ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

ЗВІТ

З ПЕРЕДДИПЛОМНОЇ ПРАКТИКИ БАКАЛАВРІВ

Місце проходження практики "НВПП Велєс-М"

у період з "2" вересня по "2" листопада 2019 р.

Тема індивідуального завдання:

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

|  |  |
| --- | --- |
| ст. гр. ПЗСм-18-1 Різник О. К. | Керівник практики Дудар З. В.  Робота захищена з оцінкою \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Дата «\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_2019 |

|  |  |
| --- | --- |
| Керівник атестаційної роботи | \_\_Мазурова О. О. \_\_\_ |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Харків, 2019

**РЕФЕРАТ/ABSTRACT**

Пояснювальна записка до звіту містить 9 рисунків, 2 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.

Результатом роботи є веб-система для вирішення алгоритмічних задач, яка оцінює складність написаного алгоритму і дає точну оцінку складності.

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.

The resulting work is an web-application for solving algorithmics tasks which does complexity analysis of written algorithm and gives an accurate complexity estimation.

**ЗМІСТ**

[Вступ 5](#_Toc22804464)

[1 Стуктура підприємства 7](#_Toc22804465)

[2 Аналіз проблемної області на постановка задачі 8](#_Toc22804466)

[2.1 Аналіз проблемної області оцінювання складності алгоритмів 8](#_Toc22804467)

[3 Перелік вимог до програмної системи 17](#_Toc22804468)

[3.1 Призначення розробки 17](#_Toc22804469)

[3.2 Вимоги до програмного продукту 17](#_Toc22804470)

[3.3 Вимоги до клієнта 18](#_Toc22804471)

[4 Опис прийнятих проектних рішень 20](#_Toc22804472)

[4.1 Дослідження лексикографічних алгоритмів з метою оцінювання складності алгоритмів 20](#_Toc22804473)

[4.2 Дослідження роботи кінечного автомату станів для реалізації лексичних правил для парсингу коду. 21](#_Toc22804474)

[4.3 Аналіз обмежень, що накладаються на код для оцінки його лексичним алгоритмом 24](#_Toc22804475)

[4.4 Дослідження та аналіз способів реалізації часового алгоритму 25](#_Toc22804476)

[4.5 Математичне моделювання 29](#_Toc22804477)

**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 задовільняти 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 неоптимальність 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. І хоча цей спосіб оцінювання складності алгоритму досить точний, він має декілька недостатків:

* по-перше це великі ресурсні затрати через необхідність викликати вхідну програму велику кількість разів з великою кількістю вхідних даних;
* по-перше це великі ресурсні затрати через необхідність викликати вхідну програму велику кількість разів з великою кількістю вхідних даних;
* по-друге при палельному оцінювання інших алгоритмів (якщо це онлайн система для вирішення алгоритмів, то паралельно велика кількість юзерів 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 практики 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.

# **1 СТРУКТУРА ПІДПРИЄМСТВА**

Підприємство "НВПП Велєс-М" надає високоякісні ІТ послуги для клієнтів у таких сферах як електронний бізнес, документообіг, високонавантажені веб-системи, мобільні додатки. Наразі Харківська філія налічує більше 20 працівників. Компанія «НВПП Велєс-М» знаходиться на ринку з 2009 року, почавши свою роботу з Net, PHP та веб-дизайну. Зараз компанія є аутсорсинговим центром, що спеціалізується на розробці .Net, PHP з філією в Харкові (Україна).

Компанія НВПП Велєс-М складається з команди досвідчених консультантів і розробників, креативних дизайнерів і фахівців в області маркетингу, які знають, як досягти найкращих результатів. Працівники повною мірою володіють популярними технологіями, але, в той же час, відкриті для нових технік і постійно прагнуть до вдосконалення. Високий професіоналізм дозволяє їм створювати додатки, які перевершують всі очікування.

Більшість проектів розробляються із використанням гнучких методологій. Гнучка розробка програмного забезпечення — клас методологій розробки програмного забезпечення, що базується на ітеративній розробці, в якій вимоги та розв'язки еволюціонують через співпрацю між самоорганізовуваними багатофункціональними командами. Більшість гнучких методологій націлені на мінімізацію ризиків, шляхом зведення розробки до серії коротких циклів, що мають назву ітерацій, які зазвичай тривають один-два тижні. Гнучка розробка - найкращий засіб для підвищення продуктивності розробників програмного забезпечення.

Різнопланове сервіс-портфоліо і, відповідно, різноманітність проектів, дозволяє не тільки стабільно зростати, але й надавати великі можливості для розвитку співробітникам.

# **АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ**

## 2.1 Аналіз проблемної області оцінювання складності алгоритмів

На даний час усі відомі веб-сервіси, що створені для тренування роз’вязання алгоритмічних задач не мають загальноприйнятого спосібу 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 прямопропорційно 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 досягення 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 [1]. Ця проблема полягає в тому, що не може існувати загального алгоритму для розв’язання проблеми зупинки для всіх пар програма-вхідні дані. Неформально за 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 алгоритм 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 виконуючихся операцій. В залежності від співвідношення розміру вхідних даних до кількості виконаних операцій програмі буде присвоєна її складність (див. рис. 2.1):

Як видно з графіку – деякі функції мають дуже схожий графік росту на невеликах наборах даних, наприклад – [2] та [3] 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 ресурсоекономніше та швидше ніж у загальновикористованому алгоритмі.



Рисунок 2.1 – Графік найпоширеніших функцій в аналізі алгоритмів, де N – кількість операцій, n – розмір вхідних даних для кожної з функцій.

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 кінечних 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 виконани інструкції під умовним оператором. Цей висновок виходить з проблеми зупинки[4], 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 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 невідсортованих – час буде різний, тому при збільшенні даних потрібно зберігати відсортованість даних, порядок чисел, тощо.

* 1. 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, який лише умовно порівнює 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 бистро відпрацював (див. рис.2.2).

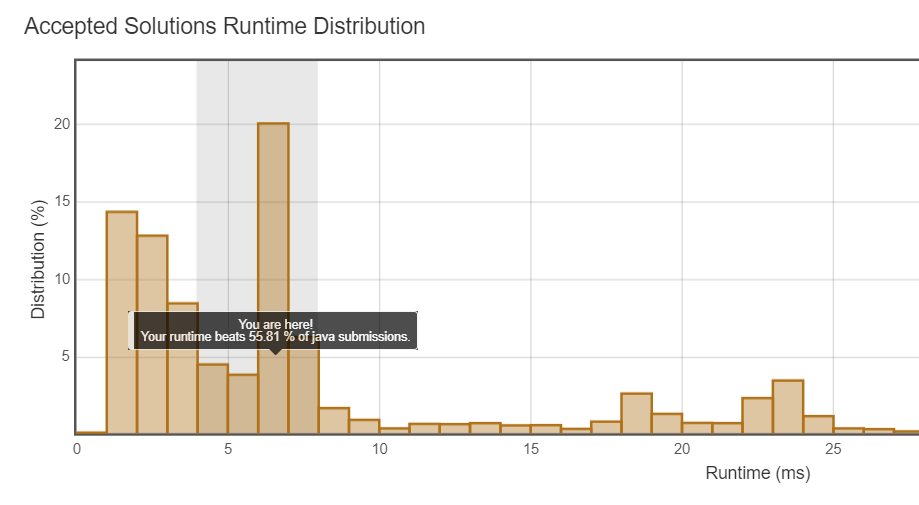


Рисунок 1.2 – Приклад оцінювання алгоритму на сайті leetcode.com.

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, 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 вложеностей циклів, на основі чого робить висновок про складність коду (див. рис.2.3).

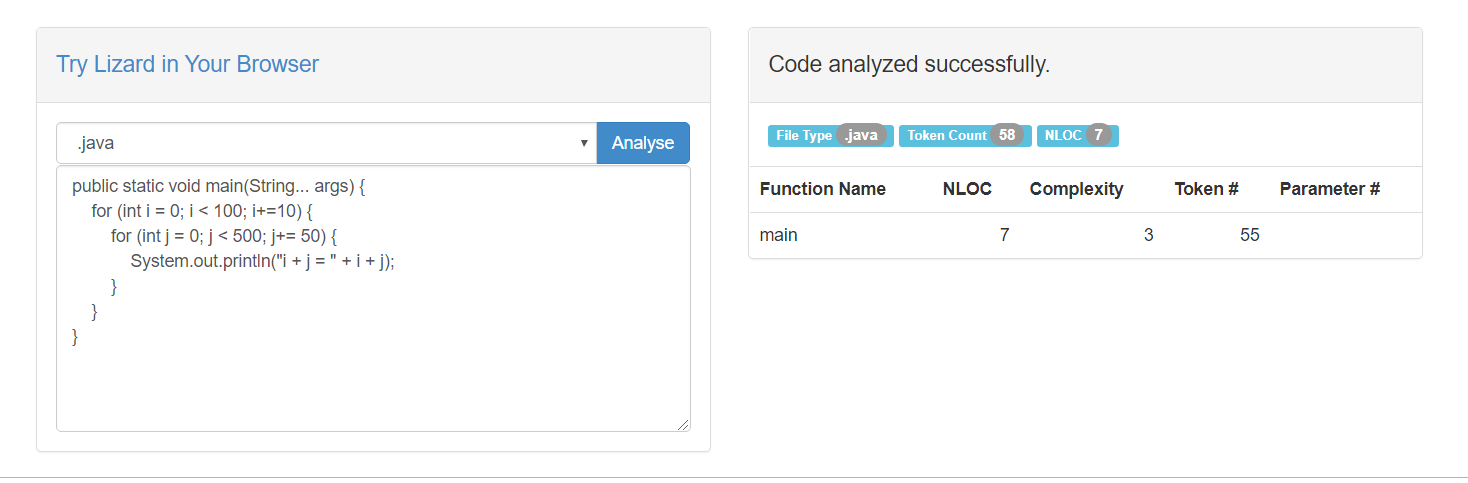


Рисунок 2.3 – Скріншот роботи сервісу lizard.ws.

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 частину від кінцевого значення індексу. Це означає, що справжня складність таких вкладених циклів буде n, при умові що n = 100.

До неділіків 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.

* 1. Постановка задачі

Метою роботи є дослідження алгоритмів оцінки складності алгоритмів, та на основі аналізу розробити алгоритм оцінювання складності коду в рамках веб-системи, яка б надавала задачі для вирішення. Ця система може використовуватись для змагань програмістів між собою. Ця система буде корисним інструментом для розвивання 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.

Кінцеву задачу можна розбити на наступні етапи:

* 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 замірень 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, фреймворну 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.

# **ПЕРЕЛІК ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ**

## 3.1 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: пришвидшення швидкості виконання запиту навіть на 10% може дуже сильно зменшити затрати на систему. У сучасному айті дуже багато людей, 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.

## 3.2 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 функціональноість 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;
* оцінювання коду часовим алгоритмом у разі якщо непідтримувані структури були знайдені;
* парсинг значень змінних;
* оцінювання коду лексичним алгоритмом;
* оцінювання 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.

## 3.3 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 персональногих комп’ютерів так і з мобільних пристроїв, проте написання коду з мобільного телефону не є зручним і не рекомендується.

# **ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ**

## Дослідження лексикографічних 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[8], 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[9]).

Регулярне визначення може виражатися регулярним виразом в даній мові програмування:

*,*

де «a-z» – діапазон, що означає усі симвлди латинського алфовіту нижнього регістру, «A-Z» – верхнього регістру.

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

,

де, letter – означає будь який символ нижнього та верхнього регістру латинського алфавіту, digit – цифри від 0 до 9, а 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 бескінечності.

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

## 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 – кінечні автомати станів (FSMs – Finite State Machines[10]). Результуючий КАС має кінечний 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» (а чи b) може бути представлений у вигляді наступного КАСу (див. рис. 4.1).



Рисунок 4.1 – Кінечний автомат станів для регулярного виразу «a | b»

Вищепредставлений 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:

,

де визначення letter та дігіт були описані вище, number – описує будь-яке ціле число, operator – описує найбільш поширені математичні оператори, parenthesis – дужки.

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

При пошуці 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. 4.2).

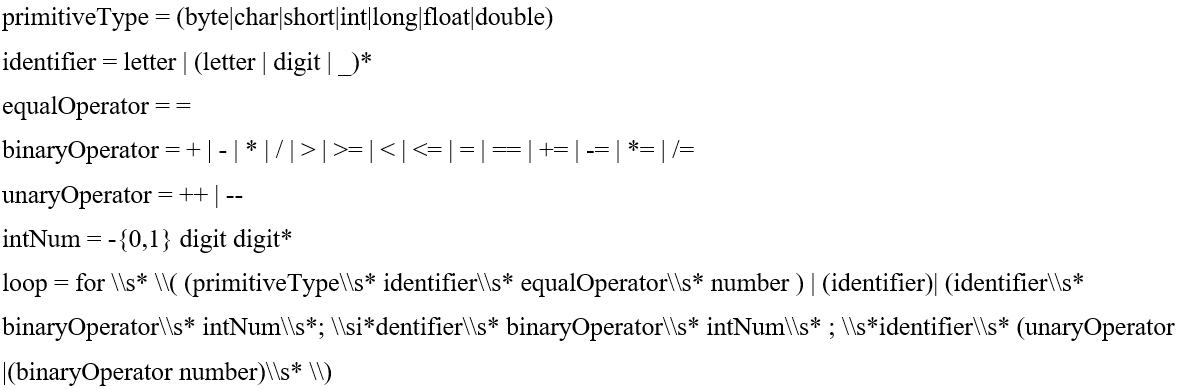


Рисунок 4.2 – Регулярні визначення, що описують правила для циклу for.

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.

## 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 аналізуємому коді.

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:

* Будь-які оператори умовного переходу (if, else, тернарні оператори, тощо);
* виклики функцій з середини оцінюємої функції – неможливо передбачити, чи не буде ця функція циклічною;
* ярликові вирази (label statements[12], такі як break label та continue label;
* вічні цикли (неможливо точно розпізнати чи буде даний цикл вічним, оскільки умова циклу може залежити від вхідних даних які неможливо передбачити), підтримуються лише найбільш очевидні способи задання вічних циклів (див. рис 4.3);

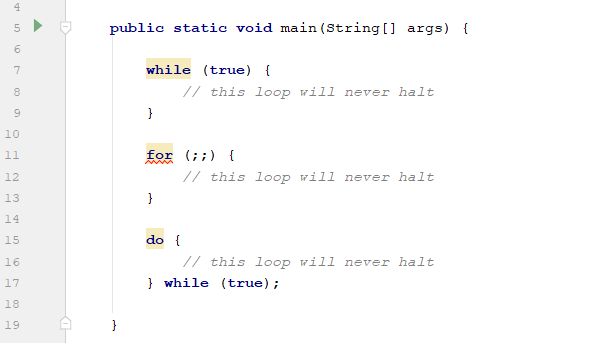


Рисунок 4.3 – Приклади вічних циклів

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.

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

З вищеописаного прикладу можно 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 4.1 – 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 A час виконання (в наносекундах) | Комп’ютер B 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 |
| 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 необіхдно 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 скланість алгоритму часовим способом – необхідно проаналізувати типові часові складності (див. табл. 4.2).

Таблиця 4.2 – Таблиця найбільш поширених складностей алгоритмів

| Назва | [Обчислювальна складність](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D1%8E%D0%B2%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0_%D1%81%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D1%96%D1%81%D1%82%D1%8C) | Час виконання (*T*(*n*)) | Приклади часу виконання | Приклади алгоритмів |
| --- | --- | --- | --- | --- |
| сталий час |  | *O*(1) | 10 | Визначенно парності числа (у двійковому запису) |
| логарифмічний час | [DLOGTIME](https://uk.wikipedia.org/w/index.php?title=DLOGTIME&action=edit&redlink=1) | *O*(log *n*) | log *n*, log(*n*2) | [Двійковий пошук](https://uk.wikipedia.org/wiki/%D0%94%D0%B2%D1%96%D0%B9%D0%BA%D0%BE%D0%B2%D0%B8%D0%B9_%D0%BF%D0%BE%D1%88%D1%83%D0%BA) |
| лінійний час |  | *O*(*n*) | *n* | Пошук найбільшого або найменшого елементу невпорядкованого [масиву](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D0%B8%D0%B2_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)) |
| квадратичний час |  | *O*(*n*2) | *n*2 | [Сортування бульбашкою](https://uk.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B1%D1%83%D0%BB%D1%8C%D0%B1%D0%B0%D1%88%D0%BA%D0%BE%D1%8E); [Сортування включенням](https://uk.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B2%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%BD%D1%8F%D0%BC) |
| кубічний час |  | *O*(*n*3) | *n*3 | Безпосереднє множення двох матриць розміру *n*×*n*. Обчислення [часткової кореляції](https://uk.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%BA%D0%BE%D0%B2%D0%B0_%D0%BA%D0%BE%D1%80%D0%B5%D0%BB%D1%8F%D1%86%D1%96%D1%8F). |
| експоненціальний час | [EXPTIME](https://uk.wikipedia.org/wiki/EXPTIME) | 2poly(*n*) | 2*n*, 2*n*2 | Вирішення [задачі перемноження матриць](https://uk.wikipedia.org/w/index.php?title=%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D1%8C&action=edit&redlink=1)[[en]](https://en.wikipedia.org/wiki/Matrix_chain_multiplication) [повним перебором](https://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%C2%AB%D0%B3%D1%80%D1%83%D0%B1%D0%BE%D1%97_%D1%81%D0%B8%D0%BB%D0%B8%C2%BB) |
| факторіальний час |  | *O*(*n*!) | *n*! | Вирішення [задачі комівояжера](https://uk.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BA%D0%BE%D0%BC%D1%96%D0%B2%D0%BE%D1%8F%D0%B6%D0%B5%D1%80%D0%B0) [повним перебором](https://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%C2%AB%D0%B3%D1%80%D1%83%D0%B1%D0%BE%D1%97_%D1%81%D0%B8%D0%BB%D0%B8%C2%BB) |

Як видно з таблиці – будь-який алгоритм можна привести до типової складності наведеної в таблиці. Наприклад, якщо алгоритм має складність 2*n*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.

## Математичне моделювання

Для правильної розробки системи, яка зможе оцінювати асимптичну часову складність алгоритму я провів аналіз предметної області і з’ясував на які етапи треба розбити процес проектування та розробки відповідної системи:

1. аналіз етапів розробки лексичного алгоритму:
   1. аналіз всіх можливих ключових (зарезервованих) слів у мові;
   2. аналіз можливих операцій;
   3. аналіз можливих операндів;
   4. аналіз способів і методів підрахунку операцій:
      1. пошук у строці коду операцій ініціалізації змінних (операторів присвоєння);
      2. пошук у строці коду операторів (додавання, віднімання, множення, ділення, бінарні, атомарні, тощо);
      3. аналіз обробки операторів циклів (for, while);
      4. аналіз обробки вкладених циклічних операторів (наприклад: якщо всередині циклу є цикл з кількістю ітерацій n, то результатов повинна бути кількість операцій n\*m, де m – кількість операцій зовнішнього циклу);
   5. аналіз способів парсингу коду;
   6. аналіз мапінгу кількості операцій до формули складності;
2. аналіз способів реалізації часового алгоритму:
   1. аналіз методів засікання часу таким чином, щоб в рамках аналізу однієї програми було задіяно фіксовану кількість ресурсів (для забезпечення точності вичислення);
   2. підготовка наборів даних різних розмірів подібного стану (наприклад: якщо на вхід алгоритму quick sort подати набір даних повністю відсортований зі зручним опорним елементом алгоритм може відпрацювати швидше ніж алгоритм з меншим розміром вхідних даних але з невідсортованими даними та незручним опорним елементом. Тому важливо, щоб дані були подібні);
   3. аналіз мапінгу росту часу до формули складності;
3. аналіз набору задач для тестування обох алгоритмів.

Математикой и не пахнет!!!

И зачеп Вы здесь это пишете, ведб по сути Вы некоторые этапы уже выше привели болем подробно или нет? Не поняла смисл???

1. **ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ**
   1. Розробка бази даних для …..????
   2. Реалізація часового алгоритму оцінки задач
   3. Вапварар

6 АНАЛІЗ РЕЗУЛЬТАТІВ ТЕСТУВАННЯ ТА МОЖЛИВОГО ЗАСТОСУВАННЯ (3-4 страниці) – это в практику можно не писать

Источники: ????

9 - <https://hackernoon.com/lexical-analysis-861b8bfe4cb0>

13 - <https://en.wikipedia.org/wiki/Analysis_of_algorithms>