Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра_____Програмної інженерії_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти - другий (магістерський)

Дослідження методів лексичного аналізу для оцінки складності алгоритмів в системі «Algorithms battle»

(тема)

Виконав: студент 2 курсу, групи ПЗСм-18-1___
_____Різник О. К._____
(прізвище, ініціали)

спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-професійної програми_____
(тип програми)

Програмне забезпечення систем_____
(повна назва освітньої програми)

Керівник _____доцент, к.т.н. Мазурова О.О._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар
(підпис)

2019р.

Харківський національний університет радіоелектроніки

Факультет_Комп'ютерних наук_____

Кафедра_Програмної інженерії_____

Рівень вищої освіти - другий (магістерський)

Спеціальність _121-Інженерія програмного забезпечення_____
(код і повна назва)

Тип програми освітньо-професійна програма

Освітня програма_ Програмне забезпечення систем_____

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові _Різнику Олександру Костянтиновичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи _ Дослідження методів лексичного аналізу для оцінки складності алгоритмів в системі «Algorithms battle» _____
затверджена наказом університету від "___" _____20 ___ р № _____
заповнюється вручну після отримання наказу

2. Термін подання студентом роботи до екзаменаційної комісії 20 грудня 2019 р.

3. Вихідні дані до роботи електронні ресурси за обраною тематикою, мінімальні вимоги до функціональності програми, загальні вимоги до архітектури системи
4. Перелік питань, що потрібно опрацювати в роботі *аналіз предметної області і постановка задачі, аналіз вимог до програмної системи, UML-моделювання предметної області, дослідження методів теорії прийняття рішень, розробка математичної моделі для підтримки дерев цілей, розробка математичної моделі для вирішення оптимізаційної задачі про призначення, розробка бази даних, розробка алгоритмів побудови дерева цілей та вирішення оптимізаційної задачі про призначення, програмна реалізація системи, тестування.*

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	доцент, к.т.н. Мазурова О. О.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка*
1	Аналіз проблемної області та постановка задачі	02.09.19-18-09.19	
2	Формування вимог до програмної системи	18.09.19-21.09.19	
3	Аналіз та UML-моделювання предметної області	21.09.19-30.09.19	
4	Дослідження методів оцінювання складності коду	01.10.19-14.10.19	
5	Розробка математичних моделей	14.10.19-24.10.19	
6	Розробка бази даних	24.10.19-26.10.19	
7	Розробка алгоритму оцінювання коду	26.10.19-13.11.19	
8	Програмна реалізація та тестування системи	14.11.19-24.11.19	
9	Підготовка пояснювальної записки	24.11.19-03.12.19	
10	Підготовка презентації та доповіді	03.12.19-12.12.19	
11	Попередній захист	13.12.19	
12	Нормоконтроль, рецензування	14.12.19	
13	Занесення диплома в електронний архів	15.12.19	
14	Допуск до захисту у зав. Кафедри	16 .12.19	
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання __2___ вересня_____2019 р.

Студент _____

(підпис)

Керівник роботи _____ доцент, к.т.н. Мазурова О. О.____

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної магістерської роботи містить 70 с., 15 рисунків, 4 таблиці.

АЛГОРИТМ, ВЕБ-СИСТЕМА, ЛЕКСИЧНИЙ АНАЛІЗ, СКЛАДНІСТЬ АЛГОРИТМА, BIG-O NOTATION, JAVA, MONGODB, KAFKA, SPRING BOOT, SPRING FRAMEWORK

Метою роботи є дослідження методів оцінки аналізу складності алгоритмів, реалізація нового методу, створення веб-системи для онлайн змагань з вирішення алгоритмічних задач та використання нового методу оцінки складності алгоритму у системі.

Методом вирішення є концептуальне моделювання предметної області, використання об’єкто-орієнтованого підходу до створення програмного продукту. Для розробки було обрано мову програмування Java, фреймворк Spring Framework 5.0 та Spring Boot 2.0, інструмент збірки проектів Maven, черга повідомлень Apache Kafka та середовище розробки Intellij IDEA. В якості основи лексичного алгоритму було взято універсальний лексичний аналізатор, що був удосконалений для вирішення необхідних обчислень.

Результатом роботи є лексичний та часовий алгоритм оцінювання програми.

ALGORITHM, ALGORITHMS COMPLEXITY, BIG-O NOTATION, DATABASE, JAVA, KAFKA, LEXICAL ANALYSIS, MONGODB, SPRING BOOT, SPRING FRAMEWORK, WEB-SYSTEM

The aim of the work is a research of algorithms complexity analysis methods, implementation of a new one, creating a web-system for online competition in solving algorithmics tasks and using the new method of algorithm complexity analysis in the system.

The solution methods are conceptual modeling of the domain, using an object-

oriented approach to software application design. Java language, Spring Framework 5.0 and Spring Boot 2.0, build tool Maven, message queue Apache Kafka and development environment Intellij IDEA were chosen as the tools of development. As a core of lexical algorithms was chosen general lexical analyzer which was improved for solving required computations.

The resulting work are two algorithms: lexical based and a time based.

ЗМІСТ

ВСТУП

У сучасному світі понад три мільярди двісті мільйонів людей мають інтернет та користуються різноманітними веб-сервісами. Тому дуже важливо мати можливість оброблювати запити всіх користувачів максимально швидко та ефективно, тому що при величезних кількостях користувачів та їх запитів – навіть найновіші сервери можуть бути недостатньо потужними аби задовольнити потреби клієнтів.

Існують архітектурні підходи, які використовуються для зменшення навантаження на сервера, такі як розширення кількості серверів які рівномірно розподіляють навантаження між собою та оброблюють запити, але наскільки б багато не було серверів, через помилки розробників – всі вони можуть не справлятись з даним навантаженням, якщо наприклад якась функція програми виконується дуже довго через не оптимальність написаного алгоритму, і якщо нею користуються велика кількість клієнтів – це може бути дуже критичним для функціонування сервісу. При правильному використанню пам’яті та часових затрат на виконання функцій – можна зберегти значну кількість ресурсів.

Для того, щоб контролювати витрати ресурсів, необхідно мати можливість точно визначати наскільки та чи інша функція/алгоритм оптимально використовує ресурси.

На сьогодні, основною мірою оптимальності алгоритму є часова складність алгоритму.

Часова складність алгоритму в комп'ютерних науках є обчислювальною складністю алгоритму, яка описує час потрібний для виконання алгоритму. Вона зазвичай визначається шляхом підрахунку кількості елементарних операцій, виконуваних алгоритмом, при цьому вважають, що кожна елементарна операція виконується за фіксовану кількість часу [1].

Як видно з визначення – для підрахунку складності алгоритму – необхідно

порахувати кількість операцій, яка виконується в програмі. Проте всі підрахунки складності на сьогодні базуються не на підрахунку кількості операцій, а на замірі часу виконання алгоритму на різних наборах даних різних розмірів та виведення формули росту функції на основі цих даних. Хоча цей спосіб оцінювання складності алгоритму досить точний, він має декілька недоліків:

– по-перше це великі ресурсні затрати через необхідність викликати вхідну програму велику кількість разів з великою кількістю вхідних даних;

– по-друге це великі ресурсні затрати через необхідність викликати вхідну програму велику кількість разів з великою кількістю вхідних даних;

– по-третє при паралельному оцінювання інших алгоритмів (якщо це онлайн система для вирішення алгоритмів, то паралельно велика кількість користувачів можуть відправляти свої рішення), різке зростання кількості алгоритмів для оцінювання можуть зменшити продуктивність системи – тобто збільшити час виконання алгоритму і тоді кореляція даних по оцінювання буде неточною.

Таким чином, під час магістерської роботи було проведено дослідження існуючих способів оцінки складності алгоритмів, досліджено їх переваги та недоліки, досліджено можливості реалізації алгоритму на основі лексичного аналізу програми, розроблено алгоритм лексичного оцінювання складності алгоритмі на основі лексичного аналізу з використанням лексичного аналізатору, розроблено часовий алгоритм оцінювання складності алгоритму та програмно їх реалізовано.

За результатами атестаційної роботи магістра було розроблено презентацію (додаток А).

Написана та подана до опублікування до науково-технічного журналу «Молоді вчені» стаття Дослідження методів лексичного аналізу для оцінки складності алгоритмів в системі «Algorithms battle» (додаток Б). Лістинг коду наведено в додатку В. Усі матеріали наведено на диску (додаток Г).

1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблемної області оцінювання складності алгоритмів

На даний час усі відомі веб-сервіси, що створені для тренування розв’язання алгоритмічних задач не мають загальноприйнятого способу визначення складності алгоритму.

Існує еталонний спосіб визначення часу виконання алгоритму на наборах вхідних даних різного розміру, та з отриманої вибірки визначається формула росту часу в залежності від збільшення вхідних даних. Проте, цей спосіб не є загальновживаним оскільки неможливо передбачити чи закінчить програма своє виконання чи ні, а виконувати програму, яка потенційно має дуже велику складність виконання особливо на великих даних – дуже невигідно.

Цей спосіб є досить точним, проте його точність прямо пропорційно залежить від кількості виконання програми на різних (зростаючих) наборах даних. Оскільки при паралельному навантаженні дані можуть бути недостатньо точними, а також якщо взяти до уваги те, що для досягнення точного результату потрібно велику кількість разів виконувати алгоритм з великою вибіркою даних, що потребує значних затрат ресурсів – то є очевидним, що є потреба у ресурсоекономному та точному алгоритмі визначення складності алгоритму.

Головною проблемою в реалізації такого алгоритму є неможливість створення універсального алгоритму, який для довільної задачі зміг би порахувати кількість її операцій. Це твердження випливає з доведеної в 1936 році Аланом Тьюрінгом проблеми зупинки [2]. Ця проблема полягає в тому, що не може існувати загального алгоритму для розв’язання проблеми зупинки для всіх пар програма-вхідні дані. Неформально ця задача звучить так: дано алгоритм та вхідні дані. Потрібно визначити, чи завершиться виконання програми у будь-який момент часу чи вона буде працювати безкінечно.

Не зважаючи на те, що універсальний алгоритм реалізувати неможливо – можна створити алгоритм, який зможе визначати складність алгоритму для визначеного набору задач (та доповнятися з часом, додаючи різні складні структури для аналізу). Корисність цього алгоритму буде полягати не в повній заміні універсального алгоритму, що працює на основі визначення часу виконання програми, а в попередньому аналізу вхідної програми оптимальним алгоритмом – і у випадку, якщо аналіз програми лексичним алгоритмом не буде можливим – то перейти до універсального алгоритму. На великих вхідних даних, якщо алгоритм буде аналізувати навіть невеликий відсоток усіх алгоритмів – економія ресурсів буде значною, а точність буде більшою.

Задачею лексичного алгоритму буде аналіз структур вхідної програми, розбивання її на найпростіші структури та аналіз кожної з структури на кількість операцій що виконуються. В залежності від співвідношення розміру вхідних даних до кількості виконаних операцій програмі буде присвоєна її складність (див. рис. 1.1):

Як видно з графіку – деякі функції мають дуже схожий графік росту на невеликих наборах даних, наприклад – $\log_2 n$ та $\sqrt{n}$ мають приблизно однакову кількість операцій (N) на відрізку n Є [0; 30], що показує досить схожу формулу росту функції і тому при підході визначення складності алгоритму через засікання часу потрібно дуже багато разів виконати алгоритм на великих даних щоб відрізнити дві схожі функції зростання.

На відміну від цього способу, лексичний алгоритм не потребує виконання алгоритму, а може підставляти значення кількості вхідних даних та проводити аналіз довільну кількість разів, до тих пір поки не буде чітко видно формулу зростання функції, що набагато ресурсоекономно та швидше ніж у загально використовуваному алгоритмі (див. рис. 1.1).

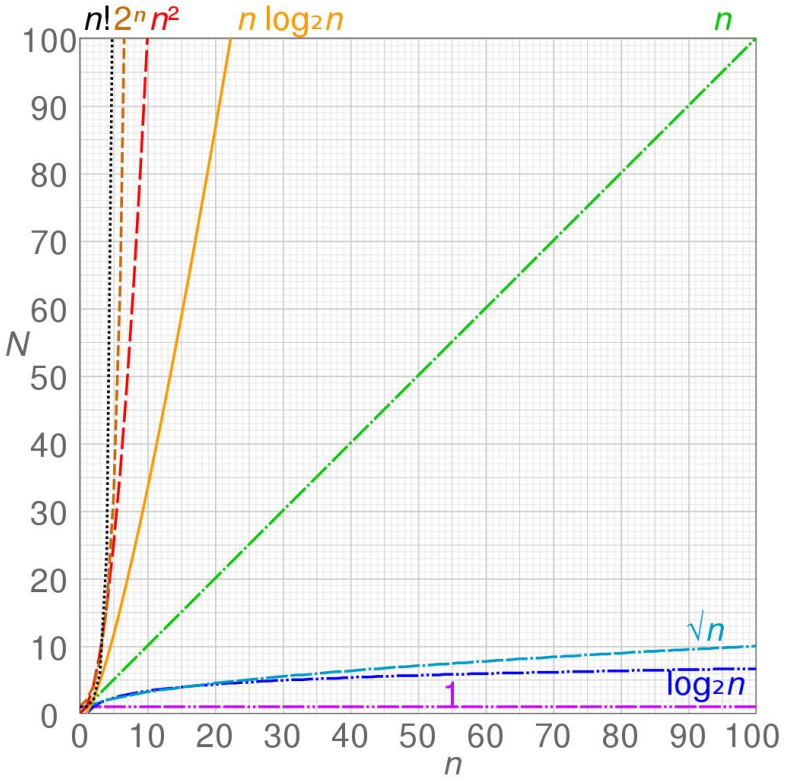

Рисунок 1.1 – Графік найпоширеніших функцій в аналізі алгоритмів.

Проте головним недоліком лексичного алгоритму буде те, що він буде ігнорувати алгоритми, які мають рекурсивні інструкції або умовні оператори, тому що він зможе обробляти лише один варіант виконання програми (через неможливість визначення в інструкції якого умовного оператору було передано виконання програми при дійсному виконанні програми). Можливо, що є способи моделювання які б включали в себе умовні оператори, проте на даному етапі аналізу цього способу не було знайдено.

Після аналізу переваг та недоліків універсального – часового та лексичного – можна виділити наступні кроки оптимальної оцінки алгоритму:

– парсинг програми на лексеми без підрахунку кількості операцій;

– пошук інструкцій, які не підтримуються на даному етапі (що означає, що даний алгоритм неможливо оцінити лексичним алгоритмом);

– у випадку, якщо інструкції, які на даному етапі не підтримуються алгоритмом були знайдені – перехід до оцінювання алгоритму часовим алгоритмом;

– у випадку, якщо алгоритм таких інструкцій не було знайдено – оцінювання алгоритму лексичним алгоритмом: парсинг лексем, їх обробка через пріоритетну чергу, підставляння початкових та кінцевих значень у цикли, тощо;

– підготовка класів задач для оцінки лексичним та часовим алгоритмом (більш прості – для лексичного алгоритму, та більш складні, тобто, з наявністю умовних операторів, рекурсії, тощо – для часового алгоритму);

– підготовка наборів даних для часового алгоритму (дані повинні мати пропорційний розмір та подібну структуру (наприклад: усі вхідні дані для даного алгоритму відсортовані), щоб збільшити точність вичислень.

Підготовка задач для оцінки алгоритмами важлива частина дослідження, тому що не всі алгоритми підлягають оцінці лексичним алгоритмом: якщо в задачі є оператори умовного переходу – то неможливо передбачити скільки разів, або чи взагалі будуть виконані інструкції під умовним оператором. Цей висновок виходить з проблеми зупинки, доведеної Аланом Тьюрінгом в 1936 році. Тому важливо підібрати такий клас задач, які можна вирішити прямими інструкціями.

Для часового алгоритму – нема обмежень по структурі програми, проте важливо правильно підібрати вхідні дані для тестування. Для того, щоб визначити складність [3] алгоритму часовим алгоритмом – необхідно виконати один і той же алгоритм на різних наборах вхідних даних, заміряти час виконання для кожного набору і побудувати графік функції f(n) = t, де n – розмір вхідних даних, t – час виконання. З отриманого графіку можна буде визначити формулу складності.

На дані також є обмеження – вони повинні бути консистентні між собою – тобто якщо виконати алгоритм на відсортованих даних, а потім на не відсортованих то час буде різний, тому при збільшенні даних потрібно зберігати відсортованість даних, порядок чисел, тощо.

1.2 Аналіз аналогів

На даний момент існують безліч веб-сервісів, що надають задачі пов’язані з реалізацією того чи іншого алгоритму, але ні один з них не бере на себе оцінювання складності алгоритму.

leetcode.com, hackerrank.com – одні з найвідоміших сервісів, які лише умовно порівнюють час виконання алгоритму відносно інших учасників, що вирішували дану задачу. В залежності від часу доби, результати можуть буди діаметрально протилежними. У вихідні дні ввечері та у понеділок вранці виконання одного алгоритму майже точно дасть абсолютно різні результати. Через навантаженість серверу у вихідні – час виконання буде низький і може програвати гіршим алгоритмам, які були виконані в той час, коли сервер був ненавантажений і тому швидко відпрацював (див. рис.1.2).

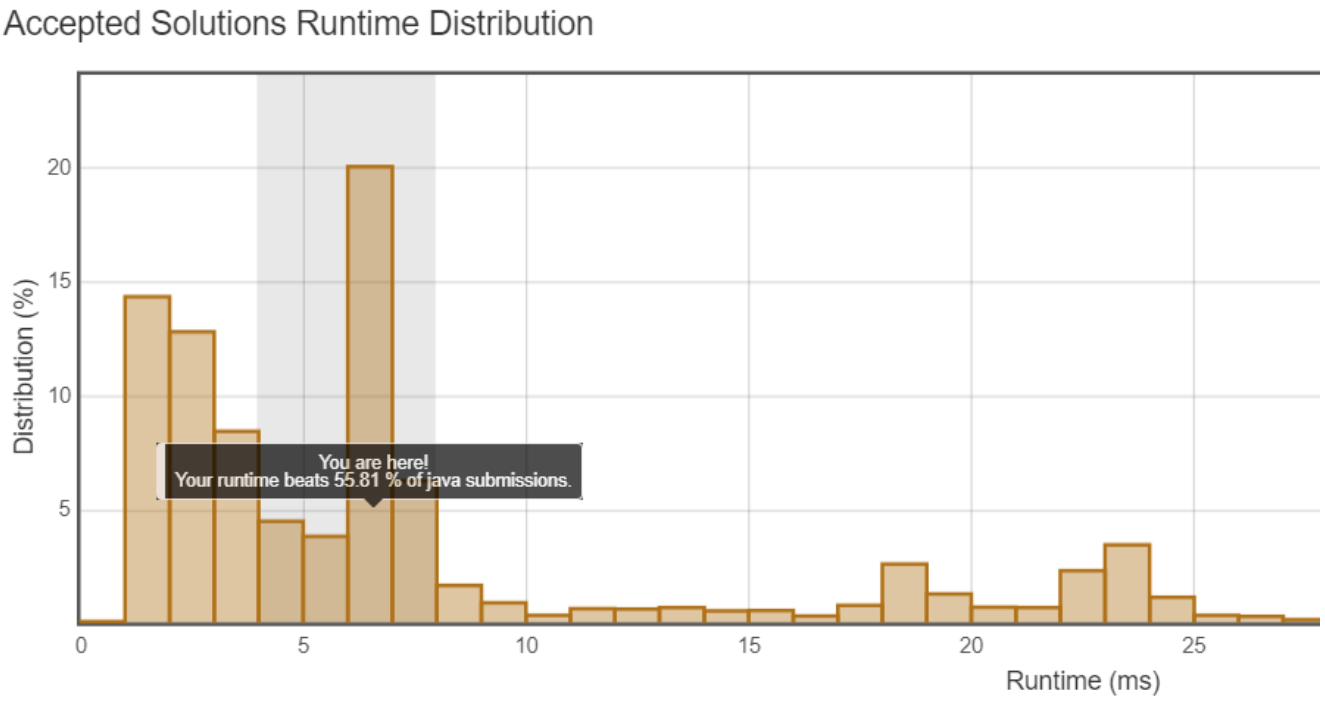

Рисунок 1.2 – Приклад оцінювання алгоритму на сайті leetcode.com.

Як видно з рисунку, оцінювання виконується лише порівняно з іншими учасниками системи, що є дуже нестабільним способом. Основною проблемою таких ресурсів є те, що для збільшення точності, якщо замірюванням часу виконання алгоритму займається декілька серверів – вони повинні бути ідентичними по обчислювальній потужності [4], або мати свою кореляцію часу на всі сервера. Такі ресурси мають обмеження по часу на виконання програми, після перевищення якого переривають виконання програми.

lizard.ws – ресурс для аналізу складності коду. Цей ресурс рахує кількість вкладеностей циклів, на основі чого робить висновок про складність коду (див. рис.1.3).

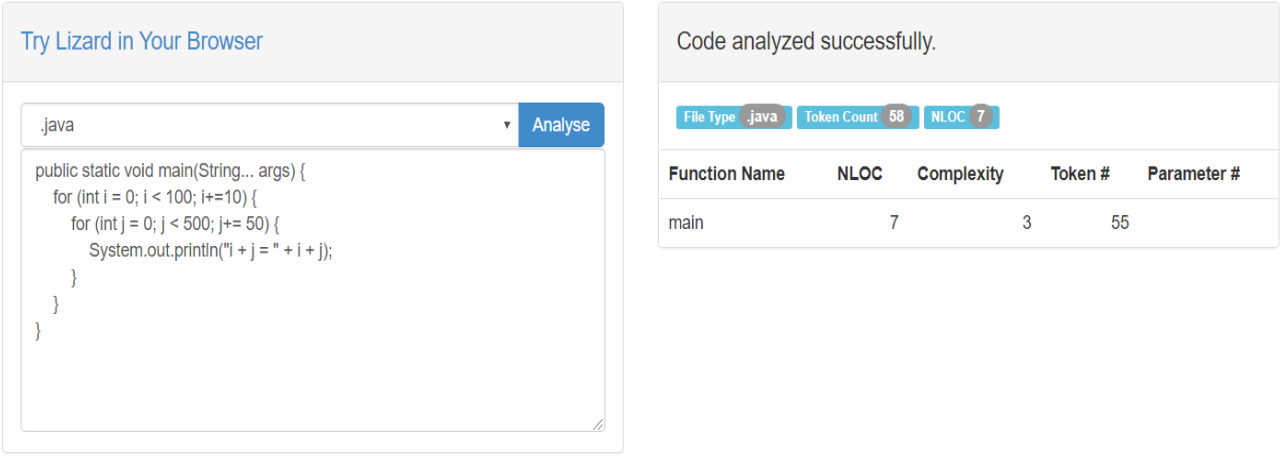

Рисунок 1.3 – Скріншот роботи сервісу lizard.ws.

Як видно зі скріншоту – програма порахувала кількість вкладених циклів і дала інформацію про складність. Проте лише з інформацією про кількість вкладеностей неможливо якісно оцінити складність коду, тому що як видно з коду на скріншоті – ітерація циклу виконується з кроком в $\frac{1}{10}$ частину від кінцевого значення індексу. Це означає, що справжня складність таких вкладених циклів буде n, при умові що n = 100.

До недоліків всіх вищеперерахованих сервісів можна віднести те, що вони не вміють вираховувати асимптотичну складність алгоритмів. Виходячи з цього – існує необхідність створити веб-систему, яка б надавала задачі для вирішення та могла б аналізувати рішення та надавати асимптотичну оцінку їх складності.

2.2 Постановка задачі

Метою роботи є дослідження підходів до оцінки складності алгоритмів та розробка програмного забезпечення для аналізу складності алгоритмів на основі проведених досліджень.

Програмне забезпечення має бути веб-додатком. Окрім основного функціоналу зв'язаного з аналізом складності алгоритмів необхідно передбачити набір алгоритмічних задач, рішення б яких було доцільно аналізувати даним додатком. Ця система може використовуватись для змагань програмістів між собою. Ця система буде корисним інструментом для розвитку аналітичного мислення та підвищення загального рівня спеціалістів. Система повинна аналізувати написаний код, розбиваючи його на лексеми та підраховуючи кількість операцій програми. У випадку, якщо роботу програми неможливо оцінити таким алгоритмом (за наявності операторів умовного переходу, рекурсії, тощо) – використати для оцінки алгоритм, який буде виконувати написану програму з різними наборами вхідних даних та аналізуватиме графік росту функції часу від розміру вхідних даних, що забезпечить велику точність оцінки.

Таким чином була поставлена наступна задача:

– провести аналіз лексичних структур для підтримки в базовій версії лексичного алгоритму;

– розробити базовий лексичним алгоритм оцінки складності коду;

– підібрати клас задач з базовою лексичною структурою для оцінки;

– доповнити алгоритм підтримкою більш складних лексичних структур;

– проаналізувати способи реалізації часового алгоритму (кількість замірянь часу, кількість даних, тощо);

– програмно реалізувати алгоритми.

До не функціональних вимог можна віднести технології розробки:

– реалізація основного функціоналу повинна бути виконана за допомогою мови програмування Java 11, фреймворку Spring Framework 5.0, Spring Boot 2.0;

– в якості сховища даних обрано NoSQL MongoDB та клауд-хостинг mLab.com;

– веб-частина додатку повинна бути реалізована за допомогою мови JavaScript та фреймворку React;

– в якості API обрано RESTful API;

– в якості серверу додатків обрано Tomcat.

2 ПЕРЕЛІК ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Призначення розробки

Веб-система оцінювання складності алгоритму призначена для широкого спектру користувачів: англомовний інтерфейс та задачі поділені за різним рівнем складності підійдуть для людей різного рівня підготовки з усього світу. Основною категорією користувачів передбачаються бути програмісти. В першому релізі передбачається підтримка однієї мови програмування для вирішення задач – Java, проте система потенційно може підтримувати багато мов.

Проблема оптимальності написаного коду дуже гостро стоїть у сучасному світі: безліч сервісів, що забезпечують сотні тисяч транзакцій за хвилину і більше дуже обережно підходять до питання продуктивності системи: збільшення швидкості виконання запиту навіть на 10% може дуже сильно зменшити затрати на систему. У сучасному айті – значна кількість розробників програмного забезпечення - це люди, які не отримували профільної освіти і навчались самі і тому дуже часто такі люди пропускали таку важливу частину програмування – як алгоритми і структури даних, тому наявність відповідної системи необхідна для заповнення пробілу в знаннях програмістів та удосконаленню навичок висококваліфікованих спеціалістів.

2.2 Вимоги до програмного продукту

Загальні відомості про розробку програмної системи полягають у наступному: веб-система повинна бути побудована за клієнт-серверною архітектурою. Серверна частина повинна використовувати мову програмування Java 11 та фреймворки Spring Framework 5.0 та Spring Boot 2.0. Клієнт повинен бути написаний за допомогою мови

програмування JavaScript та фреймворку React. API для взаємодії клієнта-сервера обрано RESTful[8], протокол обміну інформацією – HTTP. В якості сховища інформації використовується NoSQL MongoDB[9], як зручне сховище для зберігання неструктурованої інформації в великих об’ємах і зручному форматі. Для взаємодії з базою даних обрано Spring Data JPA та фрейморвк Hibernate для мапінгу об’єктів системи на таблиці бази даних. Система збірки проекту – Maven, система керуванян версіями – Git.

Основна функціональність даної системи полягає у наступному:

– виведення на екран списку доступних задач для вирішення;

– зчитування написаного коду;

– розбиття коду лексичним аналізатором;

– пошук непідтримуваних структур лексичним алгоритмом;

– оцінка коду часовим алгоритмом у разі якщо непідтримувані структури були знайдені;

– парсинг значень змінних;

– оцінка коду лексичним алгоритмом;

– оцінка коду часовим алгоритмом у разі, якщо лексичний не зміг оцінити;

– мапинг найбільш поширених формул складності до результуючої складності;

– відображення результатів оцінювання;

Отже, у ході розробки програмної системи необхідно проаналізувати предметну область лексичного аналізу коду, проаналізувати методи оцінювання вкладених інструкцій (використання спеціальних структур даних для зберігання стану даної лексеми), проаналізувати найпоширеніші функції складності та навчитись правильно переводити кількість операцій у функцію складності.

2.3 Вимоги до клієнта

Система розробляється у вигляді веб-додатку, що не потребує спеціального програмного забезпечення або високих вимог до апаратного обладнання для користування з системою. До переліку вимог можна віднести наявність веб-браузеру та наявність доступу до мережі Internet.

Веб-додатком можна користуватися як з персональних комп’ютерів так і з мобільних пристроїв, проте написання коду з мобільного телефону не є зручним і не рекомендується.

3 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

3.1 Дослідження лексикографічного алгоритму з метою оцінювання складності алгоритмів

В рамках роботи необхідно провести аналіз лексичного аналізу коду програми. Лексичний аналіз [5] – це процес розбиття коду на послідовність лексем [6]. Цю роботу виконує лексичний аналізатор [7]. Він читає код символ за символом, розпізнає лексеми та отримує послідовність токенів [8], що описують лексеми.

Лексема – це одиночна ідентифікуєма послідовність символів, наприклад, ключових слів (for, while, public, class, void, int), літералів (числа, строки), ідентифікаторів, операторів або пунктуаційних символів (таких як «{» та «(»).

Токен – це об’єкт, що описує лексему. Токен має тип (наприклад: ключове слово, літерал, ідентифікатор, оператор) та значення – справжні символи лексеми (наприклад int – це значення токену з типом ключове слово.

Для розпізнавання токенів лексичний аналізатор використовує лексичну граматику[9], набір всіх можливих лексем, що можуть бути в коді. Правила в лексичній граматиці часто трансформуються в автомат, що називається кінцевий автомат станів.

Його задача розпізнавати вхідні дані на валідність та зберігати стан і недопускати не валідні дані (наприклад закриваюча фігурна дужка не може стояти перед відкриваючою).

Лексична граматика в мовах програмування – це набір формальних правил, що визначає як валідна лексема в даній мові програмування може бути побудована. Наприклад, правила можуть заявляти, що строка це будь яка послідовність символів, що оточується подвійними кавичками з обох сторін, або те, що ідентифікатор не може починатися з цифри. Правила лексичної граматики часто виражаються з набором регулярних визначень (з англ. regular definitions[10]).

Регулярне визначення може виражатися регулярним виразом в даній мові програмування:

$$\text{l}letter = [a - zA - Z],$$

де «a-z» – діапазон, що означає усі символи латинського алфавіту нижнього регістру;

«A-Z» – верхнього регістру.

Регулярне визначення може використовуватися в регулярному виразі для будь-якого елементу в рамках тієї ж лексичної граматики:

$$letter = [a - zA - Z]$$

$$digit = [0 - 9]$$

$$identifier = (letter \mid \_)\ (letter \mid digit \mid \_)\ *,$$

де letter – означає будь який символ нижнього та верхнього регістру латинського алфавіту;

digit – цифру від 0 до 9;

identifier – валідна назва змінної.

Тобто правильний ідентифікатор повинен починатися з великої або малої літери латинського алфавіту або зі знаку нижнього підкреслювання, за яким слідують символи, цифри або знаки нижнього підкреслювання у кількості від 0 до безкінечності.

Як видно, в регулярному визначенні ідентифікатор використовує визначення символу та цифри в своєму правилі для того, щоб визначити що ідентифікатор це будь який рядок, що починається з символу або нижнього підкреслення за яким слідує символ, цифра або знак нижнього підкреслення, яке можна використати від нуля до безкінечної кількості разів.

3.2 Математичне моделювання

Основними сутностями предметної галузі в області оцінювання складності алгоритму, а отже і математичної моделі розроблюваної системи є оцінюваний алгоритм (A), набір лексем які є складовими частинами алгоритму (L), лексичний аналізатор, на основі якого реалізовується лексичний алгоритм (LA), лексичний алгоритм оцінювання складності коді (LCA), лексична граматика (LG), обмеження на код для коректної роботи алгоритму (RT), часовий алгоритм оцінювання складності коду (TCA) та набір найпоширеніших складностей (C).

Лексичний алгоритм можна виразити наступною формулою:

$$LCA = < A, L, LA, RT, LG, C, TCA >,$$

де A – алгоритм, що підлягає оцінюванню,

L – набір лексем, які є складовою будь-якого коду,

LA – лексичний аналізатор,

RT – обмеження, що накладаються на оцінюваний лексичним алгоритм (код)

LG – лексична граматика,

C – набір складностей, до яких буде приведений результат роботи програми,

TCA – часовий алгоритм оцінювання складності алгоритму.

В свою чергу, даний для оцінювання алгоритм (A), можна описати наступною формулою:

$$A = < L >,$$

де L – це набір лексем, які є основою та складовою частиною будь-якої програми і які будуть розпізнаватися лексичним аналізатором.

Лексичний аналізатор (LA), в свою чергу можна описати набором таких сутностей:

$$LA = < T, LG, RD >,$$

де *T* – набір токенів, отриманих з розпізнаних лексем;

LG – лексична граматика, тобто набір правил формальної граматики, що визначають синтаксис токенів;

RD – регулярні визначення, що визначають можливу структуру валідного токену. Регулярні визначення можна описати всіма можливими метасимволами та алфавітами.

В свою чергу, лексичну граматику можна описати як набір правил:

$$LG = < R >,$$

де R – набір правил, які описуються регулярними виразами, які задаються для даної системи і можуть бути описані як набір всіх можливих правил:

$$R = < primitiveType, identifier, equalOperator, binaryOperator,$$
$$unaryOperator, intNum, loop >,$$

де кожному з правил буде описаний регулярний вираз;

RD – регулярні визначення, що визначають можливу структуру валідного токену. Регулярні визначення можна описати всіма можливими метасимволами та алфавітами.

Обмеження системи можна описати наступною множиною:

$$RT = < CO, LS, RC, IL >,$$

де CO – оператори умовних переходів, які є основною причиною неможливості оцінювання алгоритму лексичним аналізом;

LS – ярликові вирази, тобто оператори, що використовуються для непослідовної зміни номеру інструкції, що неможливо передбачити;

RC – рекурсивні виклики методу, або виклики зовнішніх методів;

IL – безкінечні цикли.

Набір складностей (C) можна описати наступною п’ятіркою:

$$C = < O(1), O(\log n), O(n), O(n^2), O(n^3) >,$$

де O(1) – складність – константа. Наприклад, операція додавання чисел може вважатися константою;

O(log n) – складність логарифмічна. Наприклад, пошук елемента у бінарному дереві пошуку;

O(n) – складність лінійна. Наприклад, пошук елемента у невідсортованому масиві простим перебором елементів.

3.2 Дослідження роботи кінечного автомату станів для реалізації лексичних правил для парсингу коду

Для того щоб розпізнати токен описаний регулярним визначення, регулярний вираз в визначенні часто трансформується в КАСи – скінченні автомати станів (FSMs – Finite State Machines[11]). Результуючий КАС має скінченний набір станів, що включає начальний стан та набір можливих станів.

КАС переходить від одного стану до іншого шляхом «поїдання» одного з символів або елементів регулярного виразу. Після переходу з початкового стану в один з приймаючих станів виходить дійсний рядок, описаний регулярним виразом.

Наприклад, регулярний вираз «a | b» (a чи b) може бути представлений у вигляді наступного КАСу (див. рис. 3.1).

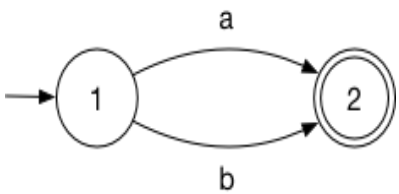

Рисунок 3.1 – Скінченний автомат станів для регулярного виразу «a | b»

Представлений вище КАС має два стани помічених номерами 1 і 2. Стрілка що вказує на 1 і приходить нізвідки означає, що 1 – це початковий стан, а внутрішнє коло в стані 2 означає, що 2 – це приймаючий стан цього КАСу.

Для ідентифікаторів, чисел, дужок та операторів нижче описані наступні регулярні визначення:

$$letter = [a - zA - Z]$$

$$digit = [0 - 9]$$

$$number = digit\ digit*$$

$$identifier = (letter \mid \_)\ (letter \mid digit \mid \_)*$$

$$operator = +|- \mid * \mid / \mid > \mid \geq \mid < \mid \leq \mid = \mid ==$$

$$parenthesis = (\mid),$$

де letter та digit були описані вище;

number – описує будь-яке ціле число4

operator – описує найбільш поширені математичні оператори;

parenthesis – дужки.

В рамках аналізу базового лексичного аналізатору для даного алгоритму необхідно на основі розпізнаних лексем навчитись добувати інформацію про значення змінних та підставляти їх у оператори циклів.

При пошуку лексеми з типом ідентифікатор необхідно створити об’єкт, що буде зберігати його значення.

Для обробки циклу, необхідно створити окреме регулярне визначення, що його описує (див. рис. 3.2).

```
primitiveType = (byte|char|short|int|long|float|double)
identifier = letter | (letter | digit | _)*
equalOperator = =
binaryOperator = + | - | * | / | > | >= | < | <= | = | == | += | -= | *= | /=
unaryOperator = ++ | --
intNum = -{0,1} digit digit*
loop = for \\s* \\( (primitiveType\\s* identifier\\s* equalOperator\\s* number ) | (identifier)| (identifier\\s*
binaryOperator\\s* intNum\\s*; \\si*dentifier\\s* binaryOperator\\s* intNum\\s* ; \\s*identifier\\s* (unaryOperator
|(binaryOperator number)\\s* \\)
```

Рисунок 3.2 – Регулярні визначення, що описують правила для циклу for.

Як видно з рисунку – для того, щоб описати оператор циклу – необхідно описати правила для всіх лексем, що використовуються при роботі з циклом. На першому етапі буде додана підтримка тільки для оператору циклу for. Регулярний вираз покриває не всі, але переважну більшість використань циклу в реальному житті, таких як:

- for (int i = 0; i <= 1000; i++);
- for (x; x > 5; x -= 1);
- for (y+1; y > -5; y/2).

Програмна реалізація регулярного виразу та парсингу операторів циклів буде описана в далі.

3.3 Аналіз обмежень, що накладаються на код для оцінки його лексичним алгоритмом

Оскільки доведено, що проблему зупинки[11] вирішити неможливо, тобто неможливо визначити чи завершить програма своє виконання чи ні, а саме ця проблема і є основною перешкодою у написанні універсального алгоритму для оцінки складності написаного коду – було прийнято рішення проаналізувати усі причини, які роблять цю проблему актуальною та накласти обмеження у вигляді заборони використання певних конструкцій в коді що аналізується.

Хоча ці обмеження не можуть гарантувати, що програма точно завершить своє виконання і тому можливо порахувати кількість операцій (ітерацій) – це значно підвищить шанс, що програма буде валідною для даного алгоритму. У разі, якщо обмеження не зможуть надати достатньої гарантії – є сенс ввести затримку на виконання алгоритму, перевищуючи яку – можна припустити, що програма попадає під класифікацію undecidable[12], тобто такі, які неможливо визначити.

До обмежень можна віднести:

– оператори умовного переходу (if, else, ternary operators);

– виклики функцій з середини оцінюваної функції;

– ярликові вирази (label statements [13]);

– вічні цикли (неможливо точно розпізнати чи буде даний цикл вічним, оскільки умова циклу може залежати від вхідних даних, тому підтримуються найбільш очевидні способи визначення вічних циклів (див. рис. 3.3.).

```
4
5       public static void main(String[] args) {
6
7           while (true) {
8               // this loop will never halt
9           }
10
11          for (;;) {
12              // this loop will never halt
13          }
14
15          do {
16              // this loop will never halt
17          } while (true);
18
19      }
```

Рисунок 3.3 – Приклади вічних циклів

В результаті аналізу було виявлено обмеження, через які неможливо оцінити довільну програму лексичним алгоритмом, і для того, щоб розширити функціонал системи було вирішено доповнити алгоритм оцінювання часовим алгоритмом – тобто у разі неможливості оцінити програму лексичним алгоритмом – вивести формулу зростання часу виконання від кількості вхідних даних.

Це дозволить алгоритму не бути вузькоспеціалізованим і бути застосованим для оцінювання будь-яких алгоритмів.

3.4 Дослідження та аналіз способів реалізації часового алгоритму

Ті алгоритми, які попали під класифікацію undecidable і не змогли бути оцінені лексичним алгоритмом – переходять на виконання до часового алгоритму.

Задача часового алгоритму – засікти час виконання даної програми декілька разів для побудування графіку росту функції, з якого можна буде визначити формулу, тобто коефіцієнт з яким зростає час виконання програми залежно від зростання кількості даних.

Аналіз часу виконання (Run-time analysis[13]) – це теоретична класифікація, яка оцінює і передбачає збільшення часу виконання (або часу виконання) алгоритму в міру збільшення його вхідного розміру (зазвичай позначається як n).

Ефективність виконання часу є великою цікавістю в галузі інформатики. Програма може зайняти секунди, години, а то й роки, щоб закінчити виконання, залежно від того, який алгоритм використовується. Хоча методи програмного профілювання можуть використовуватися для вимірювання часу виконання алгоритму на практиці, вони не можуть надати дані про терміни для всіх нескінченно багатьох можливих входів; останнього можна досягти лише теоретичними методами аналізу часу виконання.

Оскільки алгоритми не залежать від платформи (тобто даний алгоритм може бути реалізований на довільній мові програмування на довільному комп'ютері, на якому працює довільна операційна система), то дуже важливо не просто опиратися на час виконання програми, а дивитися на те, як зростає час виконання в залежності від зростання даних.

Візьмемо для прикладу програму, яка шукає конкретний запис у відсортованому списку розміром n. Припустимо, ця програма була реалізована на комп’ютері A, найсучаснішій машині, що використовує лінійний алгоритм пошуку, та на комп'ютері B, набагато повільнішій машині, використовуючи алгоритм

двійкового пошуку. Тестування на двох комп’ютерах, на яких запущені відповідні програми, може виглядати приблизно так (див. табл. 4.1).

Комп'ютер А, запускаючи програму лінійного пошуку, демонструє лінійну швидкість зростання. Час виконання програми прямо пропорційний розміру вхідних даних. Як видно з таблиці – незважаючи на те, що комп’ютер А більше ніж в 10 тисяч разів швидший комп’ютера В – кожне подвоєння розмірів вхідних даних збільшує час виконання алгоритму комп’ютером А в 4 рази, в той час як алгоритм виконання пошуку у відсортованому списку комп’ютера В демонструє логарифмічний ріст.

З вищеописаного прикладу можна зробити такі висновки, що одного разу виконання програми недостатньо для оцінки її складності, бо кожен комп’ютер має свої ресурсні можливості, а також те, що оптимальне написання алгоритмів може значно зекономити ресурси і тому важливо завжди максимально оптимізувати будь-які високонавантажені функції програми.

Таблиця 3.1 – Таблиця залежності часу виконання пошуку елементу у відсортованому списку від кількості даних для різних способів пошуку та комп’ютерах різної потужності.

n (розмір списку)	Комп’ютер А час виконання (в наносекундах)	Комп’ютер В час виконання (в наносекундах)
16	8	100,000
63	32	150,000
250	125	200,000
1,000	500	250,000
...	...	...
1,000,000	500,000	500,000

Продовження таблиці 3.1

n (розмір списку)	Комп’ютер A час виконання (в наносекундах)	Комп’ютер B час виконання (в наносекундах)
4,000,000	2,000,000	550,000
16,000,000	8,000,000	600,000
...	...	...
63,072 × 1012	31,536 × 1012 ns, or 1 year	1,375,000 ns, or 1.375 milliseconds

Як видно з таблиці – для правильної реалізації часового алгоритму необхідно не просто заміряти час виконання програми, а підготувати набори вхідних даних різних розмірів щоб мати можливість проаналізувати ріст часу виконання від росту розмірів вхідних даних.

Для того щоб мати можливість оцінити складність алгоритму часовим способом – необхідно проаналізувати типові часові складності (див. табл. 3.2).

Таблиця 3.2 – Таблиця найбільш поширених складностей алгоритмів

Назва	Обчислювальна складність	Час виконання ($T(n)$)	Приклади часу виконання	Приклади алгоритмів
сталий час		$O(1)$	10	Визначенно парності числа (у двійковому запису)

Продовження таблиці 3.2

Назва	Обчислювальна складність	Час виконання ($T(n)$)	Приклади часу виконання	Приклади алгоритмів
логарифмічний час	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Двійковий пошук
лінійний час		$O(n)$	n	Пошук найбільшого або найменшого елементу невпорядкованого масиву
квадратичний час		$O(n^2)$	n^2	Сортування бульбашкою; Сортування включенням
кубічний час		$O(n^3)$	n^3	Безпосереднє множення двох матриць розміру $n \times n$. Обчислення часткової кореляції.

Як видно з таблиці – будь-який алгоритм можна описати типовою складністю наведеною в таблиці. Наприклад, якщо алгоритм має складність $2n^2$ – то складність можна вважати квадратичною – тобто n^2 – оскільки на фоні розмірів вхідних даних коефіцієнтом 2 можна легко знехтувати.

Існують ще часові складності, які не були включені до даної таблиці, бо в рамках даної роботи вони використані не будуть.

3.5 Розробка лексичного алгоритму оцінювання складності на основі лексичного аналізу

Для реалізації відповідного алгоритму необхідно поєднати лексичний аналіз та спеціальну обробку лексем для зберігання для кожної з них її актуального значення та розробити спосіб правильного підрахунку операцій у разі наявності вкладених операцій чи циклів.

Для кожної можливої лексеми, яка грає роль у підрахунку операцій – необхідно створити відповідний об’єкт у програмі, який буде зберігати як тип і назву лексеми так і її значення.

Для швидкого доступу до змінних було обрано зберігати їх у хеш-мапі[14] – як структуру даних з доступом до даних з часовою складністю O(1). Також, перевага хеш-мапи в тому, що якщо значення змінної зміниться – то при додаванні лексеми до хеш-мапи перезапише значення ідентифікатору, а не створить новий запис.

Оскільки задача алгоритму – не просто розбивання програми на лексеми для перевірки валідності синтаксису – є сенс комбінувати лексеми таким чином, щоб найоптимальніше реалізувати алгоритм.

Наприклад, вираз

$$int\ x = 10;,$$

де int – лексема – ключове слово (тип даних);

x – ідентифікатор;

= – операція;

10 – число;

; – сепаратор.

В такому виразі немає сенсу обробляти окремо кожну лексему, а є сенс об’єднати інструкції такого вигляду в лексему з назвою «ініціалізація змінної»

(variable initialization), створити об’єкт VarInit з полями назви змінної, її типу та значення.

Алгоритм можна описати блок-схемою (див. рис 3.4).

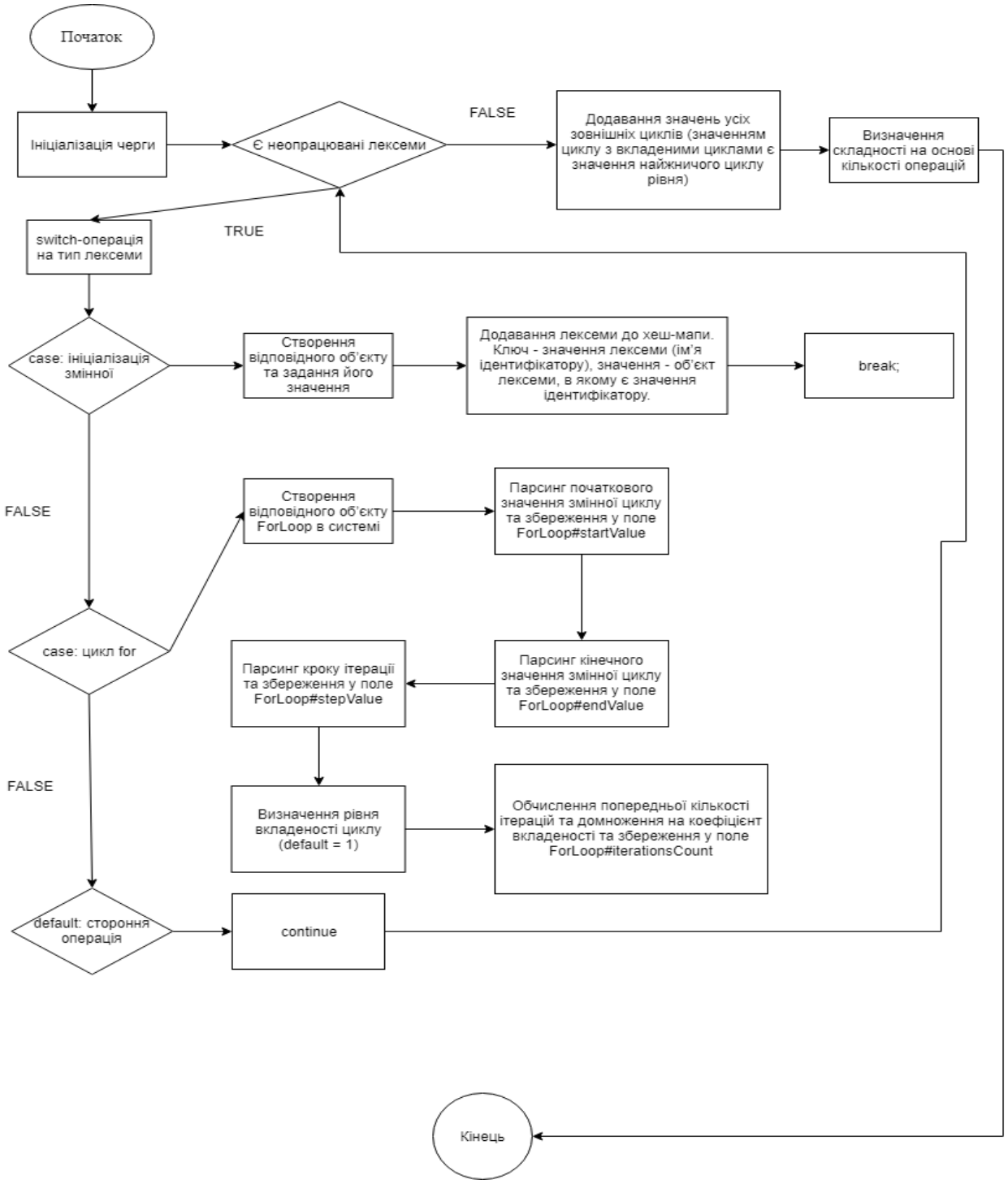

Рисунок 3.4 – Блок-схема роботи лексичного алгоритму.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Розробка бази даних

При виборі сховища даних було проаналізовано можливу структуру та прийнято рішення використовувати документо-орієнтовану NoSQL базу даних – MongoDB. Для зберігання бази даних було обрано безкоштовний клауд-хостинг mLab.com.

Оскільки база даних – NoSQL, дані зберігаються у JSON-форматі (див. рис. 4.1).

Рисунок 4.1 – Формат даних бази даних.

У базі даних необхідно зберігати всі можливі складності, що на даний момент підтримуються програмною реалізацією. Було додано п’ять складностей (див. рис. 4.2).

_id	complexity
linear	n
logarithmic	log(n)
constant	1
quadratic	n^2
cubic	n^3

Рисунок 4.2 – Таблиця складностей алгоритмів.

Для підтримки часового алгоритму необхідно підготувати набори даних на яких будуть виконуватись алгоритми для заміряння часу виконання (див. рис. 4.3).

_id	size	sorted
1	10	true
2	100	true
3	1000	true
4	10000	true

Рисунок 4.3 – Вхідні дані для тестування часового алгоритму.

Для первинного тестування було обрано тестувати алгоритми на відсортованих масивах з різницею розмірів в один порядок (див. рис. 4.4).

_id	size	sorted	data
1	10	true	["1","2","3","4","5","6","7","8","9","10"]
2	100	true	["1","2","3","4","5","6","7","8","9","10","11","12","13","14","15","16","17","18","19","20","21","22","23","24","25","26","27","28","29","30","31","32","33","34","35","36","37","38","39","40","41","42","43","44","45","46","47","48","49","50","51","52","53","54","55","56","57","58","59","60","61","62","63","64","65","66","67","68","69","70","71","72","73","74","75","76","77","78","79","80","81","82","83","84","85","86","87","88","89","90","91","92","93","94","95","96","97","98","99","100"]

Рисунок 4.4 – Приклади відсортованих масивів.

4.2 UML-моделювання

Для моделювання програмного забезпечення був використаний пакет моделювання Microsoft Visio 2010 для створення UML-діаграм [15]. UML – мова графічного опису для об'єктного моделювання в області розробки програмного забезпечення.

UML є мовою широкого профілю, це відкритий стандарт, який використовує графічні позначення для створення абстрактної моделі системи, званої UML-моделлю. На основі аналізу концептуальної моделі та постановки задачі під час проектування програмного забезпечення системи пошуку плагіату, були створені такі UML діаграми як: діаграма варіантів використання, діаграма компонентів та діаграма активності.

Діаграма варіантів використання (прецедентів) – це UML діаграма, на якій зображено відношення між акторами та прецедентами в системі. Суть даної діаграми полягає в наступному: проектована система представляється у вигляді множини сутностей та акторів, що взаємодіють із системою за допомогою так званих варіантів використання.

Варіант використання (Use Case) служить для опису сервісів, які система надає актору. Іншими словами, кожен варіант використання визначає деякий набір дій, які виконує система при взаємодії з актором. Обмеження на способи взаємодії між акторами та варіантами використання не накладаються.

Нижче зображена діаграма варіантів використання (див. рис. 4.5) де можна бачити одного актора (користувача системи) та основні операції, які забезпечує система.

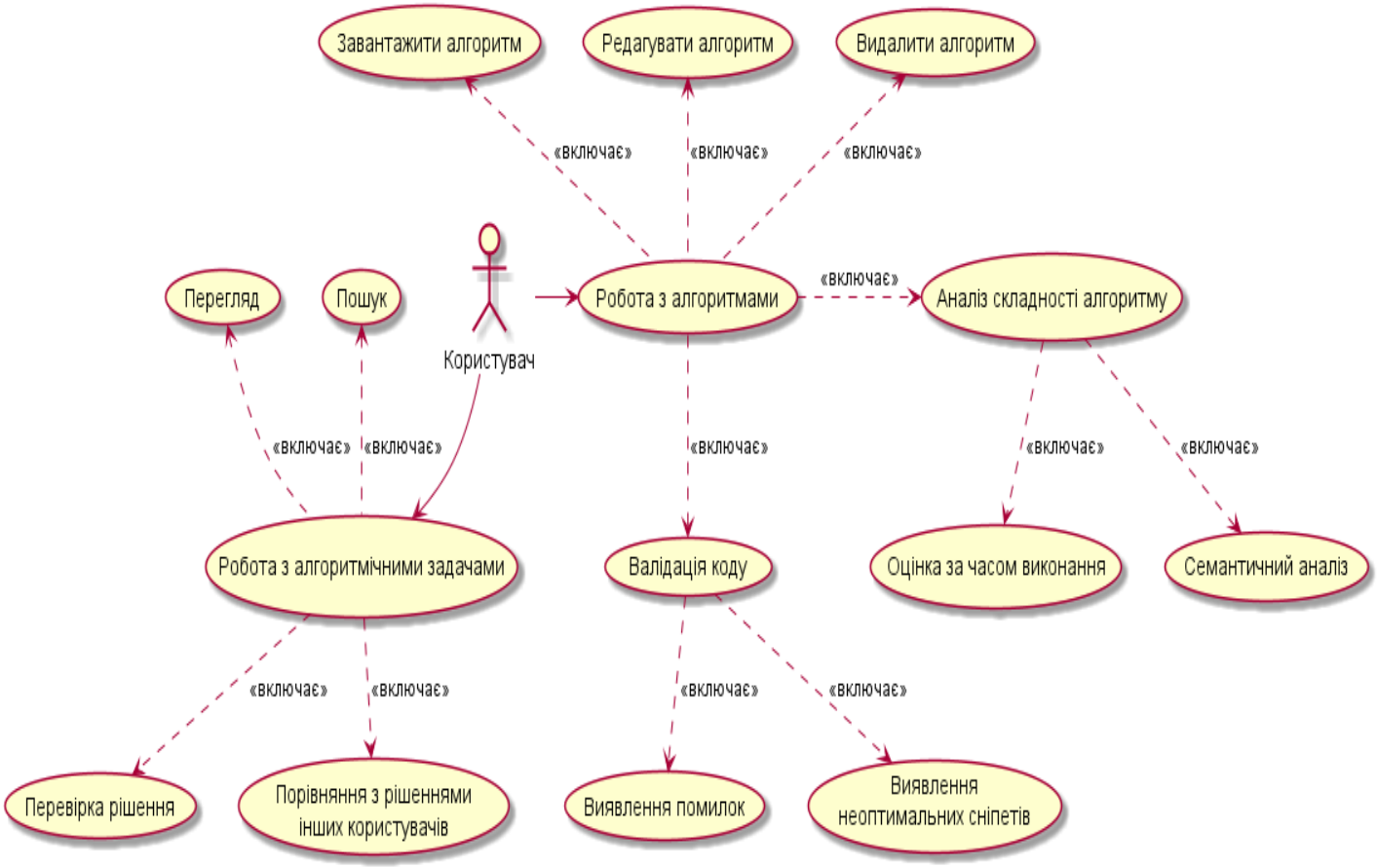

Рисунок 4.5 – Use-case діаграма функціоналу системи

Як видно з діаграми, в системі присутній один актор, що користується функціоналом системи:

– отримати задачу для її вирішення;
– перевірити рішення на його правильність;
– оцінити складність написаного коду;
– провалідувати код на наявність помилок та неоптимальних сніпетів;
– виконати семантичний аналіз програми для визначення її складності;
– виконати часовий аналіз програми для визначення її складності.

Однією з важливих діаграм є діаграма компонентів, яка демонструє всі компоненти системи, їх взаємодію між собою та дає чітке представлення про роботу системи (див. рис. 4.6).

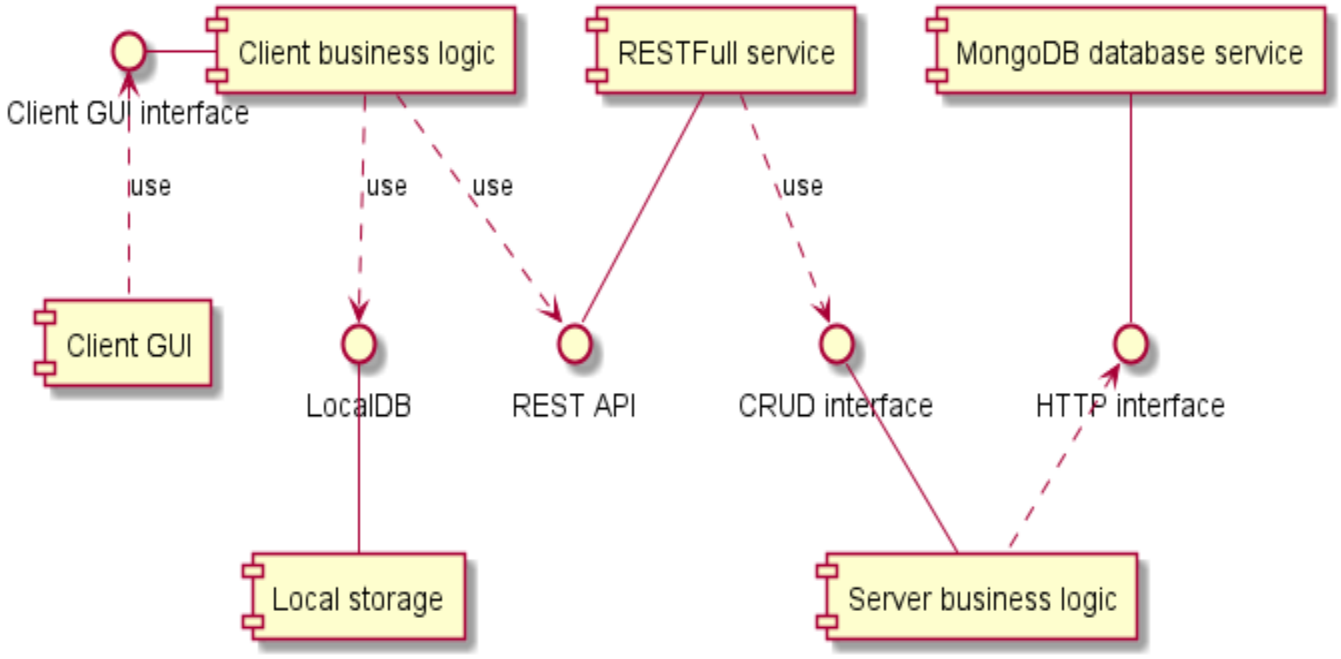

Рисунок 4.6 – Діаграма компонентів системи

Як видно з діаграми компонентів – користувач користується інтерфейсом для взаємодії з бізнес-логікою системи. Для взаємодії бізнес-логіки клієнта та сервера - використовується RESTful сервіс, взаємодія з яким відбувається за допомогою REST API завдяки HTTP запитам. Сервер в свою чергу взаємодіє з сервісом Mongo бази даних через HTTP інтерфейс.

Для розуміння внутрішньої структури системи можна використати діаграму класів, яка є структурною діаграмою мови UML та демонструє основні класи системи, їх ієрархічну структуру та їх взаємодію між собою (див. рис. 4.7).

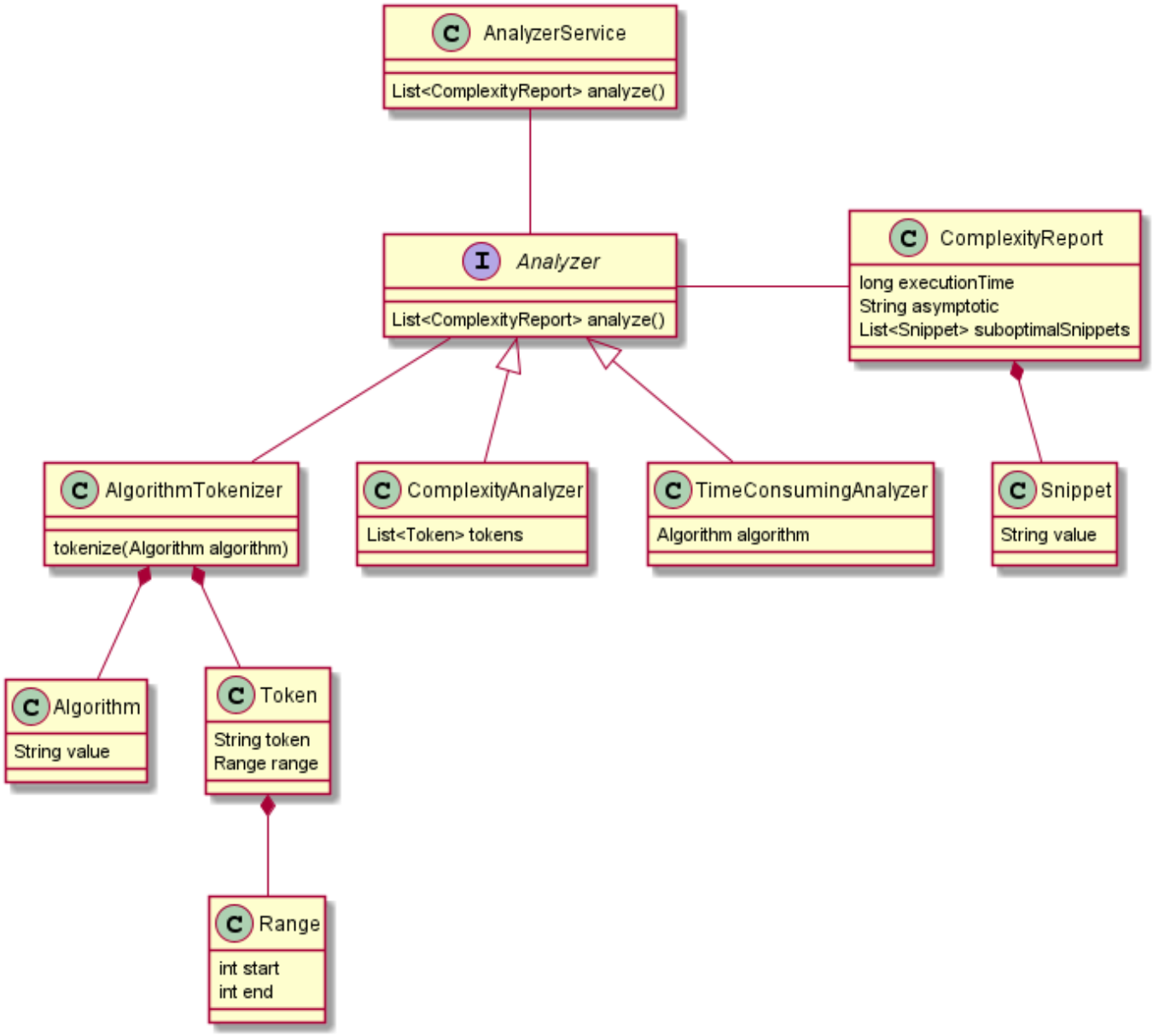

Рисунок 4.7 – Діаграма класів системи

Як видно з діаграми наведеної вище, існує єдиний інтерфейс що надає метод для оцінки складності коду. Інтерфейс має декілька реалізацій для часового та семантичного аналізу програми.

4.3 Розробка клієнтської частини

Клієнтська частина розроблена за допомогою фреймворка React. На наведеному скриншоті (див. рис. 4.8) видно описання задачі та текстове поле для вводу рішення, що має підсвічування ключових слів мови програмування.

Також була реалізована автотабуляція для зручності написання коду. При натисненні кнопки Submit відбувається відправлення написаного коду на сервер для подальшого аналізу та повернення результату оцінювання коду.

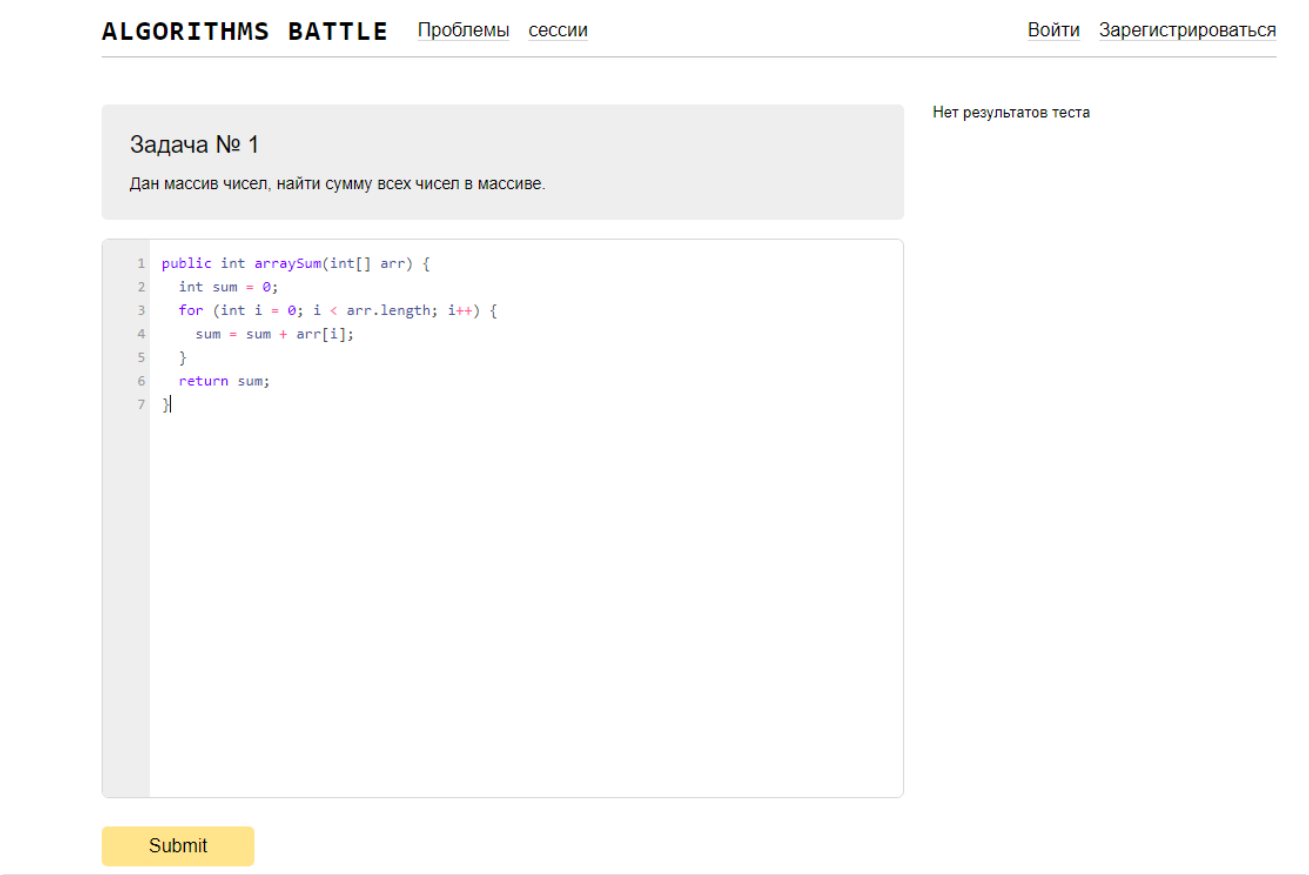

Рисунок 4.8 – Сторінка для написання алгоритму.

Результатом роботи системи є сторінка з результатами оцінювання та складності (див. рис. 4.9), що містить наступні пункти:

– назву задачі - «Task №1 - arraySum»;
– алгоритм, яким виконувалось оцінювання – лексичний аналізатор
– складність коду – O(n);
– додаткові тести, на яких тестувалась програма.

Рисунок 4.9 – Сторінка результатів.

4.4 Розробка серверної частини

Серед безлічі способів реалізації сервісів необхідно вибрати найбільш підходящий для наших потреб. Існує класифікація сервісів в межі Інтернет запропонована Леонардом Річардсоном, що передбачає три рівні зрілості на основі підтримки HTTP, гіпермедіа та URI. Найбільш простими сервісами є такі, що мають єдиний URL та які використовують один метод HTTP для передачі повідомлення

(частіше за все метод POST). Для прикладу, переважна більшість веб-сервісів використовують один URI та HTTP POST для передачі SOAP-повідомлень, не використовуючи інших методів HTTP. XML-RPC та Plain Old XML (POX) використовують подібний спосіб: HTTP POST запит з даними у обов'язково XML [16] (Extensible Markup Language) форматі, що працює з єдиним URI, та відповіддю у тілі HTTP повідомлення (також у XML форматі).

Наступний рівень сервісу використовує багато URI, але тільки один HTTP метод. На відміну від сервісів нульового рівня, перший рівень ділить сервіс на багато логічних ресурсів. Сервіс другого рівня має багато URI ресурсів. Такі сервіси підтримують декілька HTTP методів для кожного ресурсу. До цього рівня включаються CRUD (Create Read Update Delete) сервіси.

Сервіс найвищого рівня підтримує поняття гіпермедіа. Тобто представлення зберігають URI посилання, які можуть бути цікавими для споживача. Для реалізації сервісів цього рівня використовується концепція, яка дістала назву REST.

REST (скор. англ. Representational State Transfer, «передача стану подання») – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів. Був описаний і популяризований у 2000 році Роєм Філдінгом (Roy Fielding), одним із творців протоколу HTTP. Найвідомішою системою, побудованою переважно за архітектурою REST, є сучасна Всесвітня павутина. Антиподом REST є підхід, заснований на виклику віддалених процедур (Remote Procedure Call, RPC). Підхід RPC дозволяє використовувати невелику кількість мережевих ресурсів з великою кількістю методів і складним протоколом.

Систему пошуку плагіату слід реалізовувати таким чином, аби вона надавала RESTFull API за допомогою протоколу HTTP. Для цього доцільно буде використовувати Spring MVC. Він дозволяє за допомогою досить зручного, гнучкого та потужного інструментарію описувати веб-сервіси. API системи наведений в таблиці 4.1.

Таблиця 4.1 – URI та методи системи оцінювання складності коду

URI	Методи HTTP	Опис
/api/algorithm	GET, POST	Використовується для отримання існуючих алгоритмів (задач) системи та для додавання нових.
/api/algorithm/{id}	GET, DELETE	Повертає алгоритм (задачу) за заданим ідентифікаційним номером а також видаляє вказаний документ.
/api/complexity/source	POST	Відправляє рішення задачі у вигляді написаного вхідного коду на сервер для подальшого аналізу лексичним чи часовим алгоритмом.
/api/complexity/ {id}	GET	Повертає складність оціненого програмного коду з уточненням яким саме алгоритмом було проведено оцінювання.

Для використання серверного API, клієнту необхідно посилати відповідні запити на URI, дані необхідно розташовувати у тілі запиту у форматі JSON.

4.4 Опис основних сутностей та алгоритмів системи

Оскільки задача і лексичного і часового алгоритму зводиться до виконання однієї дії – оцінювання складності коду – то є сенс створити єдиний інтерфейс для будь-якої реалізації алгоритму оцінювання складності:

```
public interface CodeComplexityAnalyzer {
    Complexity calculateCodeComplexity(String source);
}
```

Єдиний метод інтерфейсу приймає на вхід код програми у вигляді строки та повертає її складністю.

Для зберігання складності коду використовується клас Complexity, який реалізований у вигляді перечислення (enum) та зберігає усі типові складності що будуть використані у нашій програмі:

```
public enum Complexity {
    CONSTANT("O(1)"),
    LOGARITHMIC("O(logN)"),
    LINEAR("O(n)"),
    QUADRATIC("O(n^2)"),
    CUBIC("O(n^3)"),
    EXPONENTIAL("O(2^poly(n)"),
    FACTORIAL("O(n!)");

    private String complexity;

    Complexity(String complexity) {
        this.complexity = complexity;
    }

    private static final Map<String, Complexity> lookup = new
HashMap<>();

    static {
        for (Complexity d : Complexity.values()) {
            lookup.put(d.getComplexity(), d);
        }
    }

    public String getComplexity() {
```

```
        return complexity;
    }

    public static Complexity get(String complexity) {
        return lookup.get(complexity); }
}
```

Клас LexicalCodeComplexityAnalyzer представляє собою клас з реалізацією лексичного алгоритму оцінювання складності коду. Основною ідеєю є обробка кожної лексеми відповідно до її призначення, кількість кроків цикла залежить від початкового та кінцевого значення змінної в ньому, а також від рівня вкладеності.

Нижче наведено приклад обробки лексем:

```
while (!lexemQueue.isEmpty()) {
    Lexem lexem = lexemQueue.poll();
    switch (lexem.getLexemName()) {
        case INPUT_SOURCE:
            inputSize = getInputSize(lexem);
        case VARIABLE_LEXEM:
            VariableLexem variableLexem =
castLexemToVariableLexem();
            variableLexemHashMap.put(lexem.getLexemName(),
variableLexem);
        case FOR_LOOP:
            ForLoopLexem forLoopLexem =
castLexemToForLoopLexem();
            forLoopLexemHashMap.put(lexem.getLexemName(),
forLoopLexem);
            forLoopLexemsStack.push(forLoopLexem);
        break;
        case FOR_LOOP_END:
            forLoopLexemsStack.pop();
        case UNDEFINED:
            continue;
    }
},
```

Як видно з наведеного вище коду, кожна лексема приводиться до одного з типів:

– вхідні дані. У цьому випадку, за допомогою рефлексії визначається тип вхідної колекції та визначається розмір вхідних даних – виклик методу size() для списків та поля length для масивів;

– змінна. У цьому випадку, при виклику методу castLexemToVariableLexem() виконується приведення лексеми до типу змінної та пошук значення цієї змінної. Якщо поле не було ініціалізовано і в лексемі є знак дорівнює – то полю класа зберігається початкове значення змінної, яке буде використано для пошуку кількості ітерацій в циклі. Потім лексема додається до мапи для подальшої підстановки у цикл;

– цикл. У цьому випадку, методом castLexemToForLoopLexem() виконується приведення лексеми до типу циклу та пошук початкових, крокових та кінцевих значень циклу. У випадку, якщо змінні були ініціалізовані ззовні – далі буде підстановка значень з колекції лексем змінних, що були описані вище;

– закриваюча дужка циклу. Видаляє останній цикл зі стеку для зменшення рівня вкладеності;

– будь-яка інша лексема. Оскільки для цілей системи нам не важливі ніякі інші лексеми – переходимо до обробки наступної лексеми.

Після обробки всіх лексем необхідно проітеруватися по всім циклам та пошукати значення змінних, що використовуються в циклі в колекції змінних, наповненої при обробці лексем. Після цього пройтись по стеку циклів та піднести кількість операцій у степінь вкладеності:

```
for (Map.Entry<Lexems, ForLoopLexem> entry :
forLoopLexemHashMap.entrySet()) {
    ForLoopLexem forLoopLexem = entry.getValue();
    String variableName = forLoopLexem.getLoopIndexName();
    if (variableLexemHashMap.containsKey(variableName)) {
        VariableLexem variableLexem =
variableLexemHashMap.get(variableName);
        forLoopLexem.setStartValue(variableLexem.getInitValue());
        forLoopLexem.setStepValue(variableLexem.getStepValue());
        forLoopLexem.setEndValue(variableLexem.getEndValue());
    }
    int levelOfNestedness = 0;
    while (!forLoopLexemsStack.empty()) {
        levelOfNestedness = levelOfNestedness + 1;
        ForLoopLexem loopLexem = forLoopLexemsStack.pop();
    }
    iterationsCount = (int) Math.pow(iterationsCount,
```

```
levelOfNestedness);
}
```

У випадку якщо існують конструкції що роблять неможливим оцінити алгоритм лексичним алгоритмом – виконується заміряння часу виконання програми на різних вхідних даних. Спочатку код компілюється для перевірки чи можливо його запустити чи ні:

```
private String compileSource(Path javaFile, String gameName) throws
IOException {
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    String fileName = javaFile.toFile().getPath();
    PrintStream out = new PrintStream(
            new FileOutputStream(
                    fileName.substring(0, fileName.indexOf(".")) +
EXTENSION_TXT),
            true);
    System.setErr(out);
    int result = compiler.run(null, null, null,
            javaFile.toFile().getAbsolutePath());

    out.close();
    if (result == 0) {
        String programResult = javaFile.getParent()
                .resolve(gameName + EXTENSION_CLASS).toString();
        System.setErr(System.err);
        return programResult;
    } else {
        return new String(Files.readAllBytes(
                Paths.get(fileName.substring(0, fileName.indexOf("."))
                        + EXTENSION_TXT)));
    }
}
```

У випадку успішної компіляції – тобто числовому результаті нуль – виконується запуск програми з тестами:

```
private List<TestResult> runClassWithTests(Path javaClass, String
gameName, Task task)
        throws ClassNotFoundException, NoSuchMethodException,
MalformedURLException {
    URL classUrl = javaClass.getParent().toFile().toURI().toURL();
    URLClassLoader classLoader = URLClassLoader.newInstance(new
URL[]{classUrl});
```

```
    Class<?> clazz = Class.forName(gameName, true, classLoader);
    Class inputParameterType =
typeManager.getTypes().get(task.getInputType());
    Method shouldBeRanMethod =
clazz.getDeclaredMethod(task.getMethodName(),
            inputParameterType);
    List<Test> testList = task.getTest();
    List<TestResult> testResults = new ArrayList<>();
    runTests(testList, shouldBeRanMethod, inputParameterType, clazz,
testResults);

    checkResults(testList, testResults);
    return testResults;
}
```

Результатом роботи є список результатів тестів, які містять очікуємий результат, актуальний та флажок що індикує пройшов тест чи ні. У разі успішного проходження тестів – виконується заміреннія часу виконання програми:

```
Map<Double, Integer> runningTimesAndSizes =  runCode(classPath,
gameName, task, inputSources);
Complexity complexity =
Complexity.calculateComplexityFromListOfPoints(runningTimesAndSizes)
```

З отриманої колекції часу та розміру вхідних даних вираховується приблизна функція росту часу від кількості даних. Час виконання можна вважати абсцисою, а розмір вхідних даних – ординатою, і на основі цих точок можна порахувати формулу функції. Для цього я використав вбудовану функцію сторонньої бібліотеки та просто передав на вхід колекцію точок:

```
Map<Double, Integer> runningTimesAndSizes =  runCode(classPath,
gameName, task, inputSources);
Complexity complexity =
Complexity.calculateComplexityFromListOfPoints(runningTimesAndSizes);
```

5 ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

Тестування програмного забезпечення (ПЗ) – це процес дослідження ПЗ з метою отримання інформації про стан програмного продукту, його якість, а також встановлення факту відповідності даного продукту специфікації, та вимогам замовника [26].

Тестування на сьогоднішній день це складний та багатогранний процес, що включає декілька видів та підходів до аналізу програмного забезпечення, так як не можливо обійтися лише одним підходом, він не зможе однозначно та повністю виявити всі дефекти та встановити коректність функціоналу програми що аналізується.

Практичний підхід до тестування орієнтований на проведення спеціалістами з тестування ПЗ тестових робіт. Швидкість і ефективність розробки ПЗ залежить від того наскільки процес тестування поєднується із загальним життєвим циклом розробки ПЗ і від ефективності використання технології тестування.

Необхідними умовами для тестування є наявність:

– об'єкта тестування, доступного для проведення іспитів;

– виконавців (залежно від виду іспитів що проводяться ними можуть бути, людина, машина, або їх сумісна праця).

Достатніми умовами для тестування є наявність:

– об'єкта тестування, доступного для проведення іспитів;

– виконавця (залежно від виду діяльності на різних фазах ним може бути людина, машина, або їх сумісна праця);

– плану тестування;

– тест кейсів / тестів;

Тест план був розроблений для виявлення недоліків в архітектурі програми та її функціонального оснащення, а також для вдосконалення роботи компонентів програми. Під час тестування було проведено декілька видів тестування. А саме:

– функціональне тестування;

– тестування за методом чорної скриньки;

– тестування за методом білої скриньки.

Функціональне тестування – це вид тестування ПЗ що дозволяє встановити відповідність реалізованих функціональних вимог до програмного продукту в той мірі, в якій вони потрібні користувачам. Функціональні вимоги визначають, що саме повинне виконувати ПЗ.

Програмне забезпечення повинно мати такі функції:

– створення сесії для змагання з іншим гравцем;

– завантаження опису задачі для вирішення;

– можливість вводу коду з автоформатуванням та табуляцією;

– відправлення коду на перевірку системою;

– повернення результату оцінки складності коду;

– повернення результату проходження тестів.

Під час проведення функціонального тестування було виявлено, що система не коректно повертає номер рядка при помилці компіляції. Це було спричинене тим, що написаний код обертався в шаблонний код, і тому починався не з першого рядка і мав зміщення по нумерації.

Метод білої скриньки – метод тестування, спрямований на локалізацію помилок, які складніше виявити, знайти і зафіксувати. З його допомогою можна виявити логічні помилки і перевірити ступінь покриття програмного коду тестами.

В таблиці 5.1 наведені тесткейси, що були створені та пройдені на етапі тестування.

Таблиця 5.1 – Перелік тестових кейсів

№	Тест кейс	Опис сценарію	Результат що очікується
1	Відправлення на оцінку коду	Під’єднатися до сесії.	Відкриється опис задачі до розв’язання.
		Написати рішення задачі описаної в завданні, натиснути «Submit»	Алгоритм буде перевірений та оцінений та результати будуть повернені на сторінку відповіді.
2	Відправлення на оцінку коду, що не компілюється	Під’єднатися до сесії.	Відкриється опис задачі до розв’язання.
		Написати будь який код, що має синтаксичні помилки.	Сервіс компіляції перевірить код та поверне помилку про помилку в коді та рядок в якому була допущена помилка.
3	Відправлення на оцінку коду з конструкціями, що не підтримуються лексичним аналізатором	Під’єднатися до сесії.	Відкриється опис задачі до розв’язання.
		Написати рішення задачі з оператором умовного переходу	Результат оцінки буде містити уточнення, що програма аналізувалася часовим алгоритмом.

ВИСНОВКИ

Під час атестаційної магістерської роботи було досліджено способи оцінювання складності алгоритму, проаналізовано актуальні проблеми пов’язані з оцінюванням коду в сучасному світі та досліджено способи реалізації лексичного алгоритму, реалізованого на основі лексичного аналізу, а також часового алгоритму, який покриває ті задачі, які неможливо оцінити лексичним аналізом.

Було проаналізовано етапи створення алгоритмів, причини, через які неможливо створити універсальний лексичний алгоритм для довільної програми, проаналізовано набори задач, які задовольняють умовам оцінювання лексичним алгоритмам. Функцію лексичного аналізатора було удосконалено та перероблено під задачі розроблюваного алгоритму, створені відповідні лексеми, способи їх обробки та підрахунку кількості операцій.

В ході роботи було знайдено які конструкції є обмеженнями для оцінювання лексичним алгоритмом, які є його недоліки та переваги.

Було проаналізовано способи реалізації часового алгоритму, підібрані набори даних, що підходять для тестування, проаналізовано способи конвертації співвідношення часу виконання до розмірів вхідних даних до формули складності.

У ході роботи було проаналізовано типові складності алгоритмів для використання їх в результаті оцінки.

Було проведено аналіз вхідних даних, на яких треба тестувати складність часовим алгоритмом. В ході дослідження було прийнято використовувати подібні дані – тобто або всі вхідні дані в рамках однієї оцінки повинні бути відсортовані, або усі довільні, що збільшить точність оцінювання.

Як підсумок даної роботи, було запропоновано два алгоритми оцінювання складності коду – лексичний та часовий, що використовуються як основний та запасний алгоритми для оцінювання коду.

Основний алгоритм має досить швидку швидкість аналізу, враховуючи що він не потребує запуску програми на виконання і обмежується лише синтаксичним аналізом. У свою чергу, часовий алгоритм працює набагато повільніше, проте може використовуватись для оцінки довільного алгоритму та має досить високу точність.

Функціонал лексичного алгоритму містить обмежений набір конструкцій, що можуть бути розпізнані системою. Для удосконалення алгоритму потрібно доповняти його більш складними конструкціями, підтримкою різних операторів циклу та посилань на зовнішні методи.

ПЕРЕЛІК ПОСИЛАНЬ

1. Електронна версія «Великої української енциклопедії» [Електронний ресурс] vue.gov.ua : Часова_складність_алгоритму. – Режим доступу: www/URL: http://stackoverflow.com (дата звернення: 25.11.2019).
2. Martin D. D., Elaine J. W. Computability, Complexity, and Languages. Elsevier Inc, 2, 1994 – 250 p.
3. Downey, A. Think Complexity. Green Tea Press, 1, 2012 – 113 p
4. Wilf, H. Algorithms and complexity. University of Pennsylvania, 1986 – 187 p.
5. Leeuwen, J. Handbook of Theoretical Computer Science, Vol. A. Elsevier Science Inc. New York, 1, 1990 – 212 p.
6. Schmuki, R., Wagner, F., Wagner, T. Modeling Software with Finite State Machines: A Practical Approach. CRC Press, 1, 2006 – 47 p.
7. Ranum, L. D., and Miller N. B. Problem Solving with Algorithms and Data Structures Using Python. Franklin, Beedle & Associates, 2, 2005 – 317 c.
8. Appel, A., and Palsberg, J. Modern Compiler Implementation in Java. Cambridge University Press, 1, 1997 – 180 c.
9. Knuth, E. D. The Art of Computer Programming. Addison-Wesley, 1,1968 – 329 p.
10. Friedl, J. Mastering Regular Expressions. O'Reilly Media, 3, 1997 – 315 p.
11. Система електронних статей Stack Abuse [Електронний ресурс] stackabuse.com : Theory of Computation: Finite State Machines – Режим доступу: www/URL: http://stackabuse.com (дата звернення: 28.11.2019).
12. Вечур О. В., Лещинська О.Л. Олімпіадні задачі з програмування за матеріалами Зимової школи. Навчальний посібник дисципліни «Алгоритми та структури даних». Україна, Харків, Компанія СМІТ, 2011 – 238 с.

13. Білуха М. Т. Методологія наукових досліджень [Текст] / М. Т. Білуха // Підручник. – К. : АБУ, 2002. – 480 с
14. Вігерс К. Розробка вимог до програмного забезпечення [Текст] / К. Вігерс // Руська редакция, 2004. – 575 с.
15. Ларман К. Застосування UML 2.0 і шаблонів проектування. Введення в об’єктно-орієнтований аналіз, проектування і ітеративну розробку [Текст] / К. Ларман // Вільямс, 2013. – 736 с.
16. Мартін Р., Ньюкирк В., Косс С. Швидка розробка програм. Принципи, приклади, практика [Текст] / Вільямс, 2004. – 752 с.