****

**MINISTRY OF EDUCATION, CULTURE AND RESEARCH**

**OF THE REPUBLIC OF MOLDOVA**

**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**Department of Software and Automation Engineering**

**Denis Pleșca**

**Report**

**Individual Work**

***of the "Data Structures and Algorithms" course***

Checked:

**Burlacu Natalia,** *PhD, associate professor*

Department of Software and Automation Engineering,

FCIM Faculty, UTM

**Chisinau – 2024**

**Table of Contents**

1. **Introduction**

* **Problem Description**
* **Time Complexity Description**
* **Problem Statement**
* **Solution Proposal**

1. **System Overview**
2. **Conclusion**
3. **The Website**

* **Running the programm**

1. **Input/Output examples**
2. **Appendix**
3. **Introduction**

**1.1 Problem Description.**

In software development, ensuring that an algorithm performs efficiently is a fundamental concern. As the size of the input data grows, the performance of an algorithm can significantly impact the overall system performance. Developers often need to analyze and compare the time complexities of different algorithms to select the most efficient one. Manually computing the time complexity of code is a tedious and error-prone process, requiring detailed analysis and deep understanding of algorithms, which becomes increasingly challenging for large and complex codebases. This often leads to inaccuracies and inefficiencies in performance optimization.

**1.2 Time complexity description.**

**Time complexity** is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to be related by a constant factor.

Since an algorithm's running time may vary among different inputs of the same size, one commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size. This is also the type of time complexity that I will use in my individual work, since it is a lot easier to compute. Less common, and usually specified explicitly, is the average-case complexity, which is the average of the time taken on inputs of a given size (this makes sense because there are only a finite number of possible inputs of a given size).

|  |
| --- |
| **1.3 Problem Statement :** Manually computing the time complexity of code is a time consuming and error prone process, especially for large and complex codebases |

**1.4 Solution Proposal.**  
 If computing the time complexity manually creates problems and errors, why not creating an algorithm / script that will do these for us, we live in the technological era after all. Although these can ask a lot of time and work to be implemented, it will pay off all the efforts. There are several ways we can accieve this:

a) *Giving different inputs to the code, assuming their complexity increases, and , after this, analyze the system time needed to receive the answer.*

*b)Analyzing the structure of the code itself and finding all the parts that influence the time complexity the most(nested loops and their conditions, recursive functions )*

Every one of these two approaches has their advantages and disadvantages.

a)The first one although seems quite fast has a big problem: How we must know what input the code requires and how to scale it. In this case, it's not even clear what to do if you don't have a whole code, but only a function or a few.

b)Analyzing the structure of the code is a lot easier, because for us it does not even matter is it a working code or if it has some bugs. But there appears the problem which does not allow us that it is a lot better then the first: It is slow and hard to get exact results, because the behavior of a program can change a lot based on the input, for example if-else statements.

In both algorithms we can seek help from Artificial Intelligence and it tips the scales slightly towards the first option.

**But, because of the concernes of the task(requiring using C/C++ and the time limit) which limits me from using AI I will preffer the method of analyzing the syntax.**

|  |
| --- |
| **Solution proposal:** An C algorithm that analyzes the syntax of the code and based on the structure to return the complexity of the program. |

1. **System overview**

In this section, I provide a detailed overview of the system architecture designed to automate the process of time complexity analysis. The Time Complexity Analyzer is implemented in C and consists of several key components, each responsible for different stages of the analysis process. By modularizing the system, we ensure that each component focuses on a specific aspect of the task, making the overall system more efficient, maintainable, and extensible. Below, we describe the core components and their roles within the system

**System Architecture**

The Time Complexity Analyzer is structured into four main components: the Reader, Lexer, Parser, and Interpreter. Each component plays a critical role in transforming source code into a time complexity analysis.

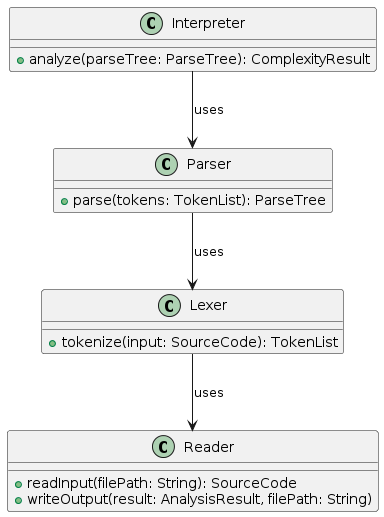


Figure 1 - **Overall Conceptual Model**

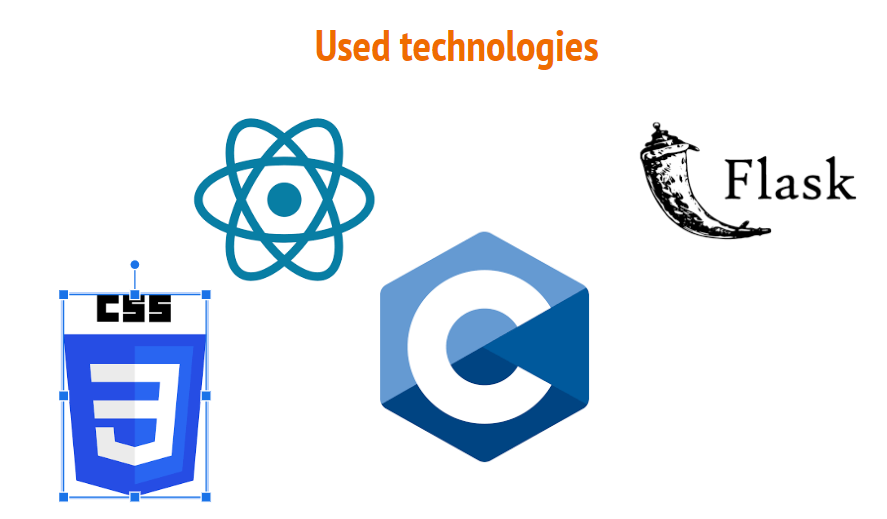
1. **Reader**
   * **Purpose**: The Reader module manages input and output operations.
   * **Functions**:
     + **get\_file\_contents(filePath: String): SourceCode**: Reads the source code from a specified file path and returns it as a string.
   * **Role**: The Reader module serves as the interface between the file system and the analyzer. It ensures the correct reading of the source code from input files.
2. **Lexer**
   * **Purpose**: The Lexer module tokenizes the input source code.
   * **Functions**:
     + **init\_lexer(input: SourceCode): TokenList**: Initializes the lexer with the contents of the file as the value of lexer->input
     + **lexer\_advance(lexer\_T\* lexer): void** : gets the next character from the input code
     + **lexer\_advance\_with(lexer\_T\* lexer, char\* value, int (\*predicate)(int)): String** : creates a token of the value type.
   * **Role**: The Lexer scans the source code and breaks it down into a sequence of tokens. These tokens represent the basic elements (keywords, operators, identifiers, etc.) needed for syntactic analysis.
3. **Parser**
   * **Purpose**: The Parser module constructs a structured representation of the code from the tokens produced by the Lexer.
   * **Functions**:
     + **parse(tokens: TokenList): ParseTree**: Analyzes the sequence of tokens and generates a parse tree.
   * **Role**: The Parser organizes the tokens into a parse tree, which represents the grammatical structure of the source code. This structured representation is essential for the subsequent complexity analysis.
4. **Interpreter**
   * **Purpose**: The Analyzer module evaluates the parse tree to determine the time complexity of the code.
   * **Functions**:
     + **analyze(parseTree: ParseTree): ComplexityResult**: Evaluates the parse tree and computes the time complexity of various code segments.
   * **Role**: The Analyzer examines the parse tree to identify computational complexity. It computes the time complexity and provides a detailed analysis of the performance characteristics of the code.

The full version of the code can be seen on my github.[1]

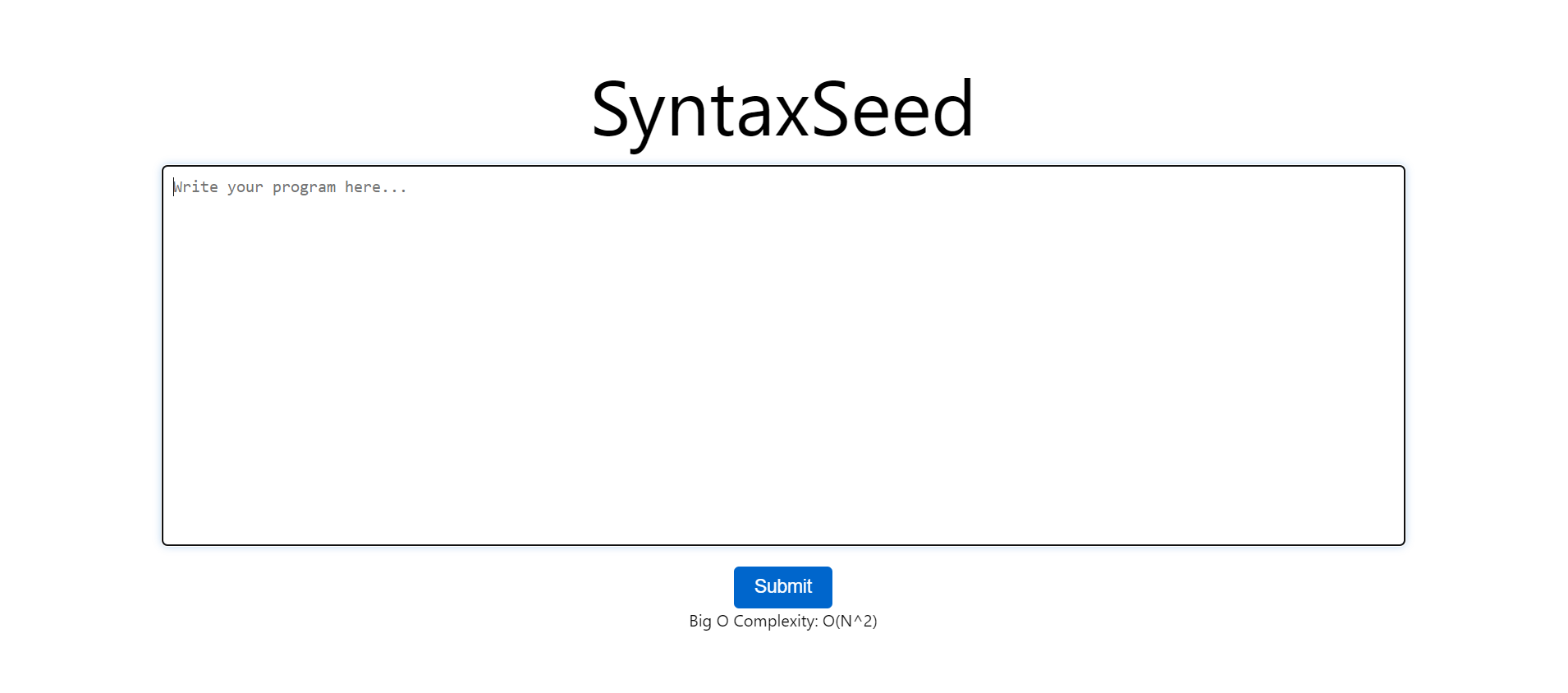
This program is a working demo of my idea. A more advanced version(although not a working one) can be seen also on my github.[2]

**4.The website**

Till now I have shown only the algorithm written in C for the analysis of the structure of the code. Excepting it my solution also contains an server written in python/flask and a website written in React and Css

****

**Figure 2 – Used Technologies**

****

**Figure 3 – Website mockup**

As it can be seen from the Figure 3 the website has a minimalistic design without any useless buttons and functions. There are no logging procedees because there is not data that needs to be saved for a particular user.

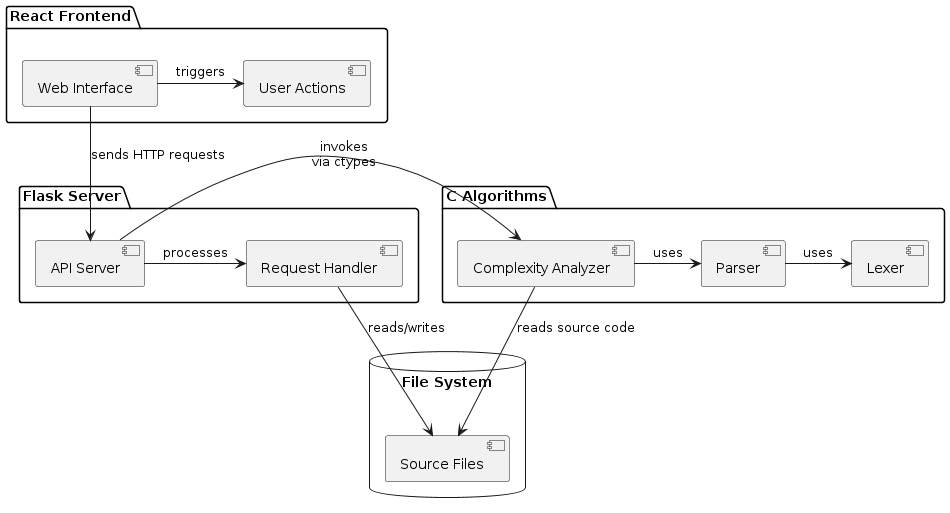


Figure 4 – Component diagram

In Figure 4 it is shown the overall logic that stands behind the process of analyzing the input code.The website, triggered by the User Actions, sends an HTTP request containing the input code to the server. The server processes the request and writes the contents in a separate file(it was created to simplify the connection between the server and C algorithm).After this the Algorithm that was explained above, triggered by the API Server analysis the program and sends back, all the way to the Web Interface, the result conyaining the BigO notation of the given program.

**4.1 Running the Programm**

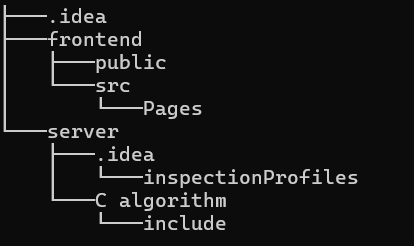
****

Figure 5 – Repository tree

In the Figure 5 is represented the github Repository Tree. To succesfully run the react website open in cmd the „frontend” folder and run *npm start*.To run the python server navigate to the server folder and run the app.py file. Make sure you have installed react,react-router-dom and the necesary librarys for python

**6.Appendix**

**Include/Lexer.h**

#ifndef LEXER\_H

#define LEXER\_H

typedef struct {

    char\* input;

    int pos;

    int len;

    char c;

} lexer\_T;

lexer\_T\* init\_lexer(char\* input);

void lexer\_advance(lexer\_T\* lexer);

char lexer\_peek(lexer\_T\* lexer, int offset);

void lexer\_skip\_whitespace(lexer\_T\* lexer);

char\* lexer\_collect\_number(lexer\_T\* lexer);

char\* lexer\_advance\_with(lexer\_T\* lexer, char\* value, int (\*predicate)(int));

#endif

**Lexer.c**

**#include "include/lexer.h"**

**#include <stdlib.h>**

**#include <ctype.h>**

**#include <string.h>**

**lexer\_T\* init\_lexer(char\* input) {**

**lexer\_T\* lexer = calloc(1, sizeof(lexer\_T));**

**lexer->input = input;**

**lexer->pos = 0;**

**lexer->len = strlen(input);**

**lexer->c = input[lexer->pos];**

**return lexer;**

**}**

**void lexer\_advance(lexer\_T\* lexer) {**

**if (lexer->pos < lexer->len && lexer->c != '\0') {**

**lexer->pos++;**

**lexer->c = lexer->pos < lexer->len ? lexer->input[lexer->pos] : '\0';**

**}**

**}**

**char lexer\_peek(lexer\_T\* lexer, int offset) {**

**return lexer->pos + offset < lexer->len ? lexer->input[lexer->pos + offset] : '\0';**

**}**

**void lexer\_skip\_whitespace(lexer\_T\* lexer) {**

**while (lexer->c == ' ' || lexer->c == '\t' || lexer->c == '\n') lexer\_advance(lexer);**

**}**

**char\* lexer\_collect\_number(lexer\_T\* lexer) {**

**return lexer\_advance\_with(lexer, calloc(1, sizeof(char)), isdigit);**

**}**

**char\* lexer\_advance\_with(lexer\_T\* lexer, char\* value, int (\*predicate)(int)) {**

**while (lexer->c != '\0' && predicate((int)lexer->c)) {**

**char str[2] = {lexer->c, '\0'};**

**strcat(value, str);**

**lexer\_advance(lexer);**

**}**

**return value;**

**}**

**Include/parser.h**

#ifndef PARSER\_H

#define PARSER\_H

#include "lexer.h"

typedef struct {

    lexer\_T\* lexer;

} parser\_T;

parser\_T\* init\_parser(lexer\_T\* lexer);

int parse\_for\_loops(parser\_T\* parser);

#endif

**Parser.c**

#include "include/parser.h"

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

parser\_T\* init\_parser(lexer\_T\* lexer) {

    parser\_T\* parser = calloc(1, sizeof(parser\_T));

    parser->lexer = lexer;

    return parser;

}

int parse\_for\_loops(parser\_T\* parser) {

    int depth = 0;

    lexer\_skip\_whitespace(parser->lexer);

    while (parser->lexer->c != '\0') {

        if (parser->lexer->c == 'f') {

            char\* potential\_for = lexer\_advance\_with(parser->lexer, calloc(1, sizeof(char)), isalpha);

            if (strcmp(potential\_for, "for") == 0) {

                depth++;

            }

            free(potential\_for);

        }

        lexer\_advance(parser->lexer);

        lexer\_skip\_whitespace(parser->lexer);

    }

    return depth;

}

**Include/read.h**

#ifndef READ\_H

#define READ\_H

char\* get\_file\_contents(const char\* filepath);

#endif

**Read.c**

#include "include/read.h"

#include <stdlib.h>

#include <stdio.h>

char\* get\_file\_contents(const char\* filepath)

{

    char\* buffer = 0;

    long length;

    FILE\* f = fopen(filepath, "r");

    if (f)

    {

        fseek(f, 0, SEEK\_END);

        length = ftell(f);

        fseek(f, 0, SEEK\_SET);

        buffer = calloc(length, sizeof(char));

        if (buffer)

            fread(buffer, 1, length, f);

        fclose(f);

        return buffer;

    }

    printf("Error reading file %s\n", filepath);

    exit(2);

}

**Include/main.h**

**//**This file is empty because I use Makefile to compile all of the functions and it needs for every .c file to exist a .h file.

**main.c**

#include "include/lexer.h"

#include "include/parser.h"

#include "include/read.h"

#include <stdio.h>

int main() {

    char\* source\_code = get\_file\_contents("./received\_contents.txt");

    lexer\_T\* lexer = init\_lexer(source\_code);

    parser\_T\* parser = init\_parser(lexer);

    int depth = parse\_for\_loops(parser);

    printf("%d",depth);

    return depth;

}