0'
                <= c <= '9')):
26              while (('a' <= s[i]<= 'z')
27                  or ('A' <= s[i] <= 'Z')
28                     or ('0' <= s[i] <= '9')):
29                  tmp_str+=s[i]
30                  i=i+1
31                  if(i==len(s)): break
32              result.append(tmp_str)
33              tmp_str=''
34              i=i-1
35          elif c == '(':
36              stack.append(c)
37          elif c == ')':
38              while stack and stack[-1] != '(':
39                  result.append(stack.pop())
40              stack.pop()  # Pop '('
41          else:
42              while stack and (prec(s[i]) < prec(stack[-1])
                    or
43          (prec(s[i]) == prec(stack[-1]) and associativity(
              s[i]) == 'L')):
44                  result.append(stack.pop())
45              stack.append(c)
46          i=i+1
47
48
49      while stack: result.append(stack.pop())
50      for i in result: string+=i+'␣'
51      return string[:-1]
```

Filtration:

```
1
2   class filteration(object):
3       def __init__(self,lst):
4           self.lst=lst
5
6       ###################### method ##################
```

```
7       def reemovNestings(self,M_lst,output,output_dict):
8           for ii in M_lst:
9               if type(ii) == list:
10                  self.reemovNestings(ii,output,output_dict
                        )
11              elif type(ii) ==tuple:
12                  self.reemovNestings(ii, output,
                        output_dict)
13              elif type(ii) ==dict:
14                  for pp in ii:
15                      if(pp not in output_dict and pp!='
                            assignE' and pp!='assignBY_1'):
16                          output_dict[pp]=[]
17                      for qq in ii[pp]:
18                          if(pp=='assignE' or pp=='
                                assignBY_1'):
19                              if ('assign' not in
                                    output_dict): output_dict[
                                    'assign'] = []
20                              output_dict['assign'].append(
                                    qq)
21                          else: output_dict[pp].append(qq)
22              elif type(ii) == str :
23                  output.append(ii)
24
25      ###################### method ##################
26      def filt(self,string):  # separating sign and word
27
28          return_lst = []
29          tmp_string = ""
30
31          for i in range(len(string)):
32
33              chr=string[i]
34              if ("a" <= chr <= "z" or "A" <= chr <= "Z" or
                    "0" <= chr <= "9" or chr == "[" or chr ==
                    "]" or chr=='_'):
35                  tmp_string += chr
36
37              else:  # only char and word insert
38                  if (tmp_string != ""): return_lst.append(
                        tmp_string)
39                  tmp_string = ""
40                  return_lst.append(chr)
41
42          if (tmp_string != ""):
43              return_lst.append(tmp_string)  # if in the
                    last their not any sign char
```

```
44          return return_lst
45
46      ###################### method ##################
47      def method(self):
48          string = ""
49          lst = []
50          temporary_lst = []
51          for x in self.lst:
52              for i in x:
53                  if (i == "\n"):  # line
54                      if (string != ""): lst.append(string)
55
56                      if (len(lst) != 0 and lst[0][0] != "/
                            "):  # deleting comment and empty
                            lines
57                          temporary_lst.append(lst)
58
59                      lst = []  # remove previous line &&
                            string
60                      string = ""
61                  else:
62                      if (i == "␣"):  # string formatting
63                          if (string != ""):
64
65                              lst.append(string)
66                          string = ""
67                      elif (i == "\t"):
68                          string += ""
69                      else:
70                          if('A'<=i<='Z' or 'a'<=i<='z' or
                                '0'<=i<='9' or i=='_'):
71                              string += i
72                              #if (i == '_'): print(i,
                                    string)
73                          else:
74                              #if(i=='_'): print(i)
75                              if(len(string)>0): lst.append
                                    (string)
76                              string=""
77                              lst.append(i)
78
79          temporary01_lst = []
80          for x in temporary_lst:
81
82              return_lst = []
83              for y in x:
84
85                  return_lst = return_lst + self.filt(y)
```

```
86
87              if(return_lst[-1]==';'):temporary01_lst.
                   append(return_lst[:-1])
88              else: temporary01_lst.append(return_lst)
89
90          return temporary01_lst
```

File control:

```
1
2  from filteration import filteration
3  from boundary import boundary
4  from pattern import pattern
5  from modification import modification
6  from calculation import calculation
7  import measurment
8  class file_control(object):
9      def __init__(self,lst):
10         self.lst=lst
11     def method(self):
12
13         filter_lst=filteration(self.lst).method()
14         lst1,boundaries=boundary(filter_lst).method()
15         pattern_lst = pattern(lst1).method()
16         modify=modification(pattern_lst).method()
17         Complexity, FunctionVariable, Function =
              calculation(modify,boundaries).method()
18         for key in Complexity:
19             first,last=key[0],key[1]
20             print("--------------Loop␣Analysis
                  --------------")
21             print('>>>>␣scope:␣',key)
22             print('>>>>␣The␣complexity␣is','O('+
                  Complexity[key]+')')
23             while(first<=last):
24                 print(lst1[first])
25                 first=first+1
26             print()
27
28         print("--------------Recursive␣Analysis
              --------------")
29         for key in FunctionVariable:
30             print(key,'is',FunctionVariable[key])
31         print()
32
33         print("--------------Numeber␣of␣function␣called
              --------------")
```

```
        for key in Function:
            if(Function[key]==0):
                if(key=='main'):print(key, 'is␣Main␣
                   function')
                else: print(key, 'is␣built␣in')
            else:
                print(key, 'is␣called',Function[key],'
                   time')
        print()
        # print(boundaries)
        # for i in boundaries:
        #     print(i)
        #     for j in range(i[0],i[1]+1):
        #         print(j,pattern_lst[j])
        #     print()
        ##################
```

Calculation:

```

MainMemory = {}
Complexity={}
LoopVariable={}
ScopeMemory={}
FunctionVariable={}
Function={}

keyword=['int','longlongint','unsignedlonglongint','
   signedlonglongint','string','float','double','
   longdouble','char']
operator = ['<', '>', '=', '&', '!', '|']
brackets=[',','(',')','{','}']
arithmetic=['+','-','/','*','%','^']
from filteration import filteration
import Infix_to_postfix
from Postfix_Expression import Postfix_Expression
import measurment
class calculation(object):

    def __init__(self,lst,boundaries):
        self.lst=lst
        self.boundaries=boundaries

################## Method ###################
    def clear_extra_plus_minus(self, par_string):
        string = ""
        for i in range(len(par_string)):
```

```
27              if (i == len(par_string) - 1):
28                  if (par_string[i] != '-' or par_string[i]
                        != '+'):
29                      string += par_string[i]
30
31                  break
32              if (par_string[i] == '+' and par_string[i +
                    1] == '+'): continue
33              if (par_string[i] == '+' and par_string[i +
                    1] == '-'): continue
34
35              string += par_string[i]
36
37          if (len(string) > 0 and string[-1] == '+'):
                string = string[:-1]
38          return string
39
40  ############## Method ############
41      def lst_to_string(self,par_lst):
42          tmp_lst = ""
43          for inner in par_lst:
44              tmp_lst=tmp_lst+inner
45          return tmp_lst
46
47  ############# Method #############
48      def nested_to_linear(self, par_lst):
49          output = []
50          output_dict = {}
51          filteration([]).reemovNestings(par_lst, output,
                output_dict)
52          return output
53
54  ############### Method ############
55  # If I have time , I will consider this method
56  # def go_to_last_variable_in_ScopeMemory(self,key,par_lst
        ,main,visited):
57  #
58  #     par_lst = self.nested_to_linear(par_lst)
59  #     visited.append(key)
60  #     #print("check", key, par_lst)
61  #     for kk in range(len(par_lst)):
62  #         if par_lst[kk] in ScopeMemory and par_lst[kk]
        not in visited:
63  #             #print("----",key,par_lst )
64  #             self.go_to_last_variable_in_ScopeMemory(
        par_lst[kk],ScopeMemory[par_lst[kk]],visited)
65  #             print("----",par_lst[kk], ScopeMemory[
        par_lst[kk]],par_lst)
```

```
66 #              par_lst[kk] = ScopeMemory[par_lst[kk]]
67 #     #print("check", key, par_lst)
68 #     return par_lst
69 ############### Method #########
70     def return_statment(self,par_lst):
71         for inner in par_lst:
72             for jj in range(len(inner)):
73                 if inner[jj] in Function:
74                     Function[inner[jj]]=Function[inner[jj
                           ]]+1
75                     a=inner[jj:].index('(')+jj+1
76                     b=inner[jj:].index(')')+jj
77                     tmp_str = ''
78                     while (a < b):
79                         tmp_str += inner[a]
80                         a += 1
81                     tmp_str = tmp_str.split(',')
82                     function_name=inner[jj]
83                     if (function_name not in MainMemory):
                           MainMemory[function_name] = []
84                     MainMemory[function_name].append(
                           tmp_str)
85
86
87
88
89 ############### Method ##############
90     def function_learning(self):
91         for key in Function:
92             if key in MainMemory and key in
                   FunctionVariable:
93                 for outer in MainMemory[key]:
94                     for inner in range(len(outer)):
95                         FunctionVariable[key][inner].
                               append(outer[inner])
96
97         for key in FunctionVariable:
98             for ii in range(len(FunctionVariable[key])):
99                 string = ''
100                outer=FunctionVariable[key][ii]
101                check=[]
102                for inner in range(len(outer)):
103                    if inner==0:
104                        check.append(outer[inner])
105                        continue
106                    tmp_lst=filteration([]).filt(outer[
                           inner])
107                    for kk in range(len(tmp_lst)):
```

```
108                         if(tmp_lst[kk] in MainMemory):
109                             tmp_lst[kk]=MainMemory[
                                    tmp_lst[kk]]
110
111                     tmp=self.nested_to_linear(tmp_lst)
112                     tmp=self.lst_to_string(tmp)
113                     tmp=self.clear_extra_plus_minus(tmp)
114                     string+=tmp+'+'
115                 if(len(string)>0):
116                     tt=Infix_to_postfix.infix_to_postfix(
                            string[:-1]).split('␣')
117                     tt = Postfix_Expression().centralfunc
                            (tt, outer[0])
118                     if(len(tt)>0):
119                         FunctionVariable[key][ii]=[outer
                                [0],tt]
120                     else:
121                         FunctionVariable[key][ii] = [
                                outer[0],string[:-1] ]
122
123
124 ################ Method #################
125     def loop_learning(self):
126
127         if (len(LoopVariable) != 0):
128             for key in LoopVariable:
129                 if key in ScopeMemory:
130                     LoopVariable[key] = ScopeMemory[key]
131
132         for key in LoopVariable:
133             tmp_lst = LoopVariable[key]
134             output = self.nested_to_linear(tmp_lst)
135             tmp_lst = output
136             # exp=tmp_lst
137             # a=self.go_to_last_variable_in_ScopeMemory(
                    key,exp,exp,[])
138             # print("exp",a)
139             for kk in range(len(tmp_lst)):
140                 if tmp_lst[kk] in ScopeMemory:
141                     tmp_lst[kk] = ScopeMemory[output[kk]]
142             output=self.nested_to_linear(tmp_lst)
143             tmp_str = self.lst_to_string(output)
144             tmp_str=self.clear_extra_plus_minus(tmp_str)
145             tt=Infix_to_postfix.infix_to_postfix(tmp_str)
                    .split("␣")
146             tt=Postfix_Expression().centralfunc(tt,key)
147             LoopVariable[key]=tt
148             #print(key, LoopVariable[key])
```

```
149
150 ############## Method #################
151     def formation(self, par_lst):
152         if (par_lst[0] == 'assign'):
153             if (par_lst[1] not in ScopeMemory):
                    ScopeMemory[par_lst[1]] = []
154             if (par_lst[1] not in MainMemory): MainMemory
                    [par_lst[1]] = []
155             MainMemory[par_lst[1]] += [par_lst[3:]] + ['+
                    ']
156             ScopeMemory[par_lst[1]] += [par_lst[3:]] + ['
                    +']
157
158         elif (par_lst[0] == 'return'):
159             if ('return' not in ScopeMemory): ScopeMemory
                    ['return'] = []
160             if (par_lst[1] not in MainMemory): MainMemory
                    ['return'] = []
161             MainMemory['return'] += [par_lst[2:]]
162             ScopeMemory['return'] += [par_lst[2:]]
163
164         elif (par_lst[0] == 'loop'):
165             par_lst_dict = par_lst[2]
166             if ('compare' in par_lst_dict):
167                 for outer_lst in par_lst_dict['compare']:
168                     for ii in outer_lst:
169                         if (ii not in operator and ii.
                                isdigit() == False):
170                             LoopVariable[ii] = []
171             if ('assign' in par_lst_dict):
172                 for inner_lst in par_lst_dict['assign']:
173                     if (inner_lst[0] not in LoopVariable)
                            : LoopVariable[inner_lst[0]] = []
174                     LoopVariable[inner_lst[0]] += [
                            inner_lst[2:]] + ['+']
175
176         elif (par_lst[0] == "function_with_curly_braces")
                :
177             if (par_lst[1] not in Function): Function[
                    par_lst[1]] = 0
178
179             FunctionVariable[par_lst[1]] = []
180             for ii in range(len(par_lst)):
181                 if ii == 0 or ii == 1: continue
182                 if (par_lst[ii] not in keyword
183                         and par_lst[ii] not in operator
184                         and par_lst[ii] not in brackets
185                         and par_lst[ii] not in arithmetic
```

```
186                 ):
187                     FunctionVariable[par_lst[1]].append([
                            par_lst[ii]])
188
189         elif (par_lst[0] == "funtion"):
190             function_name = par_lst[1]
191             a = par_lst.index('(') + 1
192             b = par_lst.index(')')
193             tmp_str = ''
194             while (a < b):
195                 tmp_str += par_lst[a]
196                 a += 1
197             tmp_str = tmp_str.split(',')
198             if (function_name not in MainMemory):
                    MainMemory[function_name] = []
199             if (function_name not in Function):
200                 Function[function_name] = 0
201             MainMemory[function_name].append(tmp_str)
202             Function[function_name] = Function[
                    function_name] + 1
203 ################## Method ##################
204     def method(self):
205         visited=[]
206
207         for i in range(len(self.lst)):
208             visited.append(0)
209         for i in self.boundaries:
210             first,last=i[0],i[1]
211             check_if_forloopE=False
212             check_if_loop = False
213
214             ## each function will clear the memory
215             if(self.lst[first][0]=="for_loopE"):
                    check_if_forloopE=True
216             if (self.lst[first][0] == "loop"):
217                 check_if_loop = True
218
219             while(first<=last):
220                 if(visited[last]==0):
221                     self.formation(self.lst[last])
222                     #print(self.lst[last])
223                 visited[last]=1
224                 last=last-1
225
226             # print("CodeMemory: ",ScopeMemory)
227             # print("loopVariable: ",LoopVariable)
228             # print("functionVariable: ",
                    FunctionVariable)
```

```
229             # print("Function: ", Function)
230             #print()
231             tmp=''
232 
233             if (check_if_loop):
234                 self.loop_learning()
235                 Complexity[(i[0],i[1])]=measurment.loop(
                        LoopVariable)
236                 LoopVariable.clear()
237                 ScopeMemory.clear()
238 
239             ##for(int i:mp)
240             if (check_if_forloopE):
241                 Complexity[i[0]]='n'
242 
243 
244         if ("return" in MainMemory): self.return_statment
                (MainMemory["return"])
245         self.function_learning()
246         for key in FunctionVariable:
247             measurment.recursion(FunctionVariable[key])
248 
249         #print("functionVariable: ", FunctionVariable)
250         # print("function: ", Function)
251         #print("DDComplexity: ", Complexity)
252         MainMemory.clear()
253         for key in Complexity:
254             Complexity[key]=measurment.
                    final_calculation_loop(Complexity[key])
255         measurment.is_nested(Complexity)
256         for key in FunctionVariable:
257             FunctionVariable[key] = measurment.
                    final_calculation_recursion(
                    FunctionVariable[key],key)
258         print("DDComplexity:␣", Complexity)
259         # print("functionVariable: ", FunctionVariable)
260         return Complexity,FunctionVariable,Function
```

Boundaries

```
1 
2 class boundary(object):
3     def __init__(self,lst):
4         self.lst=lst
5     def method(self):
6         stack = []
7         tmp_lst=[]
```

```
8           cnt = 0
9           mn = 10000000
10          mx = 0
11          for i in self.lst:
12
13              if ("{" in i):
14
15                  if (len(i) == 1 and i[0] == '{'):
16                      stack.append(cnt-1)
17                  else:
18                      stack.append(cnt)
19
20              if ("}" in i):
21
22                  if (len(stack) > 0):
23                      mn = min(mn, stack[-1])
24                      stack.pop()
25                  mx = max(mx, cnt)
26
27              cnt += 1
28
29          # If only consider only scoping line , we have to
                find min boundary and max boundary.
30          # In c++ or java code , most of the thing are
               done in scoping
31          # otherthings are include , define , const which
               will delete by below for loop
32
33          boundaries=[]
34          stk=[]
35          cnt_idx=0
36
37          for i in range(mn, mx+1):
38
39              if ("cout" not in self.lst[i] and "cin" not
                   in self.lst[i]):
40
41                  tmp_lst.append(self.lst[i])
42              else: cnt_idx-=1
43              if ("{" in self.lst[i]):
44                  if(len(self.lst[i])==1 and self.lst[i
                       ][0]=='{'):
45                      stk.append(cnt_idx-1)
46                  else:
47                      stk.append(cnt_idx)
48              if ("}" in self.lst[i]):
49
50                  if (len(stk) > 0):
```

```
51
52                      boundaries.append([stk[-1],cnt_idx])
53                      stk.pop()
54              cnt_idx+=1
55          return tmp_lst,boundaries
```