**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

До захисту допущено

Завідувач кафедри

_________ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)

"___" __________ 202___ р.

# Дипломний проєкт

**на здобуття ступеня бакалавра**

**за освітньо-професійною програмою «Спеціалізовані комп'ютерні системи»**

спеціальності **123 «Комп'ютерна інженерія»**

на тему: Система визначення часової складності програмних алгоритмів

Виконав (-ла):

студент (-ка) IV курсу, групи КВ-63
2(шифр групи)

Чорний Єгор Геннадійович
(прізвище, ім'я, по батькові) (підпис)

Керівник ас. каф. СПСКС Коляда К.В.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю, доц.каф.СПСКС, к.т.н. Клятченко Я.М.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали ) (підпис)

Рецензент ____________________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент ____________
(підпис)

Київ – 2020 року

**ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ**

| № з/п | Формат | Позначення | Найменування | Кількість листів | Примітка |
|---|---|---|---|---|---|
| 1 | А4 | | Завдання на дипломний проєкт | 2 | |
| 2 | А4 | ДП 045490. 00.000 ПЗ | Пояснювальна записка | 50 | |
| 3 | А1 | ДП 045490. 01.000 ТК | Система визначення часової складності програмних алгоритмів. Схема архітектури програми. Схема структурна | 1 | |
| 4 | А1 | ДП 045490. 02.000 ТК | Система визначення часової складності програмних алгоритмів. Схема потоку виконання програми. Схема структурна | 1 | |
| 5 | А1 | ДП 045490. 03.000 ТК | Система визначення часової складності програмних алгоритмів. Ієрархія правил розробленої граматики. Схема структурна | 1 | |
| 6 | А1 | ДП 045490. 04.000 ТК | Система визначення часової складності програмних алгоритмів. Обчислення часової складності. Схема алгоритму | 1 | |

| | ПІБ | Підп. | Дата | ДП 045490 00.000.00 | | |
|---|---|---|---|---|---|---|
| Розробн. | Чорний Є.Г. | | | Відомість дипломного проєкту | Лист | Листів |
| Керівн. | Коляда К.В. | | | | 1 | 1 |
| Консульт. | | | | | КПІ ім. Ігоря Сікорського Каф. СПіСКС Гр. КВ-63 | |
| Н/контр. | Клятченко Я.М. | | | | | |
| Зав.каф. | Романкевич В.О. | | | | | |

# Пояснювальна записка
## до дипломного проєкту

на тему: Система визначення часової складності програмних алгоритмів _

Київ – 2020 року

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
______ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)

«___» __________20___ р.

**ЗАВДАННЯ**
**на дипломний проєкт студента**

Чорний Єгор Геннадійович ______________________________
(прізвище, ім'я, по батькові)

1. Тема проєкту Система визначення часової складності програмних алгоритмів
______________________________

керівник проєкту ас. каф. СПСКС Коляда К.В. ______________________________ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «25» травня 2020 р. №1181-С

2. Термін подання студентом проєкту 20 травня 2020р.

3. Вихідні дані до проєкту технічна література на тему теорії складності алгоритмів, мови програмування Common Lisp, мови програмування Java. ___

4. Зміст пояснювальної записки складність алгоритмів, теоретичні знання та засоби реалізації, розробка компонентів системи ______________________________

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) схема архітектури програми, схема потоку виконання програми, структурна схема ієрархії правил розробленої граматики, схема алгоритму обчислення часової складності, презентація ____________
______________________________

6. Консультанти розділів проєкту*

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| нормоконтроль | к.т.н., доцент, Клятченко Я.М. | | |
| | | | |
| | | | |

7. Дата видачі завдання 15 грудня 2019 року

Календарний план

| № з/п | Назва етапів виконання дипломного проєкту | Термін виконання етапів проєкту | Примітка |
|---|---|---|---|
| 1 | Видача завдання на дипломне проєктування | 15.12.19 | |
| 2 | Аналіз існуючих рішень | 05.01.20 | |
| 3 | Вибір середовища розробки | 23.01.20 | |
| 4 | Розробка програмного продукту | 03.02.20 | |
| 5 | Відлагодження програмного продукту | 28.03.20 | |
| 6 | Підготовлення пояснювальної записки | 10.04.20 | |
| 7 | Оформлення матеріалів проєкту | 03.05.20 | |
| 8 | Попередній розгляд дипломного проєкту на кафедрі | 20.05.20 | |

Студент ______________ (підпис) ______________________ (Єгор ЧОРНИЙ)

Керівник проєкту ______________ (підпис) ______________________ (Костянтин КОЛЯДА)

* Консультантом не може бути зазначено керівника дипломного проєкту.

# АНОТАЦІЯ

Пояснювальна записка містить 50 сторінки, 2 таблиці, 6 рисунків, 20 джерел.

Об’єкт дослідження – система визначення часової складності програмних алгоритмів.

Предмет дослідження – типи часової складності алгоритмів, методи її визначення та призначення.

Мета роботи – аналіз відомих типів часової складності та методів її визначення, дослідження призначення і можливості використання отриманих даних часової складності алгоритмів, розробка програмного забезпечення для статистичного аналізу алгоритмів і визначення часової складності їх структурних одиниць.

Метод дослідження – вивчення літератури, аналіз відомих типів часової складності та способів її визначення, дослідження статистичного аналізу алгоритмів, середовища створення програмного забезпечення та його розробка, аналіз отриманих результатів і написання висновків.

В процесі розробки були проаналізовані існуючі інструменти для парсинга коду заданої граматики (ANTLR4), використання мови програмування Common Lisp для розробки серверної частини програми, а також використання мови Java для розробки користувацького інтерфейсу. Технічне завдання було розділене на структурні частини для розподілення роботи на етапи. Схема взаємодії програмних модулів була сформована на основі принципів архітектури MVC (Module View Control), що дозволило підвищити ефективність розробки. Дана реалізація програмного комплексу дозволяє досить легко додавати нові модулі, що реалізують підтримку нових можливостей програми.

Розроблена програма приймає на вхід файли з граматикою та кодом, що підлягає аналізу. Для тестування була розроблена тестова граматика. Процес роботи програми розроблений таким чином, що за допомогою вхідної граматики з вхідного коду створюється дерево розбору на якому проводиться

аналіз часової складності після приведення структури дерева розбору до графа потоку керування програми.

Було проведено ряд тестів, практичний результат яких співпав з теоретичними розрахунками.

Ключові слова: алгоритм, обчислювальна складність, часова складність алгоритму, тип складності, граф потоку керування програми, формальна граматика, ANTLR.

## ABSTRACT

The explanatory note contains 50 pages, 2 tables, 6 figures, 20 sources.

The object of research is a system for determining the temporal complexity of software algorithms.

The subject of research - types of time complexity of algorithms, methods of its definition and purpose.

The purpose of the work - analysis of known types of temporal complexity and methods of its determination, study of the purpose and possibility of using the obtained data of temporal complexity of algorithms, development of software for statistical analysis of algorithms and determination of temporal complexity of their structural units.

Research method - studying the literature, analysis of known types of time complexity and methods of its determination, research of statistical analysis of algorithms, software creation environment and its development, analysis of the obtained results and writing conclusions.

During the development process, the existing tools for parsing the code of a given grammar (ANTLR4), the use of the programming language Common Lisp to develop the server part of the program, as well as the use of Java language to develop the user interface were analyzed. The terms of reference were divided into structural parts to divide the work into stages. The scheme of interaction of software modules was formed on the basis of the principles of MVC architecture (Module View

Control), which allowed to increase the efficiency of development. This implementation of the software package allows you to easily add new modules that implement support for new features of the program.

The developed program accepts files with grammar and code to be analyzed. A test grammar was developed for testing.

The operation of the program is designed in such a way that with the help of input grammar from the input code creates a parsing tree, which analyzes the time complexity after bringing the structure of the parsing tree to the graph of the control flow of the program.

A number of tests were conducted, the practical result of which coincided with the theoretical calculations.

Keywords: algorithm, computational complexity, time complexity of the algorithm, type of complexity, program control flow graph, formal grammar, ANTLR.

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045490.002 ТЗ | Система визначення | 4 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Технічне завдання | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.003 ТП | Система визначення | 2 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Відомість технічного | | | |
| | | | проєкту | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.004 ПЗ | Система визначення | 50 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Пояснювальна записка | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.005 Д1 | Система визначення | 1 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Схема архітектури | | | |
| | | | програми | | | |

| Змін. | Арк. | № докум. | Підпис | Дата | ІАЛЦ.045490.001 ОА | | |
|---|---|---|---|---|---|---|---|
| Розробив | | Чорний Є.Г. | | | Система визначення часової складності програмних алгоритмів<br>**Опис альбому** | Літ. | Аркуш: 1 / Аркушів: 2 |
| Перевірив | | Коляда К.В. | | | | | |
| Консульт. | | | | | | НТУУ «КПІ» ім. Ігоря Сікорського, ФПМ, КВ-63 | |
| Н. контроль | | Клятченко Я.М. | | | | | |
| Зав. каф. | | Романкевич К.В. | | | | | |

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045490.006 Д2 | Система визначення | 1 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Схема потоку виконання | | | |
| | | | програми | | | |
| | | | Схема структурна | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.007 Д3 | Система визначення | 1 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Ієрархія правил | | | |
| | | | розробленої граматики | | | |
| | | | Схема структурна | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.008 Д4 | Система визначення | 1 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Обчислення часової | | | |
| | | | складності | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | | Диск CD-ROM | Текст пояснювальної | | | |
| | | | записки. | | | |
| | | | Графічний матеріал | | | |

## Зміст

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ. . . . . . . . . . . . . . . . . . . .2

2. ПІДСТАВА ДЛЯ РОЗРОБКИ. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .2

3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ. . . . . . . . . . . . . . . . . . . . . . . . . . 2

4. ДЖЕРЕЛА РОЗРОБКИ. . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . .2

5. ТЕХНІЧНІ ВИМОГИ. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

  5.1. Вимоги до програмного продукту, що розробляється. . . . . . . . . . . .2

  5.2. Вимоги до апаратного забезпечення. . . . . . . . . . . . . . . . . . . . . . . . . 3

  5.3. Вимоги до програмного та апаратного забезпечення користувача. 3

6. ЕТАПИ РОЗРОБКИ. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

| | | | | | ІАЛЦ.045490.002 ТЗ | | | |
|---|---|---|---|---|---|---|---|---|
| Зм. | Арк. | № докум. | Підп. | Дата | | | | |
| Розроб. | | Чорний Є.Г. | | | Система визнчення часової складності програмних алгоритмів | Літ. | Аркуш | Аркушів |
| Перевір. | | Коляда К.В. | | | | | 1 | 4 |
| | | | | | | КПІ ім. Ігоря Сікорського, ФПМ, КВ-63 | | |
| Н.контр. | | Клятченко Я.М. | | | **Технічне завдання** | | | |
| Затв. | | Романкевич В.О. | | | | | | |

## 1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування роботи – дипломний проєкт на тему «Система визначення часової складності програмний алгоритмів».

Область дослідження: визначення часової складності програмних алгоритмів з метою оптимізації існуючого програмного коду.

## 2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

## 3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ

Метою даної роботи є дослідження часової складності програмних алгоритмів та розробка програмного продукту, що виконує статичний аналіз програмного коду і визначає часову складність його структурних одиниць.

## 4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації в періодичних виданнях та електронні статті у мережі Інтернет.

## 5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до до програмного продукту, що розробляється:

- можливість використання граматики мови користувача
- можливість використання програмного коду користувача
- наявність графічного інтерфейсу користувача;

5.2. Вимоги до апаратного забезпечення:

- Оперативна пам’ять: 2 Гб;

5.3. Вимоги до програмного та апаратного забезпечення користувача:

- операційна система, яка підтримує JVM(Java Virtual Machine).
- збірник проєктів maven.
- утиліта make

## 6. ЕТАПИ РОЗРОБКИ

| № з/п | Назва етапів виконання дипломного проєкту | Термін виконання етапів проєкту | Примітка |
|---|---|---|---|
| 1 | Видача завдання на дипломне проєктування | 15.12.19 | |
| 2 | Аналіз існуючих рішень | 05.01.20 | |
| 3 | Вибір середовища розробки | 23.01.20 | |
| 4 | Розробка програмного продукту | 03.02.20 | |
| 5 | Відлагодження програмного продукту | 28.03.20 | |
| 6 | Підготовлення пояснювальної записки | 10.04.20 | |
| 7 | Оформлення матеріалів проєкту | 03.05.20 | |
| 8 | Попередній розгляд дипломного проєкту на кафедрі | 20.05.20 | |

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045490.004 ПЗ | Система визначення | 50 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Пояснювальна записка | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.005 Д1 | Система визначення | 1 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Схема архітектури | | | |
| | | | програми. | | | |
| | | | Схема структурна | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.006 Д2 | Система визначення | 1 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Схема потоку виконання | | | |
| | | | програми | | | |
| | | | Схема структурна | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.007 Д3 | Система визначення | 1 | | |
| | | | часової складності | | | |

| Змін. | Арк. | № докум. | Підпис | Дата | ІАЛЦ.045490.003 ТП | | |
|---|---|---|---|---|---|---|---|
| Розробив | | Чорний Є.Г. | | | Система визначення часової складності програмних алгоритмів | Літ. | Аркуш 1 / Аркушів 2 |
| Перевірив | | Коляда К.В. | | | | | |
| Консульт. | | | | | **Відомість технічного проєкту** | НТУУ «КПІ» ім. Ігоря Сікорського, ФПМ, КВ-63 | |
| Н. контроль | | Клятченко Я.М. | | | | | |
| Зав. каф. | | Романкевич К.В. | | | | | |

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | | | програмних алгоритмів | | | |
| | | | Ієрархія правил | | | |
| | | | розробленої граматики. | | | |
| | | | Схема структурна | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.008 Д4 | Система визначення | 1 | | |
| | | | часової складності | | | |
| | | | програмних алгоритмів | | | |
| | | | Обчислення часової | | | |
| | | | складності. | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | | Диск CD-ROM | Текст пояснювальної | | | |
| | | | записки. | | | |
| | | | Графічний матеріал | | | |

ЗМІСТ

ВСТУП……………………………………………………………….……………...3
1 СКЛАДНІСТЬ АЛГОРИТМІВ……………………………………………..5
1.1 Пам’ять та час………………………………………………………6
1.2 Часова складність…………………………………………………..7
1.3 Типи часу………………………………………………………...8
1.3.1 Константний час………………………………………..8
1.3.2 Логарифмічний час……………………………………9
1.3.3 Полілогарифмічний час………………………………10
1.3.4 Сублінійний час………………………………………11
1.3.5 Лінійний час…………………………………………...11
1.3.6 Квазілінійний час……………………………………...12
1.3.7 Субквадратичний час…………………………………13
1.3.8 Поліноміальний час…………………………………...13
1.3.9 Суперполіноміальний час……………………………16
1.3.10 Квазіполіноміальний час……………………………16
1.3.11 Субекспоненціальний час…………………………...18
1.3.12 Експоненціальний час………………………………19
1.3.13 Подвійний експоненціальний час…………………..19
1.4 Таблиця складності за часом…………………………….....20
1.5 Статистичне призначення……...…………………………..22
2 ТЕОРЕТИЧНІ ЗНАННЯ ТА ЗАСОБИ РЕАЛІЗАЦІЇ…………………27
2.1 Задання вхідної граматики…………………………………………...27
2.1.1 Види граматик………………………………………………29
2.1.2 ANTLR та необхідна йому граматика……………………..32

| | | | | | ІАЛЦ.045490.004 ПЗ | | | |
|---|---|---|---|---|---|---|---|---|
| Зм. | Лист | № докум. | Підп. | Дата | | | | |
| Розробив | | Чорний Є.Г. | | | Система визначення часової складності програмних алгоритмів | Літ. | Аркуш | Аркушів |
| Перев. | | Коляда К.В. | | | | | 1 | 50 |
| | | | | | | | | |
| Н. контр. | | Клятченко Я.М. | | | | КПІ ім. Ігоря Сікорського, ФПМ, КВ-63 | | |
| Затвер. | | Романкевич В.О. | | | | | | |

3 РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ……………………………….33
3.1 Загальна структура……………………………………………….33
3.1.1 Користувацький інтерфейс…………………………………34
3.1.2 Допоміжний модуль для зв’язку користувацького інтерфейсу та серверної частини………………………...35
3.1.3 Модуль серверної частини……………………………...36
3.2 Використання мови програмування Lisp……………………….36
3.3 Використання мови програмування Java……………………….37
3.4 Вхідні дані……………………………………………………...37
3.5 Створення графу потоку керування програми………………38
3.6 Обчислення складності алгоритмів…………………………..39
3.7 Зв’язок між модулем користувацького інтерфейсу та серверної частини…………………………………………………….40
3.8 Інструкція користувача………………………………………..41
3.9 Тестування розробленої програми та аналіз результатів…...44
ВИСНОВКИ…………………………………………………………….47
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ…………………………...49
ДОДАТКИ
Додаток А. Копії графічних матеріалів
ІАЛЦ045490.005 Д1 Структурна схема архітектури розробленої програми. Схема структурна
ІАЛЦ045490.006 Д2 Схема потоку виконання розробленої програми. Схема структурна
ІАЛЦ045490.007 Д3 Ієрархія правил розробленої граматики. Схема структурна
ІАЛЦ045490.008 Д4 Обчислення часової складності. Схема алгоритму
Додаток Б. Розроблена граматика
Додаток В. Лістинг програмного коду

# ВСТУП

Поняття алгоритму визначається як метод вирішення класу задач на комп'ютері. Кожен алгоритм при виконанні має свою складність, яка вимірюється часом роботи, необхідним обсягом пам'яті, або будь-якими відповідними одиницями, які використовуються для реалізації алгоритму. Кожна задача, яка вирішується таким чином, займає певний проміжок часу. Деякі з проблем займають досить багато часу, інші вирішуються швидше. В такому випадку говорять про обчислювальну складність алгоритму, яку розглядає, в свою чергу, теорія складності обчислень.

Теорія складності обчислень зосереджена на вивченні складності виконання алгоритмів для вирішення задач обчислення. Основним завдання аналізу алгоритмів є визначення складності алгоритму, а також розуміння принципу роботи та зв'язку явищ та дій, які стосуються одного алгоритму. Тому, можна сказати, що теорія складності обчислень ставить перед собою два типи цілей. Перший - абсолютна відповідь щодо конкретних обчислень, другий, в свою чергу, концентрується на питаннях відношень між задачами обчислення.

Актуальність роботи зумовлена постійною необхідністю оптимізації уже існуючих алгоритмів та тих, які розробляються. Визначення обчислювальної складності структурних одиниць (функцій) алгоритму дає можливість аналізу і, відповідно, наступного удосконалення алгоритму для пришвидшення його роботи.

Задача даної роботи - дослідження типів часової складності та способів її визначення, статистичного аналізу алгоритмів, розробка програмного забезпечення, аналіз отриманих результатів і написання висновків.

Мета роботи:

1) аналіз відомих типів часової складності та методів її визначення

2) дослідження призначення і можливості використання отриманих даних часової складності алгоритмів

3) розробка програмного забезпечення для статистичного аналізу алгоритмів і визначення часової складності їх структурних одиниць.

# 1 СКЛАДНІСТЬ АЛГОРИТМІВ

Теорія складності обчислень класифікує обчислювальні задачі та відповідно пов'язує їх між собою. Такою є основне її завдання. В свою чергу, задачі, що вирішуються комп'ютером, називають обчислювальними задачами та можуть бути вирішені шляхом застосування певних алгоритмів, тобто відповідних математичних методик та послідовних кроків.

Часто вирішення проблеми або задачі потребує значних затрат ресурсів. В такому випадку вона вважається суттєво складною незалежно від алгоритму, який використовується. Поняття алгоритму та його складності досить довгий час не були формалізовані. Тобто, логічно та інтуїтивно зрозумілі, але не обґрунтовані з наукової точки зору. І саме теорія складності обчислень дає визначення цього терміну, шляхом вивчення існуючих проблем, введення математичних моделей та оцінки необхідних ресурсів для їх виконання[1].

Для вирішення будь-якої задачі і виконання відповідного алгоритму необхідно використати такі ресурси як час та пам'ять. Саме їх аналіз проводиться для можливості оцінки алгоритму. В такому випадку розглядаються кількість зв'язків (складність зв'язків), кількість виходів в ланцюзі (складність схем) та кількість процесів (використовується в паралельних обчисленнях). Подібний аналіз встановлює практичні обмеження можливостей техніки, яка використовується для реалізації математичної моделі.

До галузей теоретичної інформатики, крім теорії складності обчислювальної техніки, відносять складність алгоритмів та їх аналіз. Головною відмінністю між аналізом алгоритмів та теорією складності обчислювальної техніки є те, що перший розглядає кількість ресурсів,

необхідних конкретному алгоритму для вирішення тієї чи іншої задачі, а другий, в свою чергу, шукає та досліджує всі можливі алгоритми для вирішення однієї проблеми. Питання про те, які проблеми можна вирішити алгоритмічно, задає теорія складності алгоритмів.

## 1.1 Пам’ять та час

Все швидший розвиток обчислювальної техніки розширює можливості для вирішення та реалізації багатьох задач, але, як і раніше, має обмежені ресурси. Тому, дуже важливо мати уявлення про кількість необхідних ресурсів (пам’ять та час) і їх обмеження, при використанні певних алгоритмічних моделей (наприклад, абстрактну машину Тьюринга). Дослідженням цього питання займається саме аналіз складності алгоритмів.

Складність алгоритмів — кількісна характеристика, яка визначає час, що необхідних для виконання алгоритму (часова складність), і об’єм пам’яті, необхідний для його виконання (складність пам’яті).

При аналізі обчислювальної складності алгоритму час є одним з головних ресурсів, які оцінюються. Найчастіше, коли мова йде про “складність” — це означає часову складність алгоритму.

В теорії складності не використовуються звичні одиниці часу (секунди, хвилини тощо), адже вони залежать від обчислювальної техніки та її покоління, яке використовується. Наприклад, більш сучасний комп'ютер виконає той же алгоритм за набагато коротший проміжок часу, ніж обчислювальна техніка 1960-х років. Таким чином, аналіз складності алгоритмів дає можливість визначити вимоги до комп’ютера для реалізації певного алгоритму. Це відбувається завдяки підрахунку кількості елементарних операцій, які виконуються під час обчислень. Приймається, що для виконання певних дій (операцій) витрачається постійний час на даному

комп'ютері. Тобто, на час, необхідний для тієї чи іншої операції, впливає не розмір вхідних даних, а кількість виконаних операцій[2].

Незважаючи на те, що аналіз часу є одним з головних для оцінки складності алгоритму, в той же час, ще один важливий ресурс — об'єм пам'яті комп'ютера, необхідної для роботи алгоритму. Очевидно, що дані ресурси є взаємозалежними та однаково важливими.

Щоб оцінити складність алгоритму необхідно проаналізувати ресурси, які використовуються, а саме пам'ять та час. Зокрема, для вирішення більш складних алгоритмів необхідно більше ресурсів. Таким чином, щоб визначити час або пам'ять, які витрачаються на вирішення певної проблеми, обчислюється функція від розміру вхідних даних. Зазвичай це розмір вхідних даних у бітах. Теорія складності досліджує збільшення часу виконання алгоритмів відповідно до збільшення розміру вхідних даних.

## 1.2 Часова складність

Час, який використовується при виконанні певного алгоритму описує один із типів обчислювальної складності — часова складність. Для того, щоб оцінити часову складність алгоритму, припускається, що кожна операція певного алгоритму займає константний проміжок часу і підраховується кількість елементарних дій, які виконуються для вирішення задачі. Розрахунок складності алгоритму за часом стає можливим, якщо вважати, що час для однієї елементарної операції є сталим та позначається як нотація О-велике і оцінюється як O(1). Враховуючи, що час роботи одного і того ж алгоритму може бути різним для різної обчислювальної техніки, на якій він виконується, то зазвичай використовується час роботи для гіршого випадку, який позначається як T(n) та є найбільшим часом роботи алгоритму для всіх вхідних даних розміру n. Лише в деяких випадках розраховується та

використовується середнє значення часу роботи алгоритму, що є математичним очікуванням часу роботи для всіх можливих вхідних даних.

Порахувати кількість кроків алгоритму на всіх можливих входах неможливо. Адже складність, як правило, збільшується відповідно до розміру вхідних даних. Тому, для вираження складності алгоритму за часом зазвичай використовується кілька функцій складності[3].

Найгірша складність — це максимум складності на усіх входах розміру n, середня складність, в свою чергу, визначається середнім значенням складності для всіх входів розміру n. Визначення різних функцій складності має сенс і є можливим лише в тому випадку, коли розмір і кількість можливих входів даної задачі скінченні.

Час роботи алгоритму класифікується в залежності від того, яка функція знаходиться під O-нотацією. Наприклад, алгоритм $T(n) \in O(n)$ називають алгоритмом з лінійним часом роботи, а алгоритм $T(n) \in O(n^a)$ для деякого $a>1$ називають поліноміальним.

## 1.3 Типи часу

### 1.3.1 Константний час

Алгоритм може вважатися алгоритмом константного часу O(1), якщо кількість елементарних операцій і час, необхідний на їх виконання, є відомими і константними. В такому випадку значення часу T(n) не залежить від розміру вхідних даних.

Одним з прикладів такого алгоритму є вибір одного елементу з масиву даних. В такому випадку виконується лише одна команда, яка потребує сталого проміжку часу. Інакше ж, коли масив не відсортований, алгоритм не є алгоритмом константного часу, оскільки необхідно пройти кожен його елемент для того, щоб обрати потрібний. Таким чином, ця операція буде займати лінійний час, O(n).

Ще один приклад задачі константного часу є задача "обміняти значення a і b, якщо необхідно в результаті отримати a≤b". Незважаючи на те, що даний алгоритм є алгоритмом константного часу, необхідно зауважити, що це не залежить від того, виконується уже дана нерівність чи ні. В будь-якому випадку існує певна константа t, для якої час виконання задачі завжди більша, ніж t.

Нижче наведено приклад коду, який працює константний час:

```
int index  = 5;
unt item = list[index];
if (умова вірна) then
виконати деякі операції з постійним часом роботи
else
виконати деякі операції з постійним часом роботи
for i = 1 to 100
for j = 1 to 100
        виконати деякі операції  постійним часом роботи
```

Якщо T(n) рівне O(α), де α — константа, то це еквівалентно тому, що T(n) рівне O(1).

1.3.2 Логарифмічний час

Ще один тип часу, який виділяється при класифікації — логарифмічний. Вважається, що алгоритм виконується за логарифмічний час, якщо T(n) = O(log n).

Використання логарифму з основою 2($\log_2 n$) зумовлено тим, що комп'ютер використовує двійкову систему числення. Але, незважаючи на це, в записі як O-велике при різних основах логарифму, наприклад, $\log_a n$ і $\log_b n$, вони відрізняються лише на постійний множник $\log_a b$. Тому, O(log n) є стандартним записом для алгоритмів логарифмічного часу.

Використання алгоритмів логарифмічного часу найбільш ефективне і поширене при вирішенні задач з двійковими деревами або при використанні

двійкового пошуку, а також для роботи з масивами даних розміру n. У випадку останніх важливо те, що співвідношення часу виконання однієї операції та розміру масиву зменшується зі збільшенням цього розміру.

Розглянемо приклад алгоритму логарифмічного часу — пошук слів в словнику. Нехай маємо словник D, який містить n слів, відсортованих по алфавіту. Також необхідно врахувати, що довжина всіх слів O(1) і доступ до будь-якого елементу словника можна отримати за константний час, завдяки індексу. Якщо D(k) — це k-й елемент словника, то можна перевірити наявність певного слова ω в словнику за час O(log n). Для такої перевірки розглянемо елемент словника D([n/2]), де [] – округлення числа вниз. Якщо D(⌊n/2⌋)<ω, то нам варто перейти в праву половину масиву, інакше — в ліву. В кінці ми отримаємо індекс першого слова, який рівний або лексикографічно більше, ніж ω.

```
int find(vector<string> &D, string w)
{
int n = D.sixe();
int l = -1, r = n;
while(r - l > 1) {
	int m = (l + r) / 2;
	if(D[m] < w) [
		l = m;
	} else {
		r = m;
	}
}
return r;
}
```

### 1.3.3 Полілогарифмічний час

Алгоритм працює за логарифмічний час, якщо $T(n) = O(\log^k n)$ для деякого k. Задача про порядок множення матриць є прикладом алгоритму полілогарифмічного часу, якщо вона вирішується на паралельній РАМ-машині[4].

1.3.4 Сублінійний час

На відміну від вище описаних алгоритмів та відповідних їм типів часу, алгоритми з сублінійним часом роботи працюють на звичайних послідовних моделях машин і передбачаються наявність апріорних знань про структуру входу. Для таких алгоритмів можливе використання ймовірнісних методів, які в більшості випадків мають бути рандомізованими для будь-якої нетривіальної задачі.

Враховуючи властивості алгоритмів сублінійного часу, а саме відсутність потреби повного читання вхідних даних, він досить сильно залежить від дозволених способів доступу у вхідному потоці[5]. У більшості випадків бітовий рядок $b_1$, …, $b_k$ і є вхідними даними (потоком), тому вважають, що за час O(1) алгоритм може знайти значення $b_i$ для будь-якого i.

Зазвичай алгоритми сублінійного часу є ймовірнісними і дають апроксимоване рішення, а також використовують розпаралелювання процесів, некласичне обчислення або мають гарантоване припущення щодо структури входу. Відповідно до цього, $T(n) = o(n)$.

1.3.5 Лінійний час

Дослідження апаратного та програмного виконання алгоритмів дають можливість класифікувати типи часу та створити алгоритми з лінійним або навіть кращим часом роботи. Лінійний час є досить важливим для виконання алгоритму та вирішення певних задач.

З математичної точки зору, в стандартних моделях розрахунків, існує мала ймовірність досягнення лінійного часу виконання для деяких алгоритмів, але в той же час можуть працювати за лінійний час. Це стає можливим при використанні розпаралелювання потоків виконання алгоритму

та вхідних даних. Прикладом роботи алгоритму за таким принципом є алгоритми Бойєра-Мура та Укконена.

Робота алгоритму описується як робота за лінійний час, якщо його складність рівна O(n). Фактично, якщо маємо великий розмір вхідних даних, то час роботи алгоритму буде збільшуватись відповідно до цього лінійно. Наприклад, алгоритм, який розраховує суму всіх елементів списку, працює проміжок часу, пропорційний довжині списку. Але час роботи в будь-якому випадку може не бути точно пропорційним, якщо значення n є досить малим[6].

1.3.6 Квазілінійний час

Відповідно до всіх вимог та класифікації, існує квазілінійний час, що тісно пов'язаний з описаним вище лінійним. Адже алгоритм працює за квазілінійний час, якщо $T(n) = O(n\log^k n)$ для деякої константи k. При k = 1 час роботи є лінійно-логарифмічним, а при $T(n) \in o(n^{(1+\varepsilon)})$ для будь-якого $\varepsilon>0$, оцінка часу роботи алгоритму визначає його як алгоритм з квазілінійним часом роботи. В такому випадку, час роботи даних алгоритмів є меншим, ніж час роботи полінома від n, степінь якого більше 1[7].

Квазілінійний та лінійно-логарифмічний час роботи демонструють такі алгоритми, як сортування злиттям на місці $O(n\log^2 n)$, швидке сортування, сортування злиттям, за допомогою бінарного дерева, плавне сортування та інші, швидке перетворення Фур'є (O(nlog n), алгоритм на матрицях Монжа O(nlog n).

В свою чергу лінійно-логарифмічний час є лише окремим випадком квазілінійного часу, де k=1, тобто T(n) = O(nlog n). Таким чином, зростання лінійно-логарифмічної функції відбувається швидше, ніж лінійної, але повільніше, ніж будь-який многочлен від n зі степенем, більшим 1[8].

Сортування за допомогою двійкового дерева, як приклад виконання операцій n раз з часом роботи O(log n), виконується шляхом вставки кожного

елементу масиву розміром n послідовно і демонструє алгоритм з лінійно-логарифмічним часом роботи.

З теоретично-інформаційної точки зору, обґрунтування необхідності як мінімум лінійно-логарифмічного числа порівнянь для найгіршого випадку, полягає у тому, що у результаті сортування отримуємо перестановку однозначно визначеного порядку довжини n, а загальна кількість сортувань, в свою чергу, дорівнює n!. У випадку однозначного закодування кожної з цим послідовностей певною послідовністю біт, необхідно в середньому $\log n! \sim (n\log\ n)$ біт інформації на одну перестановку. При попарному порівнянні двох елементів масиву, в результаті отримаємо рівно один біт даних, який показує відношення цих елементів, тобто або перший елемент менший, ніж другий, або — ні. Подібне використання попарних порівнянь для сортування дає можливість виконати алгоритм за час, кращий, ніж $O(n\log n)$.

1.3.7 Субквадратичний час

Сортування загального вигляду не працюють за лінійний час, тому перехід від квадратичного до субквадратичного часу є досить важливим з практичної точки зору.

Виконання алгоритму за субквадратичний час описується як $T(n) = o(n^2)$. Прикладом таких алгоритмів є алгоритми сортування, засновані на порівняннях, такі, як сортування вставками. Попри це існують більш сучасні алгоритми, які можуть бути прикладом субквадратичних алгоритмів. Наприклад, сортування Шелла.

1.3.8 Поліноміальний час

Теорема Кобхема визначає поліноміальний час, як синонім понять “легко піддається обробці”, “виконуваний”, “ефективний” або “швидкий”[9].

Центральним в теорії складності обчислень є час складності P, який визначається в задачах поліноміального часу. Алгоритми, які працюють за

поліноміальний час, мають верхню межу часу роботи, обмежену многочленом від розміру входу, тобто $T(n) = O(n^k)$ для деякої константи k.

Прикладом поліноміальних алгоритмів є алгоритм швидкого сортування n цілих чисел (максимум $An^2$ операцій для деякого сталого A), базові арифметичні операції (додавання, віднімання, множення, ділення і порівняння), максимальне утворення пар в графах.

Розрізняють алгоритми зі строгим і слабо поліноміальним часом, які стосуються вхідних даних, що складаються з цілих чисел.

Строго поліноміальний час визначається в арифметичній моделі обчислень. В такому випадку, за одиниці виконання приймаються базові арифметичні операції, незалежно від довжини операндів. Алгоритм працює за строго поліноміальний час, якщо кількість операцій в арифметичній моделі обчислень обмежена многочленом від числа цілих у вхідному потоці, і пам'ять, яка використовується алгоритмом, обмежена многочленом від розміру входу. Якщо виконуються обидві умови, алгоритм можна привести до алгоритму поліноміального часу, якщо замінити арифметичні операції на відповідні алгоритми виконання даних операції на машині Тьюринга.

При заданому цілому числі $2^n$, яке в машині Тьюринга займає пропорційний об'єм пам'яті, можна обчислити $2^{2^n}$ за допомогою n операцій, повторно підвівши до степеню. Пам'ять, яка використовується для представлення $2^{2^n}$ пропорційна $2^n$, але скоріше експоненціально, ніж поліноміально залежить від об'єму пам'яті вхідних даних. Тому, на машині Тьюринга неможливо виконати ці обчислення за поліноміальний час, але можна за поліноміальний час арифметичних операцій.

В той же час є алгоритми, які працюють за кількість кроків машини Тьюринга, обмежених поліноміальною довжиною бінарно закодованого входу, але не працюють за кількість арифметичних операції, обмежених многочленом від кількості чисел на вході. До таких алгоритмів можна віднести алгоритм Евкліда для обчислення найбільшого спільного дільника

двох цілих чисел. Час роботи такого алгоритму для двох цілих чисел a і b обмежений $O((\log a + \log b)^2)$ кроками машини Тьюринга. В даному випадку час роботи не є строго поліноміальним, оскільки кількість арифметичних операцій неможливо обмежити цілим числом на вході, а число кроків є многочленом від розміру бінарного представлення чисел a і b, що можна представити як (log a + log b). Час роботи алгоритму не залежить від чисел a та b, але залежить від кількості цілих чисел на вході. При роботі алгоритму за поліноміальний час, а не строго поліноміальний, кажуть, що він працює за слабо поліноміальний час. Прикладом виконання алгоритму за слабо поліноміальний час є лінійне програмування.

Алгоритми поліноміального часу в теорії складності обчислень також визначають деякі класи складності, пов'язані з їх використанням:

- P: клас складності задач можливості розв'язання, які можуть бути вирішені в детермінованій машині Тьюринга за поліноміальний час;
- NP: клас складності задач можливості розв'язання, які можуть бути вирішені в недетермінованій машині Тьюринга за поліноміальний час;
- ZPP: клас складності задач можливості розв'язання, які можуть бути вирішені з нульовою похибкою у ймовірнісній машині Тьюринга за поліноміальний час;
- RP: клас складності задач можливості розв'язання, які можуть бути вирішені з односторонніми похибками у ймовірнісній машині Тьюринга за поліноміальний час;
- BPP: клас складності задач можливості розв'язання, які можуть бути вирішені з двосторонніми похибками у ймовірнісній машині Тьюринга за поліноміальний час;

- BQP: клас складності задач можливості розв'язання, які можуть бути вирішені з двосторонніми похибками в квантовій машині Тьюринга за поліноміальний час.

Найменшим класом складності на детермінованій машині Тьюринга є клас P, в той же час він є стійким в термінах зміни моделі машини. Тобто, при переході від однострічкової машини Тьюринга до мультистрічкової, не призведе до зміни часу виконання лише якщо алгоритм працює за поліноміальний час, в іншому випадку, наслідком такого переходу може бути квадратичне прискорення.

1.3.9 Суперполіноміальний час

Якщо час роботи алгоритму T(n) має верхню межу, яка є поліномом, то такий алгоритм працює за суперполіноміальний час. В такому випадку час дорівнює $\omega(n^c)$ для всіх констант, де n — вхідний параметр, зазвичай — число біт входу. Суперполіноміального (точніше, експоненціального) часу потребує алгоритм, який здійснює $2^n$ кроків для входу розміру n. Якщо алгоритм використовує експоненціальні ресурси, він є суперполіноміальним, але в деяких випадках такі алгоритми можуть бути слабо суперполіноміальні. Тест простоти Адлемана-Померанса-Румелі, який працює за час $n^{(O(\log \log n))}$ на n-бітному вході, показує, що час його роботи зростає швидше, ніж поліном, якщо розмір входу достатньо великий.

Алгоритми суперполіноміального часу лежать поза класом складності P, а теза Кобхема доводить, що дані алгоритми є непрактичними. Враховуючи, що задача рівності класів P і NP не має рішення, то очевидно, що алгоритму для вирішення NP-повний задач на сьогодні не існує.

1.3.10 Квазіполіноміальний час

Існують алгоритми, час роботи яких суттєво більший, ніж час роботи поліноміальних алгоритмів, але в той же час менший, ніж алгоритмів експоненціального часу. До такого типу алгоритмів відносяться алгоритми квазіполіноміального часу, час роботи яких в гіршому випадку дорівнює

$2^{O((logn)^c)}$, якщо с є сталим значенням. Класичний алгоритм розкладання цілого числа на множники, загальний метод решета числового поля, час роботи якого $2^{O(n^{\frac{1}{3}})}$, не є квазіполіноміальними, адже представлення часу роботи не можна виразити як $2^{O((logn)^c)}$. У випадку, коли константа "c" у визначенні квазіполіноміального алгоритму дорівнює 1, маємо алгоритм поліноміального часу, інакше — сублінійного.

Найчастіше алгоритми квазіполіноміального часу виникають і використовуються для вирішення і приведення NP-складних задач. У випадку приведення задачі 3SAT до іншої задачі B, її розмір зміниться і стане $2^{O((logn)^c)}$. Завдяки цьому стає очевидно, що для задачі B не існує поліноміального алгоритму, якщо, в свою чергу, не існує квазіполіноміального алгоритму для 3SAT, як і для всіх NP-задач.

В апроксимаційних алгоритмах існують задачі, для яких є алгоритм квазіполіноміального часу, але не існує алгоритму поліноміального часу. Наприклад, орієнтована задача Штайнера. Для неї можна описати апроксимаційний квазіполіноміальний алгоритм з апроксимаційним коефіцієнтом $O(\log^3 n)$, де n — число вершин, але неможливо — алгоритм з поліноміальним часом.

Задачі квазіполіноміального часу відносяться до класу складності QP, який можна визначити як QP = $\bigcup_{c \in N} DTIME(2^{(logn)^c})$ .

Наявність алгоритму вирішення задач з класу NP за поліноміальний час є одним з основних питань теорії складності, як і рівність класів P і NP.

Сучасні алгоритми для NP-повних задач, наприклад 3SAT, виконуються за експоненціальний час. Крім того, для більшості NP-повних задач досі не існує алгоритмів з субекспоненціальним часом виконання. Дану гіпотезу називають гіпотезою експоненціального часу, адже вона обумовлює, що результати неапроксимованості в області апроксимаційних алгоритмів виходять з того, що NP-повні задачі не мають алгоритмів квазіполіноміального часу.

### 1.3.11 Субекспоненціальний час

Визначення субекспоненціального часу не є формалізованим та загальноприйнятим. Але кажуть, що алгоритм виконується за субекспоненціальний час, якщо час його виконання росте швидше, ніж поліноміальний, але залишається менше, ніж експоненціальний. Дане поняття показує, що такі алгоритми є більш гнучкими, ніж алгоритми експоненціальні[10]. Розглянемо два найбільш поширені визначення терміну "субекспоненціальний час".

1. Вирішення задачі алгоритмом, логарифм часу роботи якого зростає менше, ніж будь-який заданий многочлен, дає можливість говорити, що така задача вирішується за субекспоненціальний час. В такому випадку час роботи алгоритму для будь-якого ε>0, описується як $O(2^{n^{\varepsilon}})$, а множина подібних задач складає клас складності SUBEXP, який також можна виразити як SUBEXP = $\cap_{\varepsilon>0} DTIME(2^{n^{\varepsilon}})$. Необхідно також врахувати, що ε – не частина вхідних даних алгоритму, тому для кожного e може існувати свій власний алгоритм вирішення задачі.

2. Ще один варіант визначення субекспоненціального часу — $2^{o(n)}$. В такому випадку час роботи алгоритму буде більший, ніж у першому випадку. Наприклад, класичний алгоритм розкладання цілих чисел на множники, загальний метод решета числового поля, а також задача ізоморфізму графів.

Зазначимо, що є різниця, чи являється алгоритм субекспоненціальним по кількості вершин або числу ребер. В параметризованій складності ця різниця присутня явно шляхом вказання пари (L, k), задачі можливості розв'язання і параметра k. SUBEPT є класом всіх параметризованих задач, які працюють за субекспоненціальний час по k і за поліноміальний по n:

SUBEPT = DTIME($2^{o(k) * poly(n)}$).

Точніше, SUBEPT є класом всіх параметризованих задач (L, k), для яких існує функція f: N → N з f∈o(k) і алгоритм, який вирішує L за час $2^{f(k) * poly(n)}$.

Гіпотеза про експоненцільний час (ETN) стверджує, що 3SAT, задача виконання булевих формул в кон'юнктивній нормальний формі з максимум трьома літералами на пропозицію і n змінними, не може бути вирішена за час $2^{o(n)}$. Точніше, гіпотеза говорить, що існує деяка константа c>0, така, що 3SAT не може бути вирішена за час $2^{cn}$ на будь-якій детермінованій машині Тьюринга. Якщо через m позначити кількість пропозицій, ETN еквівалентна гіпотезі, що k-SAT не може бути вирішена за час $2^{o(n)}$ для будь-якого цілого k≥3. Із гіпотези про експоненціальний час слідує, що P≠ NP.

1.3.12 Експоненціальний час

Виконання алгоритму за час T(n), обмежений значенням $2^{poly(n)}$, де poly(n) — деякий многочлен від n, визначає його алгоритм з експоненціальним часом. Ще одне визначення експоненціального — T(n) обмежено O($2^{n^k}$) з деякою константою k. Алгоритми експоненціального часу виконуються на детермінованих машинах Тьюринга і утворюють час складності EXP = $\cup_{c\in N} DTIME(2^{n^c})$.

В деяких випадках використання терміну експоненціальний час має T(n) = $2^{O(n)}$, де O(n) — лінійна функція. Це приводить до класу складності E, E = $\cup_{c\in NN}$ DTIME ($2^{cn}$).

1.3.13 Подвійний експоненціальний час

Клас складності 2-EXPTIME (2-EXPTIME = $\cup_{c\in N} DTIME(2^{2^{n^c}})$) утворюють алгоритми подвійного експоненціального часу. Алгоритм виконується за двічі експоненціальний час (подвійний експоненціальний час), якщо T(n)обмежений значенням $2^{2^{poly(n)}}$, де poly(n) — многочлен від n.

Прикладом відомих подвійних експоненціальних алгоритмів є:

- процедура обчислення для арифметики Пресбургера;

- Обчислення базису Гребнера (в гіршому випадку);
- Елімінація кванторів в суттєво замкнутих полях потребує як мінімум подвійний експоненціальний час виконання і може бути виконана за цей час.

## 1.4 Таблиця складності за часом

Загальна інформація про відомі класи складності наведена в таблиці 1.1, де poly(x) означає многочлен від x, тобто poly(x) = $x^{O(1)}$[10].

Таблиця 1.1 — Класи складності за часом

| Назва | Клас складності | Час роботи, T(n) | Приклад часу роботи | Приклад алгоритму |
|---|---|---|---|---|
| Константний час | | O(1) | 10 | Визначення парності цілого числа (представленого в двійковому вигляді) |
| Обернена функція Аккермана | | O(a(n)) | | Амортизаційних аналіз на одну операцію з використанням множин, які не перетинаються |

Продовження таблиці 1.1

| Назва | Клас складності | Час роботи, T(n) | Приклад часу роботи | Приклад алгоритму |
|---|---|---|---|---|
| Ітеровано логарифмічний час | | O(log* n) | | Розподіл розфарбування циклів |
| Подвійний логарифмічний час | | O(log log n) | | Час амортизації на одну операцію при використанні обмеженої черги з пріоритетами |
| Логарифмічний час | DLOGTIME | O(log n) | log n, $\log n^2$ | Двійковий пошук |
| Полілогарифмічний час | | poly(log n) | $(\log n)^2$ | |
| Сублінійний час | | $O(n^c)$ при $0<c<1$ | $n^{½}$, $n^{2/3}$ | Пошук в k-мірному дереві |
| Лінійний час | | O(n) | n | Пошук найменшого або найбільшого елемента у не відсортованому масиві |

Продовження таблиці 1.1

| Назва | Клас складності | Час роботи, T(n) | Приклад часу роботи | Приклад алгоритму |
|---|---|---|---|---|
| "n log-зірочка n" | | O(n log* n) | | Алгоритм триангуляції багатокутника Зайделя |
| Лінійно-логарифмічний час | | O(n log n) | n log n, log n! | Найшвидше сортування порівнянням |
| Кубічний час | | $O(n^3)$ | $n^3$ | Звичайне множення двох матриць n x n. Обчислення часткової кореляції |
| Поліноміальний час | P | $2^{O(\log n)}$ = poly(n) | n, n log n, $n^{10}$ | Алгоритм Кармакара для лінійного програмування, АКС-тест на простоту |

Продовження таблиці 1.1

| Назва | Клас складності | Час роботи, T(n) | Приклад часу роботи | Приклад алгоритму |
|---|---|---|---|---|
| Квазіполіноміальний час | QP | $2^{poly(\log n)}$ | $n^{\log \log n}$, $n^{\log n}$ | Найшвидший відомий $O(\log^2 n)$ – апроксимаційний алгоритм для орієнтованої задачі Штейнера |
| Субекспоненціальний час (перше визначення) | SUBEXP | $O(2^{n^{\varepsilon}})$ для всіх ε > 0 | $2^{\log n \wedge \log \log n}$ | Якщо прийняти теоретичні гіпотези, BPP міститься в SUBEXP |
| Субекспоненціальний час (друге визначення) | | $2^{o(n)}$ | $2^{n^{\frac{1}{3}}}$ | Найшвидші відомі алгоритми розкладання на множники цілих чисел і перевірки ізоморфізму графів |

Продовження таблиці 1.1

| Назва | Клас складності | Час роботи, T(n) | Приклад часу роботи | Приклад алгоритму |
|---|---|---|---|---|
| Експоненціальний час (з лінійною експонентою) | E | $2^{O(n)}$ | $1.1^n$, $10^n$ | Вирішення задачі комівояжера за допомогою динамічного програмування |
| Експоненціальний час | EXPTIME | $2^{poly(n)}$ | $2^n$, $2^{n^2}$ | Вирішення задачі про порядок динамічного множення матриць за допомогою повного перебору |
| Факторіальний час | | O(n!) | n! | Вирішення задачі комівояжера повним перебором |

Продовження таблиці 1.1

| Назва | Клас складності | Час роботи, T(n) | Приклад часу роботи | Приклад алгоритму |
|---|---|---|---|---|
| Подвійний експоненціаль-ний час | 2-EXPTIME | $2^{2poly(n)}$ | $2^{2^n}$ | Перевірка вірності заданого твердження в арифметиці Пресбургера |

1.5 Статистичне призначення (Практичне застосування показників обчислювальної складності алгоритмів)

Дослідження властивостей алгоритмів необхідне для обґрунтованого їх застосування. Однією з таких властивостей є обчислювальна складність алгоритмів, яка враховує кількість операцій та операндів, кількість унікальних операцій та операндів, обсяг програми, час виконання алгоритму та інші[11].

Існує певне розмежування обчислювальної складності як властивості алгоритмів і її відповідних метрик та показників, до визначення яких виділяють два принципово різні підходи: імовірнісний та статистичний. Імовірнісний підхід був визначений раніше та є більш поширеним у використанні[12]. Разом з цим для аналізу обчислювальної складності алгоритмів необхідні деякі інші показники, котрі можна визначити двояко: апріорно на основі аналізу алгоритму і апостеріорно, підрахунком кількості обчислень під час його виконання.

Показники обчислювальної складності опосередковано дають уявлення про час виконання алгоритмів, що дає можливість оцінити ефективність роботи алгоритму та його удосконалення.

Враховуючи відомі властивості обчислювальної складності алгоритму, можна сказати, що це провідний фактор, що визначає його часову ефективність. Досить важливо. передумовою її використання є відсутність потреби виконання алгоритму для її визначення. Але при застосуванні показників обчислювальної складності алгоритму слід враховувати наступне:

- асимптотичні показники обчислювальної складності алгоритму дозволяють визначити найбільш конкурентоздатні алгоритми. У випадку однакової асимптотичної складності алгоритмів ці показники не дозволяють розв'язати задачу взагалі;
- обчислювальна складність алгоритмів залежить від обчислювальних механізмів, тому реальна складність може бути як занижена, так і підвищена засобами обчислювальних механізмів;
- обчислювальна складність може бути змінена алгоритмічно.

Крім того, необхідно враховувати, що обчислювальна складність алгоритмів залежить від усієї множини вхідних даних, що можна розглядати як важливий типовий окремий випадок.

# 2 ТЕОРЕТИЧНІ ЗНАННЯ ТА ЗАСОБИ ДЛЯ РЕАЛІЗАЦІЇ

## 2.1 Задання вхідної граматики

Для опису формальної мови, тобто виділення деякої підмножини всіх слів деякого скінченного алфавіту в теорії формальних мов використовується термін формальна граматика або просто граматика. Виділяють два типи граматик — породжувальні та аналітичні. Перші встановлюють правила, які використовуються для побудови будь-якого слова мови, другі — допомагають визначити чи входить дане слово в мову. Дані визначення формальних мов були введені американським вченим, математиком та філософом, Н. Хомським у 50-х роках XX сторіччя[16].

Алфавіт — скінченна множина символів, e — порожній ланцюжок, слово, послідовність. Алфавітом є об'єднання алфавітів, перетин, різниця алфавітів. Нехай T — алфавіт, тоді:

- $T^+$ — множина усіх можливих послідовностей, що складені з елементів цього алфавіту крім порожньої послідовності e.
- T* — множина усіх можливих послідовностей, що складені з елементів цього алфавіту, будь-якої довжини. Отже: $T^* = T^+ \cup \{\varepsilon\}$.
- $T^k$ — множина усіх можливих послідовностей, що складені з елементів цього алфавіту, довжини не більше k.

Мова в алфавіті T — це множина ланцюжків скінченної довжини в цьому алфавіті. Зрозуміло, що кожна мова в алфавіті T є підмножиною множини T*.

Формальна граматика — це четвірка G = {N, T, P, S}. Де:

- T — алфавіт термінальних символів, терміналів (від англ. terminate — завершитись). Термінальні символи є алфавітом мови.

- N — алфавіт нетермінальних символів, нетерміналів. T∈N = Ø; Нетермінали не входять в алфавіт мови.

- S — аксіома, спеціально виділений нетермінальний символ з якого починається опис граматики. S∈N

- P — правила виводу, скінченна підмножина множини $(T \cup N)^{+} \times (T \cup N)^{*}$.

Інколи P визначають так:

$\alpha \in (T \cup N)^{*} \times N \times (T \cup N)^{*}$, $\beta \in (T \cup N)^{*}$.

Елемент $(\alpha,\beta)$ з множини P називається правилом виводу і записується у вигляді $\alpha$ — $\beta$. Таким чином, ліва частина правила не може бути порожньою. Правила з однаковою лівою частиною записують:

$\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$, $\beta_i$, i = 1,2,..,n — називаються альтернативами виводу з ланцюжка $\alpha$.

Ланцюжок $\beta \in (T \cup N)^{*}$ назвемо безпосередньо виведеним з ланцюжка $\alpha \in (T \cup N)^{+}$ в граматиці G = {T, N, S, P} (позначається $\alpha \Rightarrow \beta$), якщо $\alpha = \xi_1 \gamma \xi_2$, $\beta = \xi_1 \delta \xi_2 : \xi_1, \xi_2, \delta \in (T \cup N)^{*}, \gamma \in (T \cup N)+, \gamma \to \delta \in P$.

Ланцюжок $\beta \in (T \cup N)^{*}$ назвемо виведеним з ланцюжка $\alpha \in (T \cup N)+$ в граматиці G = {T, N, S, P} (позначається $\alpha \Rightarrow^{*} \beta$), якщо $\exists \gamma_0, \gamma_1, \ldots, \gamma_n, n \geq 0 : \alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n = \beta$

Термінальний рядок називається правильним (або сентенціальною формою) відносно граматики G якщо він виводиться з аксіоми цієї граматики.

Граматика G називається неоднозначною, якщо існує декілька варіантів виводу слова ω в G (ω∈L(G)).

Приклад. Розглянемо таку граматику G = <N, $\Sigma$, P, S> зі схемою P.

S → S + S |S *S |a

Покажемо, що для ланцюжка ω = a+a+a існує щонайменше два варіанти виводу:

1. $S \Rightarrow S + S \Rightarrow S + S + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$

2. $S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$

Теорія граматик розглядає декілька стратегій виведення ланцюжка ω в G.

Лівостороння стратегія передбачає послідовність кроків безпосереднього виводу, при якій на кожному кроці до уваги береться перший зліва направо нетермінал.

Правостороння стратегія, в свою чергу, протилежна. Незалежно від стратегії, з виводом ω в G пов'язане з абстрактним синтаксичним деревом, яке визначає синтаксичну структуру програми[17].

Множина всіх правильних рядків, тобто термінальних рядків, що виводяться з аксіоми називається формальною мовою породженою граматикою G. В такому випадку $L(G) = \{a \in T^* \mid S \Rightarrow^* \alpha\}$.

Множина термінальних рядків, які визначаються та можуть бути виведені в граматиці G, називається мовою, що може бути розпізнана в даній граматиці, або допускається даною граматикою.

- Граматики $G_1$ та $G_2$ називаються еквівалентними, якщо $L(G_1) = L(G_2)$.
- Граматики $G_1$ та $G_2$ називаються майже еквівалентними, якщо $L(G_1) \cup \{\varepsilon\} = L(G_2) \cup \{\varepsilon\}$, тобто мови, ними породжувані, відрізняються не більш ніж на ε.

В залежності від того, в якому "напрямку" застосовуються відомі правила, граматика може бути породжуючою або розпізнавальною. Для перших виведення починається з аксіоми і закінчується термінальним рядком, другі — аналізують вхідний термінальний рядок на правильність.

2.1.1 Види граматик

Поняття Ієрархії Чомскі, або Ієрархії Чомскі-Шутценбергера вперше було описано Ноамом Чомскі в 1956 році і визначає ієрархію формальних граматик, які породжують формальні мови. Загалом було виділено та

описано чотири типи граматик, які виходять від базової, необмеженої граматики (тип 0) та послідовно обмежуються певними правилами.

В залежності від їх властивостей та можливостей генерації заданої формальної мови, формальні граматики ділять на відповідні категорії від типу 0 до типу 3.

Якщо формальна граматика позначена G = (N, E, P, S), то N — множина нетермінальних символів, E — множина термінальних символів, P — множина правил виводу та S — початковий символ[18].

В таблиці 2.1 наведено огляд чотирьох типів формальний граматик, правил виводу, формальних мов та автомати, здатні її розпізнати.

Таблиця 2.1 — Типи формальних граматик

| Граматика | Правила | Мови | Автомати | Скорочення |
|---|---|---|---|---|
| Тип-0 Довільна формальна граматика | $\alpha \to \beta$ $\alpha \in V^*$ N $V^*$, $\beta \in V^*$ | Рекурсивно зліченна | Машина Тьюринга | KSV* |
| Тип-1 Контекстно-залежна граматика | $\alpha$ A $\beta$ -> $\alpha$'γ'$\beta$ A∈N, $\alpha$, $\beta \in V^*$, $\gamma \in V^{+,}$ S → e дозволене, коли серед правил P відсутнє $\alpha \to \beta$ Sγ | Контекстно-залежна | Лінійний обмежений автомат | КЗ |

Продовження таблиці 2.1

| Граматика | Правила | Мови | Автомати | Скорочення |
|---|---|---|---|---|
| Тип-2 Контекстно-вільна граматика | A → γ A∈N, γ∈Y* | Контекстно-вільна | Недетермінований автомат з магазинною пам'яттю | КВ |
| Тип-3 Регулярна граматика | S → ε<br>A → aB (праволінійна) або<br>A → Ba (ліволінійна)<br>A → a<br>A → ε<br>A, B∈N, a∈Σ | Регулярна | Скінченний автомат (як детермінований, так і недетермінований) | А |

Позначення множин:

- Σ — множина термінальних символів
- N — множина нетермінальних символів
- V = Σ∪N — словник граматики (множина всіх термінальних та нетермінальних символів)

Операції:

- C — доповнення множин
- K — конкатенація формальних мов
- S — перетин множин
- V — об'єднання множин
- * — зірочка Кліні

Формальна мова належить до типу i, якщо її породжує граматика типу i. Формально, мова L належить до типу $i \in \{0, \ldots, 3\}$, якщо існує граматика $G \in Тип_i$ така, що $L = L(G)$. Тоді пишуть $L \in L_i$.

В ієрархії Чомскі, множина мов типу i є підмножиною мов типу(i - 1).

Визначення та опис формальних граматик Чомскі було необхідне для створення математичного опису природної мови, але на сьогодні вдалось розробити формальні граматики лише для декількох мов, в першу чергу, штучних. Проблема створення формальної граматики для більшості мов полягає в багатозначності слів. Правильне значення в такому випадку можна отримати шляхом аналізу розширеного контексту, в якому знаходиться аналізоване речення. В іншому випадку встановити його значення неможливо[19].

2.1.2 ANTLR та необхідна йому граматика

ANTLR (буквально англ. Another Tool For Language Recognition) — генератор синтаксичних аналізаторів, дозволяє автоматично створювати програму-парсер (як і лексичний аналізатор) однією з декількох цільових мов програмування (Java, C++, C#, Python, Ruby) за описом LL(*) – граматики мовою, близькою до EBNF. З його допомогою стає можливим конструювати компілятори, інтерпретатори, транслятори з різних формальних мов. Також однією з важливих функцій ANTLR є засоби для відновлення після помилок та повідомлення про них[20].

В основному, ANTLR зчитує граматику, яка використовується алгоритмом і допомагає у розпізнаванні мови, яка визначена граматикою. Тобто, програма читає вхідні дані і генерує помилку, якщо вхідний потік не відповідає синтаксису, визначеному граматикою. У випадку, коли синтаксичних помилок не виявлено, дія за замовчуванням – вихід без жодного повідомлення. Для того, щоб мати можливість реалізації певних алгоритмів, до граматичних елементів граматики додаються дії. які описуються мовою програмування, яка використовується для розпізнавання

помилок. В такому випадку, дії використовуються для створення та перевірки таблиць символів та передачі інструкцій цільовою мовою, у випадку використання компілятора.

Окрім використання ANTLR як лексер або парсер, він може використовуватись для створення парсерів дерев. Це так звані розпізнавачі, які обробляють абстрактні синтаксичні дерева, які можуть бути згенеровані автоматично. Вони є унікальними для ANTLR і допомагають обробляти абстрактні синтаксичні дерева.

Головними перевагами використання ANTLR є:

- доступне програмне забезпечення;
- використання єдиної нотації для опису лексичних та синтаксичних аналізаторів;
- зручність роботи з абстрактним синтаксичним деревом;
- надання повідомлень про помилки та відновлення після них;
- наявність візуальних середовищ розробки (ANTLR Works, ANTLR Studio, плагінів до Eclipse і IntelliJ IDEA), які дають можливість створювати та налаштовувати граматики, підтримують виділення синтаксису, автодоповнення, візуальне відображення граматик, яке будується в реальному часі відповідно до введених даних.

# 3 РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ

## 3.1 Загальна структура

Система передбачає певну залежність її модулів, тобто один модуль системи має використовувати або функціонал іншого модуля, або результат роботи іншого модуля. Модулі також складаються з підмодулів.

Головні модулі:

1) користувацький інтерфейс;
2) серверна частина;
    a. утилітарний модуль;
    b. модуль побудови графа;
    c. модуль обчислень;
3) допоміжний модуль для зв'язку користувацького інтерфейсу та серверної частини.

Структурна схема архітектури програми наведена у додатку А.

### 3.1.1 Користувацький інтерфейс

Користувацький інтерфейс – це інтуїтивно зрозумілий користувачу інтерфейс. Користувач задає в ньому певні налаштування необхідні для виконання обчислень: файли з граматикою та кодом програми користувача в нашому випадку. Головне вікно користувацького інтерфейсу зображено на рисунку 3.1.

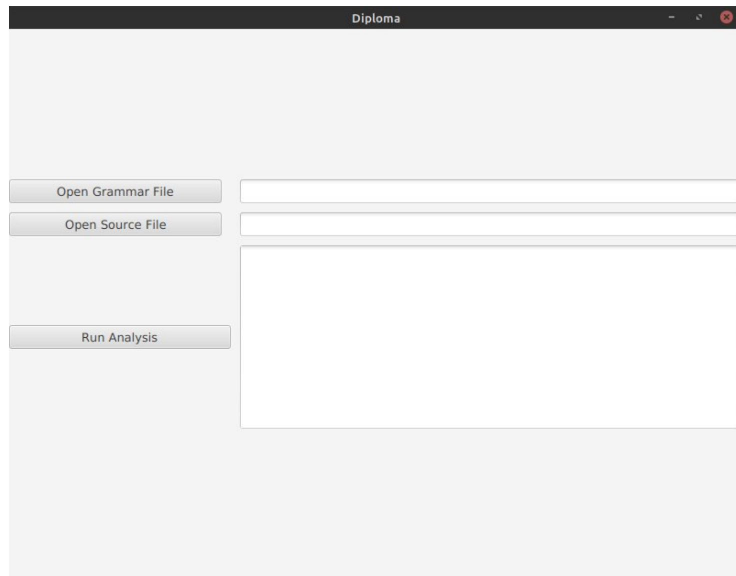

Рисунок 3.1 – Головне вікно програми

Далі за допомогою допоміжного модуля для зв’язку користувацького інтерфейсу та серверної частини задані користувачем данні та налаштування передаються до серверної частини де відбуваються необхідні обчислення. Після завершення всіх обчислень модуль користувацького інтерфейсу презентує користувачу результат виконання програми в зручному для розуміння вигляді.

3.1.2 Допоміжний модуль для зв’язку користувацького інтерфейсу та серверної частини

Для зв’язку користувацького інтерфейсу та серверної частини зазвичай використовують проміжне представлення вхідних даних та/або допоміжні скрипти, як це реалізовано в нашому випадку. Після отримання даних від користувача (шляхів до файлів з граматикою та кодом) за допомогою зовнішніх скриптів відбувається валідація даних (перевірка файлів на існування). Далі за допомогою ANTLR4 створюється парсер на основі заданої вхідної граматики. За допомогою створеного парсера та допоміжних функцій ANTLR4 на основі заданого файлу з кодом програми будується дерево розбору програми. Далі це дерево передається до модуля серверної частини на подальший аналіз. Після виконання всіх необхідних дій серверна

частина створює файл формату JSON який зчитує модуль користувацького інтерфейсу та презентує результат користувачу.

3.1.3. Модуль серверної частини

Серверна частина – це частина програми, що виконує всі необхідні обчислення над вхідними даними та, як правило, є незалежною від користувацького інтерфейсу.

## 3.2. Використання мови програмування Lisp

Для реалізації серверної частини була обрана мова Lisp, а саме її діалект SBCL – Steel Basic Common Lisp.

Lisp – мова програмування, що була розроблена наприкінці 50-х років для досліджень проблем штучного інтелекту. Особливості мови:

- підтримує такі парадигми програмування як: функціональну, імперативну та об'єктно-орієнтовну;
- підтримує динамічну типізацію;
- має об'єктну систему CLOS (Common Lisp Object Sytstem), що підтримує мультиметоди та їх комбінації;
- має стандартний механізм макросів.

Оскільки для обробки дерева розбору зручно використовувати функціональну мову програмування через її швидкодію та зручність написання і підтримку коду з великою кількістю рекурсивних функцій була обрана саме мова Lisp та її діалект SBCL. Також дерево розбору що віддає програма ANTLR4 після своєї роботи представлене у вигляді Lisp-подібного списку, через що нема необхідності робити приведення вхідних даних до іншої структури, що пришвидшує швидкодію розробленої програми.

### 3.3. Використання мови програмування Java

Для реалізації модуля користувацького інтерфейсу була обрана мова програмування Java.

Java – об’єктно-орієнтовна мова, що була випущена 1995-го року. В офіційній реалізації Java-програма компілюється в байткод та виконується віртуальною машиною, що робить розроблені програми крос-платформеними.

Java була обрана для реалізації модулю користувацького інтерфейсу через існуючу платформу JavaFX.

JavaFX – це платформа та набір інструментів для створення застосунків з насиченим користувацьким інтерфейсом та можливістю завантаження медіа.

Для збирання модуля користувацького інтерфейсу використовується засіб автоматизації роботи с програмними проєктами – Maven. Він регулює залежності модулів програми та сторонніх бібліотек та надає змогу запуску розробленого програмного продукту.

### 3.4. Вхідні данні

Розроблена програма отримує на вхід файл з кодом на будь-якій мові програмування та її граматику. Для демонстрації роботи програми нами була розроблена проста граматика, що містить всі необхідні конструкції мови, що підтримуються розробленою програмою на даний момент.

Розроблена граматика та структура її ієрархії зображена в додатку А.

Нижче наведений приклад коду написаний на розробленій нами тестовій граматиці.

```
function foo(param_1, param_2){
    var tmp = param_1 + 0;
```

```
        loop(var i = 0; i < tmp; i++){
            tmp = tmp - param_1;
        }
        loop(var i = 0; i < tmp; i++){
            tmp = tmp - param_1;
        }
}
```

3.5. Створення графу потоку керування програми

Граф потоку керування (CFG – Control Flow Graph) – це множина всіх можливих шляхів виконання програми представлених у вигляді графа.

За допомогою аналізу графу потоку керування можна визначити часову складність вхідного алгоритму.

Граф потоку керування будується з дерева розбору програми. В розробленій програмі за зчитування дерева розбору виконує функція get-parsed-tree з пакета backend/utils.

Під час розробки програми був розроблений список класів, що відповідають за певні сорти граматики, які необхідні для подальшого аналізу. Список таких класів визначений в пакеті backend/graph/gomain.

- string-node – нода строкового літералу;
- variable-node – нода змінної;
- loop-node – нода циклу;
- declaration-node – нода оголошення змінної;
- assignment-node – нода присвоєння змінної;
- expression-node – нода виразу;
- operation-node – нода операції;
- function-node – нода оголошення функції;
- program-node – нода оголошенна програми;

- root-node – нода, з якої граф бере початок;
- parameter-node – нода параметра функції;
- number-node – нода чисельного літералу;
- condition-node – нода умови.

Для створення графа був розроблений набір рекурсивних методів make/add-node та допоміжний макрос make/add-node-aux в пакеті backend/graph/build. Набір даних методів створює вище описані ноди та додає їх до загального графу на основі піддерев з дереву розбору програми. Інтерфейсною є функція build-graph з цього ж пакету, що приймає на вхід дерево розбору та створює граф потоку програми.

### 3.6. Обчислення складності алгоритмів

Розроблена програма підтримує обчислення складності алгоритмів на основі циклів та їх вкладених конструкцій. За процес обчислення часової складності на основі графу потоку програми відповідає інтерфейсна функція process-loops-analysis з пакета backend/analysis/loops. Діаграма роботи алгоритму наведена в додатку А.

Алгоритм підраховує кількість вкладених циклів для кожної функції та кількість циклів на зовнішньому рівні. Оскільки при обчисленні складності алгоритму нам важливо знати порядок складності, то коефіцієнт складності при передачі результату до модуля користувацького інтерфейсу опускається.

## 3.7. Зв’язок між модулем користувацького інтерфейсу та серверної частини

Зв’язок між модулем користувацького інтерфейсу та модулем серверної частини виконується за допомогою допоміжного скрипта run.sh, makefile та файлу формату JSON.

Скрипт run.sh містить набір команд вдосконаленої і модернізованої командної оболонки Bourne Shell. Цей скрипт приймає шляхи до файлів граматики та коду програми, що підлягає аналізу, виконує валідацію даних, копіює вхідні дані для більшої надійності та викликає makefile.

Makefile – це набір команд ультиліти make, що використовються для автоматизації комплілювання та/або компонування програми. Розроблений makefile викликає ультиліту ANTLR4 з усіма необхідними для її коректної роботи параметрами. Розроблений makefile містить інструкції для очищення тимчасової директорії та всіх тимчасових файлів директорії розробленої програми. Також makefile містить інструкції для створення образу модуля серверної частини та його запуску. Повний перелік розроблених інструкцій makefile:

- help - виводить повідомлення з переліком та описом всіх інструкцій;
- clean-grammar – очищує всі файли згенеровані ультилітою ANTLR4 с тимчасової директорії;
- clean-tmp – повністю очищує тимчасову директорію;
- clean-lisp – видаляє всі *.fasl та *~ файли;
- clean – виконує всі 'clean-*' інструкції;
- build-grammar – генерує файли парсера граматики за допомогою ультиліти ANTLR4 в тимчасовій директорії;
- build-image – створює образ серверної частини;
- build – виконує всі 'build-*' інструкції;

- parse – створює дерево розробу з вхідного файлу з кодом;
- run – створює образ серверної частини та запускає його;
- all-no-clean – виконує інструкції 'build', 'parse' та 'run';
- all – виконує інструкції 'clean', 'build', 'parse' та 'run'.

Після виконання аналізів щодо підрахування часової складності алгоритмів серверна частина виконує перетворення з внутрішнього варіанту представлення даних до формату JSON. За це відповідає функція encode-time-difficult-respons з пакету backend/encode.

JSON – це текстовий формат призначений для обміну даних. Оскільки JSON базується на тексті, то може бути прочитаним людиною та дає змогу легко описувати об’єкти, масиви та інші структури даних.

Даний формат представлення даних є зручним, оскільки є універсальним и до такого формату можна привести будь-яку структуру даних.

## 3.8. Інструкція користувача

Перед початком роботи необхідно підготувати вхідні дані: файли з граматикою та кодом програми, що підлягає аналізу.

Далі необхідно зайти в директорію с розробленим проєктом та виконати команду терміналу make run. Після цього з’явиться вікно користувацького інтерфейсу (рисунок 3.2) .

Далі неохідно вибрати підготовлені файли за допомогою кнопок Grammar File та Source File. Після натискання будь-якої з кнопок користувачу буде представлене вікно з файлами комп’ютера, де необхідно вибрати файли граматики та коду (рисунок 3.3).

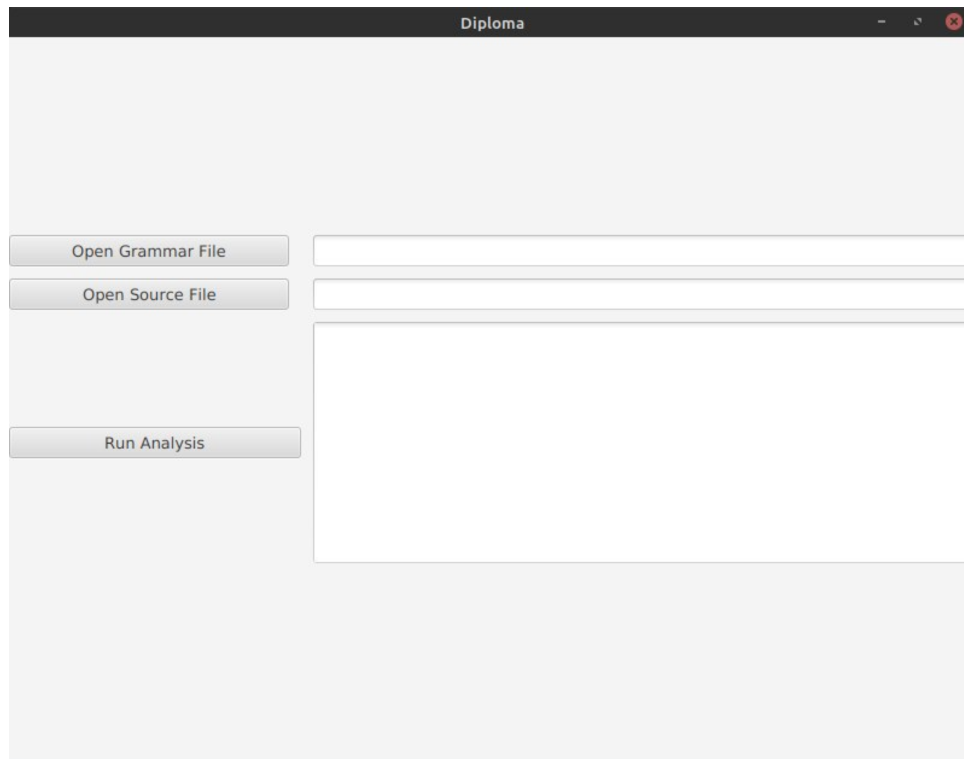


Рисунок 3.2 – Головне вікно користувацького інтерфейсу

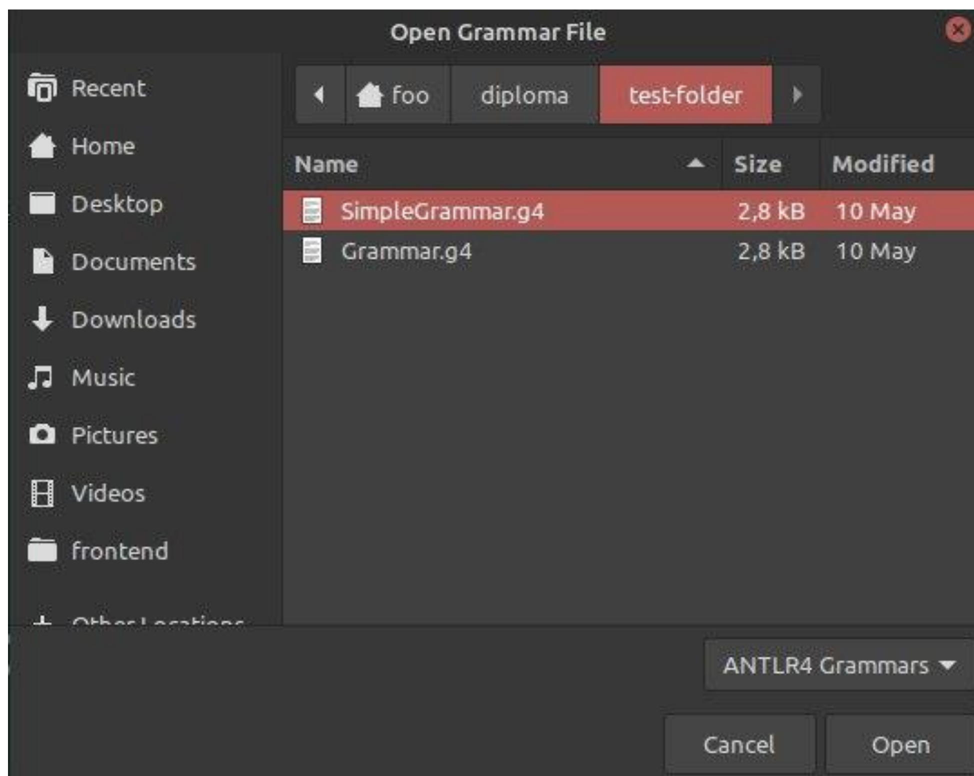


Рисунок 3.3 – Вікно вибору файлів

Після цього необхідно натиснути кнопку Run Analysis, після чого буде запущений аналіз і через декілька секунд користувач зможе побачити результат у вікні поряд з кнопкою (рисунок 3.4).

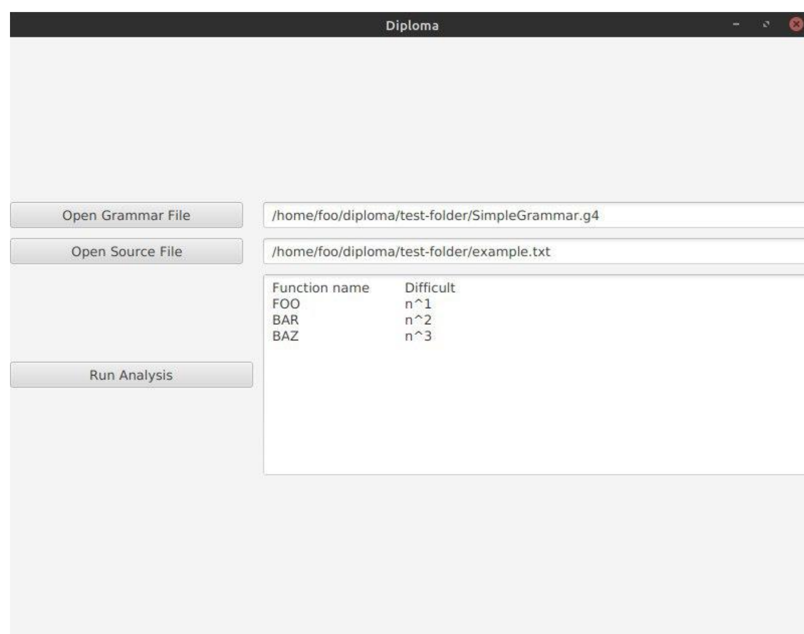


Рисунок 3.4 – Вивід результату роботи програми

Якщо якийсь з необхідних двох файлів не буде завантажений до розробленої програми в тому ж вікні з відображенням результату буде відображене повідомлення про нестачу вхідних даних (рисунок 3.5).

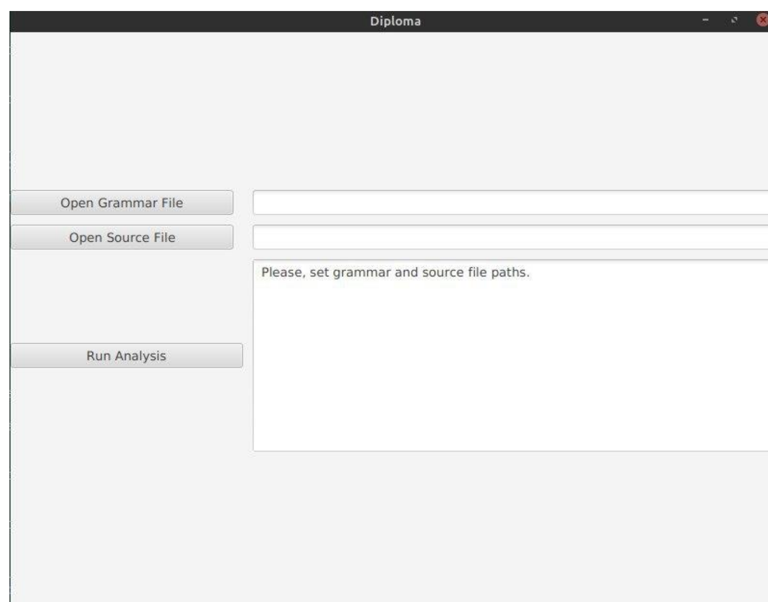


Рисунок 3.5 – Вивід повідомлення про не введені вхідні дані

3.9. Тестування розробленої програми

Тестування розробленої програми поводиться на основі порівняння практично отриманих результатів та теоретично доведених значень. Теоретично доведено, що цикл має складність $n$, вкладений цикл - $n^2$, а двічі вкладений - $n^3$. Якщо в коді зустрінеться два цикли одного рівня вкладеності, то складність буде дорівнювати $2n$, але при нескінченно великому $n$, коефіцієнт не грає великого значення, більш важливим є показник степеня, тому коефіцієнт не беруть до уваги.

Для тестування було виділено три випадки: два цикли без вкладень (функція Foo), вкладений цикл (функція Bar) та два двічі вкладені цикли (функція Baz). Виходячи з описаного вище складність алгоритмів має складати $n$, $n^2$ та $n^3$відповідно.

Нижче приведено код програми на розробленій нами тестовій граматиці, який містить цикли різного рівня вкладеності та містить декілька циклів розташованих на одному рівні вкладеності.

```
function foo(param_1, param_2){
        var tmp = param_1 + 0;
        loop(var i = 0; i < tmp; i++){
            tmp = tmp - param_1;
        }
        loop(var i = 0; i < tmp; i++){
            tmp = tmp - param_1;
        }
}
function bar(param_1, param_2){
        var tmp = param_1 + 0;
        loop(var i = 0; i < tmp; i++){
            tmp = tmp - param_1;
```

```
            loop(var j = 0; i < tmp; j++){
                    tmp = tmp - param_1;
            }
        }
}

function baz(param_1, param_2){
        var tmp = param_1 + 0;
        loop(var i = 0; i < tmp; i++){
            tmp = tmp - param_1;
            loop(var j = 0; i < tmp; j++){
                    tmp = tmp - param_1;
                loop(var k = 0; i < tmp; k++){
                    tmp = tmp - param_1;
                }
            }
        }
        loop(var i = 0; i < tmp; i++){
            tmp = tmp - param_1;
            loop(var j = 0; i < tmp; j++){
                    tmp = tmp - param_1;
                loop(var k = 0; i < tmp; k++){
                    tmp = tmp - param_1;
                }
            }
        }
}
```

Практично отримані результати за допомогою розробленої програми співпадають з теоретично доведеними (рисунок 3.6).



Рисунок 3.6 – Результати тестування розробленої програми

# ВИСНОВКИ

Основна задача теорії складності обчислень – визначення складності алгоритмів, що дає можливість їх удосконалення. Незважаючи на досить значні можливості сучасної техніки, проведення такого аналізу є необхідним. Адже саме це дає можливість оптимізації процесів, які відбуваються та загалом пришвидшення роботи алгоритму.

Складність алгоритмів — кількісна характеристика, яка визначає час, що необхідних для виконання алгоритму (часова складність), і об’єм пам’яті, необхідний для його виконання (складність пам’яті).

Пам’ять та час - основні ресурси, необхідні для вирішення задач та, відповідно, виконання алгоритмів. Для оцінки алгоритму проводиться саме аналіз ресурсів, які використовуються. Такий аналіз може встановити практичні обмеження можливостей техніки, яка використовується для реалізації математичної моделі.

У ході роботи було проведено дослідження існуючих типів складності, способів її визначення та можливих варіантів проведення аналізу складності алгоритмів. Також на основі проведених досліджень було розроблено програмне забезпечення, яке дає можливість статистичного аналізу алгоритмів.

Розроблене програмне забезпечення дає можливість аналізу складності структурних одиниць (функцій) певних алгоритмів та різних граматик. В свою чергу для тестування програми було розроблено власну граматику, яка описувала програмний код аналіз якого також було проведено в ході роботи.

Практично отримані результати співпадають з теоретичними відомостями щодо складності алгоритмів та їх частин, що дає зрозуміти, що розроблене програмне забезпечення працює коректно та надає вичерпну інформацію щодо складності обчислень алгоритмів.

Саме такий аналіз дає можливість оптимізації, зменшення кількості необхідних ресурсів для роботи алгоритму, адже в більшості випадків

вирішення певної задачі потребує досить великої кількості ресурсів, які не є безмежними.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1 Прийма С.М. Теорія алгоритмів: Навчальний посібник. – Мелітополь: ФОП Однорог Т.В., 2018. – 116 с.

2 Хопкрофт, Джон, Э., Мотвани, Раджив, Ульман, Джеффри, Д.. Введение в теорию автоматов, языков и вычислений, 2-е изд.. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2008. — 528 с. : ил. — Парал. тит. англ.

3 Naik, Ashish V.; Regan, Kenneth W.; Sivakumar, D. (1995). "On Quasilinear Time Complexity Theory" (PDF). Theoretical Computer Science. 148 (2): 325–349. doi:10.1016/0304-3975(95)00031-q. Retrieved 23 February 2015

4 J. E. Rawlins, Gregory E. Shannon. Efficient Matrix Chain Ordering in Polylog Time // SIAM Journal on Computing. — Philadelphia: Society for Industrial and Applied Mathematics, 1998. — Т. 27, вып. 2. — С. 466—490. — ISSN 1095-7111. — doi:10.1137/S0097539794270698.

5 Ravi Kumar, Ronitt Rubinfeld. Sublinear time algorithms // SIGACT News. — 2003. — Т. 34, вып. 4. — С. 57—67. — doi:10.1145/954092.954103.

6 Dr K N Prasanna Kumar, Prof B S Kiranagi and Prof C S Bagewadi. A general theory of the system ‘quantum information - Quantum entanglement, subatomic particle decay – Asymmetric spin states, non locally hidden variables – A concatenated model // International Journal of Scientific and Research Publications. — July 2012. — Т. 2, вып. 7. — ISSN22503153.

7 Ashish V. Naik, Kenneth W. Regan, D. Sivakumar. On Quasilinear Time Complexity Theory (англ.) // Theoretical Computer Science (англ.)русск.. — Vol. 148. — P. 325—349.

8 Sedgewick, R. and Wayne K (2011). Algorithms, 4th Ed. p. 186. Pearson Education, Inc.

9 Cobham, Alan (1965). "The intrinsic computational difficulty of functions". Proc. Logic, Methodology, and Philosophy of Science II. North Holland.

10 Arora, Sanjeev; Barak, Boaz (2009), Computational Complexity: A Modern Approach, Cambridge University Press, ISBN 978-0-521-42426-4, Zbl 1193.68112

11 Papadimitriou, Christos (1994), Computational Complexity (1st ed.), Addison Wesley, ISBN 978-0-201-53082-7

12 Шинкренко В.І. Особливості практичного застосування показників обчислювальної складності алгоритмів. — Проблеми програмування. 2008. №2-3. Спеціальний випуск

13 Michael Sipser. Introduction to the Theory of Computation. — Course Technology Inc, 2006.

14 Herbert S. Wilf Algorithms and Complexity. - University of Pennsylvania Philadelphia, Internet Edition, Summer, 1994

15 James Aspnes Notes on Computational Complexity Theory CPSC 468/568: Spring 2020

16 Parr, Terence (May 17, 2007), The Definitive Antlr Reference: Building Domain-Specific Languages (1st ed.), Pragmatic Bookshelf, p. 376

17 Серебряков В.А., Галочкин М.П., Гончар Д.Р., Фуругян М.Г. Теория и реализация языков программирования //М.: МЗ-Пресс, 2006 г., 2-е изд. ISBN 94073-094-9.

18 Noam Chomsky. [PDF Three models for the description of language]. — 1956. — T. Vol.2. — C. 113–124. — (IRE Transactions on Information Theory)

19 Noam Chomsky, Marcel P. Schützenberger. The algebraic theory of context free languages, Computer Programming and Formal Languages / P. Braffort, D. Hirschberg. — Amsterdam, 1963. — C. 118–161.

20 Офіційний сайт проєкту ANTLR, URL: https://www.antlr.org/