МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет компьютерных технологий и прикладной математики
Кафедра информационных технологий

КУРСОВАЯ РАБОТА

ПРОГРАММА ОЦЕНКИ СЛОЖНОСТИ АЛГОРИТМОВ

Работу выполнил _____ Ф. Р. Петренко
(подпись)
Направление подготовки 02.03.03 — «Математическое обеспечение и
(код, наименование)
администрирование информационных систем»

Направленность (профиль) Технология программирования

Научный руководитель
д-р физ.-мат. наук, проф. _____ А. И. Миков
(подпись)
Нормоконтролер
ст. преп. _____ А. В. Харченко
(подпись)

Краснодар
2021

РЕФЕРАТ

Курсовая работа 88 стр., 30 рис., 7 источников, 6 приложений.

ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМА, ПОСТРОЕНИЕ ГРАФИКА СЛОЖНОСТИ, ТРАНСЛЯТОР, ЛИНЕЙНАЯ РЕГРЕССИЯ

Объектом исследования в данной работе является процесс оценки сложности алгоритмов с помощью программных средств.

Цель работы: разработка программы оценки сложности алгоритмов, включающая язык программного описания рассматриваемых алгоритмов.

Методологическая основа исследования включает в себя элементы математической статистики, метод гипотез, графический метод, анализ полученных зависимостей, классификацию.

В результате работы было разработано приложение, которое организует трансляцию кода алгоритма из разработанного языка Algolite в код на языке C++, включающее дальнейший анализ сложности алгоритма методом линейной регрессии и построение графика сложности.

Научная новизна и уникальность работы заключается в том, что предлагается оригинальный способ программного анализа сложности различных алгоритмов на основе числа элементарных операций, которые они содержат.

В результате решения поставленных задач был определен метод программного анализа сложности алгоритмов с использованием линейной регрессии, организовано построение графика, разработан язык Algolite для описания алгоритмов и транслятор программного кода с данного языка в язык C++. С учётом ограничений числа рассматриваемых классов сложности до $O(n^5)$, проведённые экспериментальные исследования на основе некоторого количества алгоритмов показали высокую точность определения их сложности. Разработанный алгоритм определения класса сложности в большинстве случаев верно решает поставленную задачу при условии подбора благоприятных параметров для построения линейной регрессии.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Несмотря на сегодняшние тенденции к упрощению процесса разработки программного обеспечения благодаря введению различных вспомогательных средств (библиотеки, фреймворки, средства визуального программирования и т.д.), создание огромного числа программных продуктов всё ещё неразрывно связано с самостоятельной реализацией и внедрением алгоритмов, решающих набор определённых проблем. Отдельные проблемы пришли из классической математики, к примеру, численные методы решения дифференциальных и иных уравнений.

Актуальность данной работы объясняется тем, что каждый разработчик, находясь на этапе внедрения в систему некоторого конкретного алгоритма и задумываясь над резонностью данного действия, тратит время на ручное определение сложности рассматриваемого алгоритма. Подобные временные затраты могут быть легко исключены использованием программы, автоматизирующей данный процесс.

Основной целью работы является разработка программы для автоматизированного определение класса сложности некоторого алгоритма с дальнейшим построением графика сложности.

Для реализации поставленной цели предполагается решить следующие задачи:

– разработать независимый язык для описания произвольного алгоритма, содержащий исчерпывающее число конструкций и механизмов;

– организовать перевод текста алгоритма из данного языка в команды языка C++ путём проектирования программы-транслятора;

– реализовать вставки счётчиков в конечный программный код для подсчёта числа элементарных операций, выполняемых при запусках с различными значениями некоторого параметра;

– внедрение некоторого метода определения класса сложности рассматриваемого алгоритма с высокой точностью.

Объектом исследования в данной работе является процесс оценки сложности алгоритмов с помощью программных средств.

Предметом исследования является качество работы разработанной программы, характеризующееся точностью определения сложности рассматриваемых алгоритмов.

Методологическая основа исследования включает в себя элементы математической статистики, метод гипотез, графический метод, анализ полученных зависимостей, классификацию.

Научная новизна работы заключается в том, что предлагается оригинальный способ программного анализа сложности различных алгоритмов на основе числа элементарных операций, которые они содержат.

Теоретическая и практическая значимость работы характеризуется пользой, которую может привнести программа, позволяющая разработчикам автоматизированным способом определять сложность рассматриваемых алгоритмов при проектировании некоторой системы, тем самым исключая излишние временные затраты на ручное определение сложности.

1 Постановка задачи

Традиционно в программировании понятие сложности алгоритма связано с использованием ресурсов компьютера. Приходится задаваться вопросом: сколько процессорного времени требует программа для своего выполнения, как много памяти машины при этом расходуется? Учет памяти обычно ведется по объему данных. Время рассчитывается в относительных единицах так, чтобы эта оценка, по возможности, была одинаковой для различных машин [1].

Будем рассматривать временную сложность алгоритмов, которая подсчитывается в исполняемых командах: количество арифметических операций, сравнений, пересылок.

Временная сложность алгоритма обычно выражается с использованием нотации «О» большое (Big O), которая учитывает только слагаемое самого высокого порядка (т.е. константы, входящие в функцию сложности, не учитываются) [2]. Если сложность выражена таким способом, говорят об асимптотическом описании временной сложности, то есть при стремлении размера входа к бесконечности. Для лучшего понимания сначала опишем данный процесс с формальной стороны, приведя ряд определений.

Проколотой окрестностью точки называется окрестность точки, из которой исключена эта точка.

Пусть $f(x)$ и $g(x)$ — две функции, определенные в некоторой проколотой окрестности точки x_0, причем в этой окрестности g не обращается в ноль. Говорят, что f является «О» большим от g при $x \to x_0$, если существует такая константа $C > 0$, что для всех x из некоторой окрестности точки x_0 имеет место неравенство:

$$|f(x)| \leq C|g(x)| \tag{1}$$

Иначе говоря, в первом случае отношение модулей $f(x)$ и $g(x)$ ограничено сверху некоторой константой C в окрестностях x_0. Нотация «О» большое описывает количество операций при наихудшем сценарии.

В рамках рассмотрения сложности алгоритмов Big O по своей сути позволяет разделять различные алгоритмы по классам сложности [3]. К таким классам относятся: $O(1)$ (константное время), $O(n)$ (линейное время), $O(n^{\alpha})$ (полиномиальное время, где $\alpha > 1$), $O(\log n)$ (логарифмическое время), $O(n * \log n)$ (линейно-логарифмическое время), $O(2^n)$ (экспоненциальное время) и т.д.

Задача курсовой работы состоит в разработке программы для автоматизированного оценивания сложности произвольного алгоритма с отображением графика его сложности.

После завершения разработки приложения будет проведено некоторое число тестирований, направленных на выявление качества работы программы, проявляющегося в точности определения класса сложности данного алгоритма. Тестирования будут проводиться при разных значениях некоторого параметра, влияющего на число выполняемых операций, на основе набора общеизвестных алгоритмов для лучшего понимания того, насколько близок результат работы программы к настоящему классу сложности того или иного алгоритма.

Результаты исследования должны быть приведены в виде графиков сложности и наименований классов сложности в рамках нотации «О» большое (Big O).

Ввиду существования немалого числа классов сложности на задачу будет наложено следующее ограничение: наиболее точный результат будет определяться только для алгоритмов, принадлежащих классам сложности $O(1)$, $O(\log n)$, $O(n)$, $O(\mathrm{n} * \log n)$, $O(n^2)$, …, $O(n^5)$. Если алгоритм принадлежит иному классу сложности, ответ будет дан либо приближенно к одному из вышеуказанных, либо отмечен как $> O(n^5)$.

Преобладание самостоятельности в работоспособности программы (без вмешательства человека) рассматривается как один из наиболее важных аспектов данной курсовой работы, так как именно такой подход может нести наибольшую пользу для разработчиков, уменьшая затраты их времени на ручное определение сложности интересующих их алгоритмов.

2 Математические методы решения задачи

Для проведения программного анализа сложности произвольного алгоритма он должен быть записан в форме, приемлемой для разрабатываемой программы. Для этого необходимо принять некоторый набор команд и организовать на его основе программный язык для описания алгоритмов.

В целях решения данной задачи был разработан язык описания алгоритмов Algolite, представляющий собой небольшой набор команд из языка C++ с соответствующей семантикой и функционалом. Функционал из C++, используемый в качестве основы разработанного программного языка, был выбран таким образом, чтобы оказаться вполне достаточным для описания произвольного алгоритма. Для описания языков нередко используется форма Бэкуса-Наура (БНФ, BNF), представляющая собой формальную систему описания синтаксиса. Далее следует описание команд из Algolite в виде БНФ с периодическим использованием форм записи из регулярных выражений:

<цифра> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<буква> ::= /[A-Za-z]/

<число> ::= <цифра> [{<цифра>} ['.' <цифра> [{<цифра>}]]]

<логическое значение> ::= 'true' | 'false'

<имя> ::= <буква> [{<буква> | <цифра>}]

<переменная> := <имя> [{'[' <выражение> ']'}]

<тип данных> ::= 'int' | 'bool' | 'long' | 'long long' | 'double'

<сумма> ::= <слагаемое> [{<аддитивный оператор> <слагаемое>}]

<and-выражение> ::= <сумма> | <условие>

<or-выражение> ::= <and-выражение> [{'&&' <and-выражение>}]

<выражение> ::= <or-выражение> [{'||' <or-выражение>}]

<слагаемое> ::= <множитель> [{<мультипликативный оператор> <множитель>}]

<аддитивный оператор> ::= '+'|'-'

<мультипликативный оператор> ::= '*' | '/' | '%'

<множитель> ::= ['-'] <атом>

<атом> ::= <число> | <логическое значение> | <переменная> | <вызов функции> | '(' <выражение> ')'

<аргумент функции> ::= <тип данных> <имя> ['[]'] [{'[' <выражение> ']'}]

<знак> ::= '>=' | '<=' | '>' | '<' | '==' | '!='

<условие> ::= <сумма> <знак> <сумма>

<объявление функции> ::= (<тип данных> | 'void') <имя> '(' [<аргумент функции> [{',' <аргумент функции>}]] ')' (';' | <определение функции>)

<определение функции> ::= '{' <команда> '}'

<вызов функции> ::= <имя> '(' [<выражение> [{ ',' <выражение> }]] ')'

<объявление переменной> ::= <тип данных> <переменная> ['=' <выражение>] [{',' <переменная> ['=' <выражение>]}]

<объявление параметра сложности> ::= 'parameter' <имя> '(' <выражение> ',' <выражение> ',' <выражение> ')' ';'

<присваивание значения переменной> ::= <переменная> '=' <выражение>

<инкремент> ::= (<переменная> '++') | ('++' <переменная>)

<декремент> ::= (<переменная> '--') | ('--' <переменная>)

<break> ::= 'break'

<continue> ::= 'continue'

<return> ::= 'return' <выражение>

<уменьшение значения> ::= <переменная> '-=' <выражение>

<увеличение значения> ::= <переменная> '+=' <выражение>

<if> ::= 'if' '(' <выражение> ')' <команда> ['else' <команда>]

<while условие> ::= 'while' '(' <выражение> ')'

<while> ::= <while условие> <команда>

<do while> ::= 'do' <команда> <while условие>

<for> ::= 'for' '(' [<простая команда>] ';' [<выражение>] ';' [<простая команда>] ')' <команда>

<case> ::= 'case' <выражение> ':' [{<команда>}]

<switch> ::= 'switch' '(' <выражение> ')' '{' [{<case>}] '}'

<печать> ::= 'cout' '<<' (<выражение> | <строковый литерал>) [{'<<' (<выражение> | <строковый литерал>)}]

<блок> ::= '{' [{<команда>}] '}'

<простая команда> ::= <вызов функции> | <объявление переменной> | <присваивание значения переменной> | <печать> | <инкремент> | <декремент> | <уменьшение значения> | <увеличение значения> | <break> | <continue> | <return>

<команда> ::= (<простая команда> ';') | (<do while> ';') | <if> | <for> | <switch> | <while> | <блок>

<программа> ::= <объявление параметра сложности> [{<объявление функции>}]

Код, написанный на языке Algolite, принимаемый программой, будет подвержен процессу трансляции в код на языке C++. Данный процесс необходим ввиду возможного значительного отхождения синтаксиса языка Algolite от C++ в будущем. Процесс трансляции подразумевает, что разработанная программа будет включать в себя функционал транслятора (в нашем случае – синтаксически-ориентированного однопроходного компилятора). Программа, полученная путём трансляции кода из языка Algolite в C++, будет содержать программные блоки, необходимые для проведения анализа сложности и построения графика [4].

При обычном подсчёте сложности некоторого алгоритма разработчикам приходится вручную добавлять счётчики для нахождения числа элементарных операций, выполняемых рассматриваемым алгоритмом при заданном значении параметра. В процессе решения задачи данной курсовой работы необходимо автоматизировать максимально возможное число процессов, в том числе и данный.

Для проведения анализа сложности изначальный текст алгоритма обязан содержать функцию main без параметров, с которой и начнётся выполнение

процесса анализа сложности, и специальную команду, описывающую объявление параметра сложности. Эта команда содержит наименование параметра, на котором проходит запуск алгоритма при анализе, начальное значение параметра при запуске анализа, максимальное значение, которое параметр не может достичь, и шаг, на который увеличивается значение параметра в цикле анализа сложности.

Конечная программа в командах языка C++ будет запускать рассматриваемый алгоритм на различных значениях параметра сложности и получать число выполняемых операций, соответствующих данным значениям.

Реализовать процесс анализа сложности можно при помощи линейной регрессии, представляющей собой метод восстановления зависимости между двумя переменными, где в качестве переменных будут выступать значение параметра сложности и число выполняемых при нём операций [5, 6].

Рассмотрим следующий набор алгоритмов, необходимых для проведения этапов трансляции (опустим этап семантического анализа ввиду требования большего числа времени для комплексного подхода к его реализации) алгоритма и анализа его сложности.

2.1 Алгоритм лексического и синтаксического анализа

В основе произвольного естественного или искусственного языка лежит алфавит, представляющий собой набор допустимых знаков (буквы, цифры и т.д.). Знаки объединяются в слова (или лексемы), а слова - в элементарные конструкции языка, рассматриваемые в тексте программы как связанный набор информации, несущий некий смысл.

В качестве ведущего алгоритма синтаксического анализа будем использовать метод рекурсивного спуска, представляющий собой алгоритм нисходящего анализа, реализуемый путём вызова функций разбора команд, проверяющих эти команды на соответствие установленному набору правил

(например, БНФ). Эти функции принимают на вход объекты-токены, возвращаемые лексическим анализатором [4].

Процесс работы во время лексического и синтаксического анализа имеет вполне устоявшуюся форму. Принцип выполнения данного этапа трансляции можно разделить на 3 шага:

– проверка слов (лексем), входящих в изначальную программу, на принадлежность допустимому алфавиту;

– проверка конструкций из слов, используемых в изначальной программе, на соответствие синтаксическим правилам языка (допустимость указанного следования лексем);

– формирование промежуточной формы (внутреннего представления) конечной программы;

В целях проверки текста на соответствие нормам программного языка входная программа подвергается синтаксическому анализу, который в зависимости от типа транслятора может сразу же включать в себя этап лексической обработки, где проводится лексический контроль, т.е. выявление в программе недопустимых слов. Синтаксический анализатор принимает результат работы лексического анализатора и проверяет допустимость порядка следования проанализированных лексем. В случае соответствия изначальной программы упомянутым правилам транслируемого языка синтаксический анализатор переводит последовательность лексем в форму промежуточной программы (внутреннему представлению), из которой будет организован переход к командам конечного языка.

Формирование внутреннего представления программы начинается, как правило, с распределения и выделения памяти для основных программных объектов, которые будут представлять проанализированный исходный текст в том виде, который будет задействован для создания конечного результата. Производится исследование каждого предложения и генерируется эквивалентные структуры объектного языка. Для промежуточной формы программы хорошо подходит древовидная структура, состоящая из узлов,

которыми и являются те самые объекты внутреннего представления, имеющие ссылки друг на друга.

При наличии ошибок хотя бы на одном из этапов процесс трансляции завершается.

Принцип работы данного этапа трансляции (без упоминания дальнейшего перехода к анализу семантики) представлен на рисунках 1, 2, 3 [4].

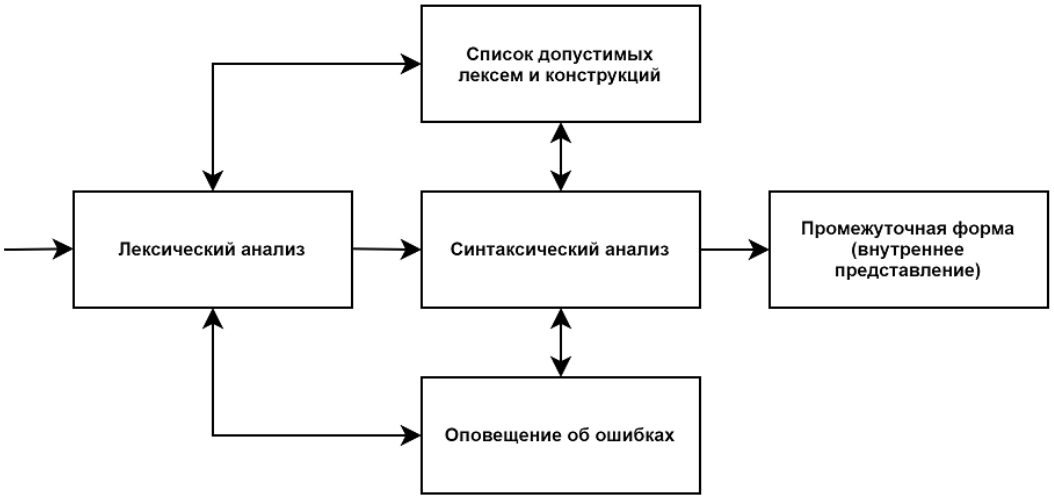


Рисунок 1 – Общая схема работы лексического и семантического анализатора

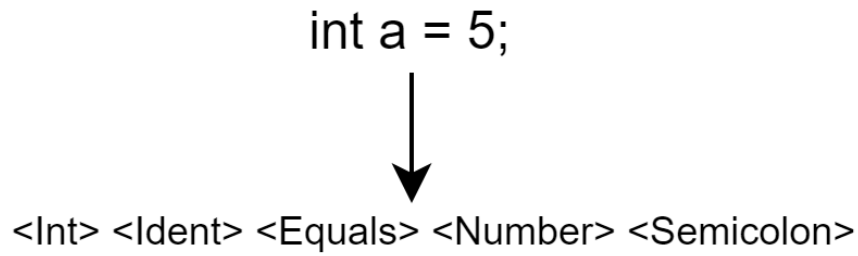


Рисунок 2 – Пример разбора команды лексическим анализатором

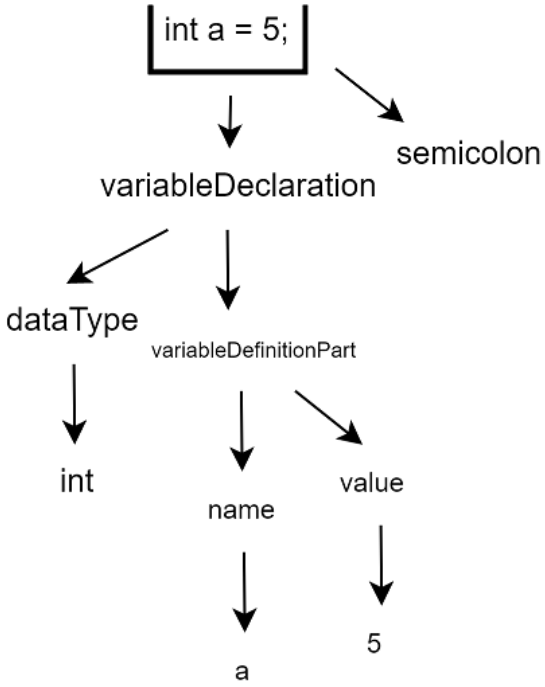


Рисунок 3 – Пример разбора команды синтаксическим анализатором методом рекурсивного спуска

2.2 Алгоритм генерации кода и вставки счётчиков

На этапе генерации кода конечного языка в качестве входной информации используется синтаксическое дерево исходной программы (в случае наличия семантического анализатора также будут использоваться сформированные выходные таблицы лексического анализатора – таблица идентификаторов, таблица констант и другие). Каждый элемент дерева может содержать ссылку на другой элемент, представляющий собой часть общей конструкции, в которые входят оба этих элемента (узла дерева). Анализ дерева позволяет выявить последовательность генерируемых команд объектной программы [4].

Примерную схему построенного дерева отображает рисунок 4:

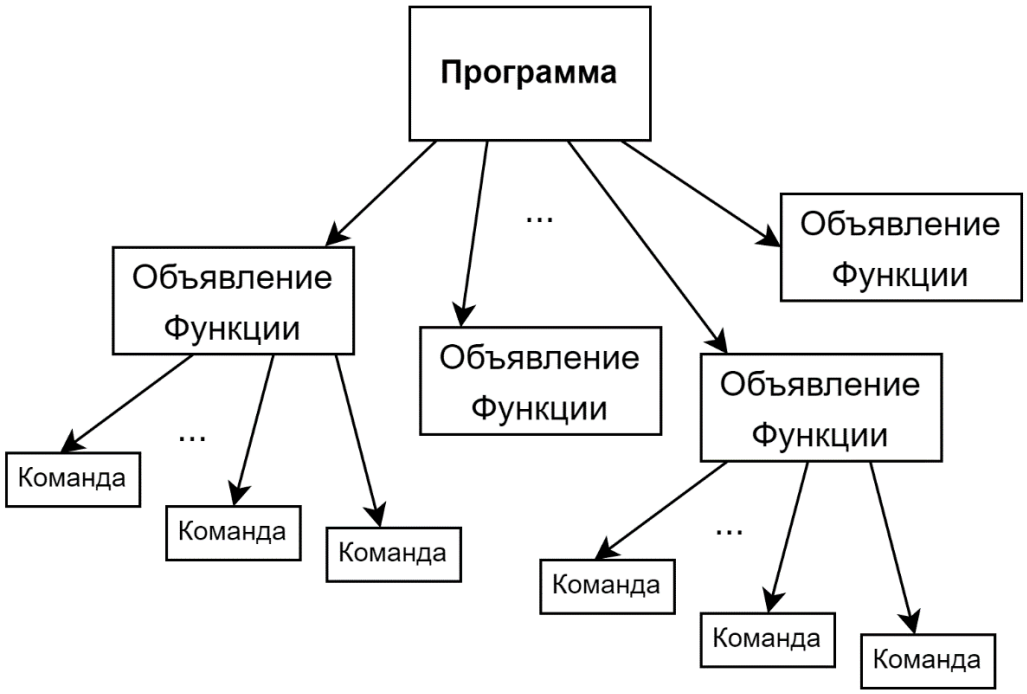


Рисунок 4 – Структура синтаксического дерева

Для проведения анализа синтаксического дерева необходимо организовать его полный проход, для чего хорошо подходит паттерн Посетитель (Visitor) [7]. Посетитель представляет собой поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Многие узлы в конечном синтаксическом дереве будут сильно напоминать друг друга, ввиду чего паттерн Посетитель предлагает вынести поведение при анализе каждого такого узла в отдельный класс. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого они будет передаваться в методы объекта – посетителя, необходимого для полного прохождения синтаксического дерева. Так как узлы будут содержать ссылки друг на друга, то и посетитель сможет провести анализ каждого из них, перемещаясь от одного к другому.

Пример архитектуры с использованием паттерна Visitor в UML (язык графического описания для объектного моделирования) нотации указан на рисунке 5:

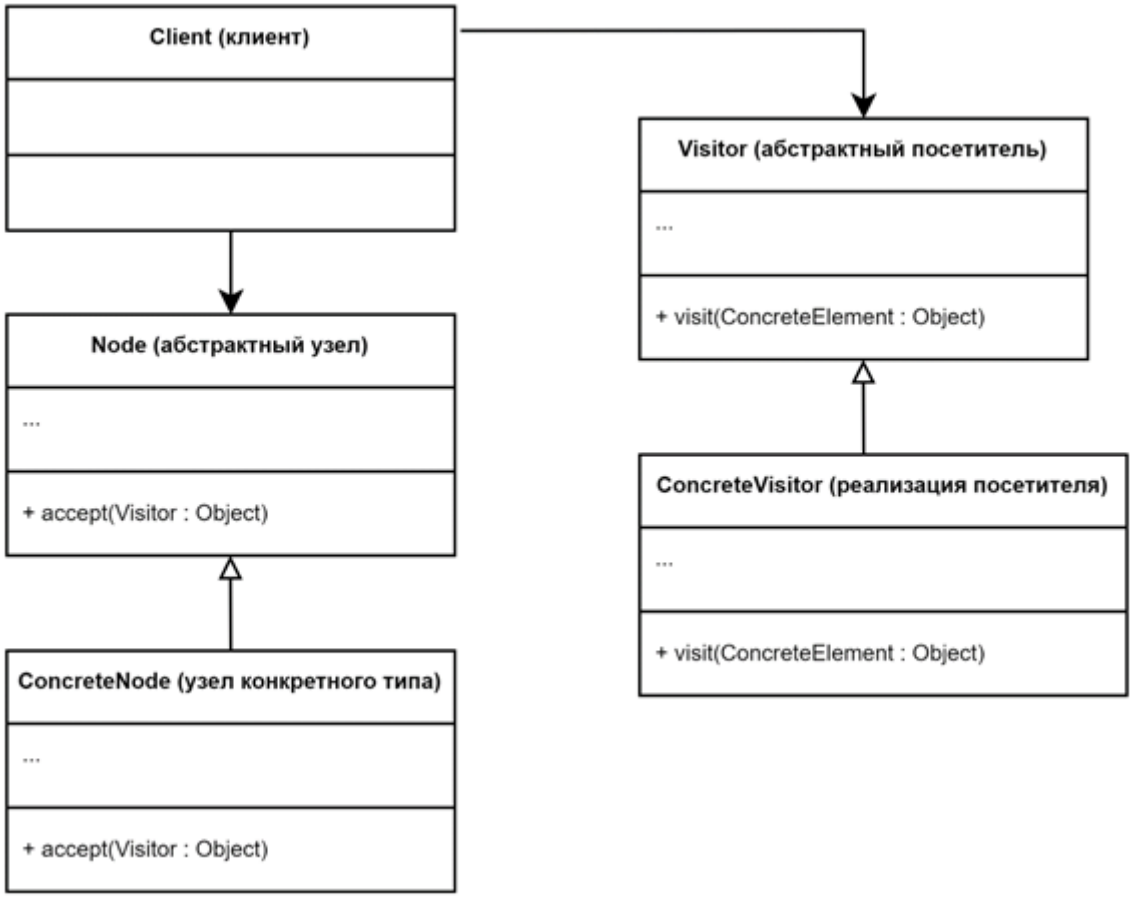


Рисунок 5 – Структура паттерна Visitor в нотации UML

Так как целью курсовой работы является проведение анализа сложности транслируемого алгоритма, то в процесс генерации кода конечной программы можно внедрить расстановку команд для увеличения счётчика, символизирующего число выполняемых операций в зависимости от встреченной конструкции. В случае встреченной конструкции, содержащей некоторое число элементарных операций, увеличивающих счётчик, команда вида counter += N, где N – число встреченных элементарных операций, будет ставиться сразу же после сформированной команды конечного языка через запятую, так как подобный способ выполнения операций поддерживается конечным языком C++ (кроме выражений внутри цикла for на месте этапа инициализации – в данном случае увеличение счётчика будет вынесено за пределы цикла).

2.3 Алгоритм анализа сложности на основе линейной регрессии

Одним из уникальных принятых решений в рамках данной курсовой работы оказалось введение алгоритма анализа сложности алгоритма на основе используемой в статистике линейной регрессии, представляющей собой регрессионную модель зависимости некоторого числа переменных друг от друга [5, 6].

Метод линейной регрессии даёт возможность найти коэффициенты для уравнения прямой, которая наиболее точно соответствует найденным данным. Данными для анализа будут представимы как пары точек, где одной из переменных является значение параметра, на котором происходит запуск вычисления числа проводимых операций, а второй – само число операций. В общем случае мы получим следующие типы классов сложности: степенные ($O(n^{\alpha})$ при $\alpha \geq 0$) и логарифмические ($O(n^{\alpha} * \log n)$ при при $\alpha \geq 0$).

Для анализа сложности в изначальной программе необходимо задать граничные значения, которые будут использоваться для определения числа выполняемых операций на конкретном значении параметра, и шаг, с которым это значение будет изменяться. Для наилучшего результата определения класса сложности следует задавать такие границы, чтобы область параметров переходила хотя бы через 1 порядок размера: мы будем анализировать функцию сложности, предполагая её принадлежность к логарифмическому типу, вследствие чего увеличение параметра в 10 раз приводит к увеличению значения десятичного логарифма на 1. Чтобы замерить коэффициент наклона, надо чтобы хоть насколько то увеличились значения логарифма, что будет обеспечено наличием большой области между граничными значениями параметра.

При достаточно больших x (исходный параметр) и предположении, что функция сложности - степенная, можно пренебречь слагаемыми при младших степенях. Предположим, что $y = c * x^{k}$ (y – число операций при данном x). Тогда $log(y) = log(c) + k * log(x)$. Пусть $u = log(y)$, а $v = log(x)$.

Значит в новых переменных функция сложности обретает вид $u = log(c) + k * v$ (уравнение прямой). Тогда можно провести линейную регрессию и узнать значения $log(c)$ и k, где k - степень x в исходной функции (если функция сложности представляет собой постоянную, то $k = 0$). Для определения степени x в функции сложности используем следующие формулы:

$$b = \frac{n\sum x'y' - (\sum x')(\sum y')}{n\sum (x')^2 - (\sum (x'))^2} \quad (2)$$

$$a = \frac{\sum y' - b(\sum x')}{n} \quad (3)$$

В этих формулах $a = log(c)$, $b = k$, $x' = v$, $y' = u$. Отсюда получим, что $u` = a + b * v = log(c) + k * v$. Теперь посчитаем среднюю ошибку:

$$e = \frac{\Sigma (u - u`)^2}{n} \quad (4)$$

Средняя ошибка (отображающая величину погрешности при попытке приближения прямой, полученной из предположения принадлежности функции к одному из двух типов классов сложности, к найденным парам значений параметра и числа выполнимых элементарных операций) представима как сумма квадратов разностей исходных значений u и предполагаемых значений функции $u`$ при каждом из x и делённая на число точек.

Аналогично рассмотрим случай с логарифмами. Предположим, что $y = c * (x^k) * log(x)$. Разделим всё на $log(x)$. Тогда $log(\frac{y}{log(x)}) = log(c) + k * log(x)$. Пусть $u = log(\frac{y}{log(x)})$, а $v = log(x)$ и т.д. Таким образом получим

точное значение степени x (если это просто логарифмическая сложность, то $k = 0$) и значение ошибки.

Если первая ошибка не превышает второй, то класс сложности относится к степенному типу, иначе – класс функции сложности относится к логарифмическому типу.

Суть перехода к новым переменным при различных предположениях о типе функции сложности алгоритма представлена на рисунках 6, 7:

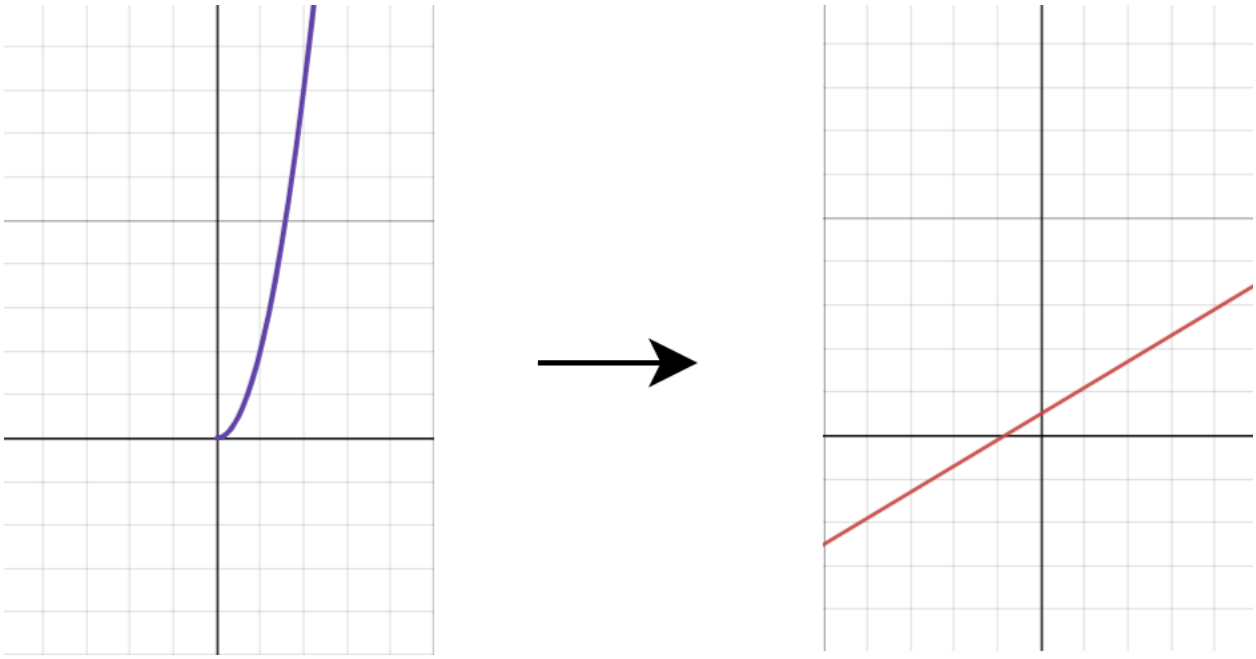


Рисунок 6 – Отображение процесса перехода к новым переменным в случае принадлежности функции сложности к степенному типу

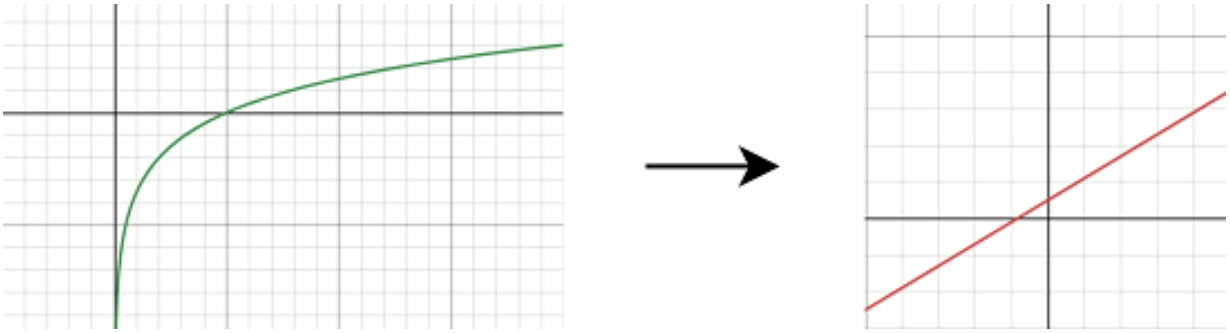


Рисунок 7 – Отображение процесса перехода к новым переменным в случае принадлежности функции сложности к логарифмическому типу

3 Программная реализация

В качестве среды разработки была выбрана Microsoft Visual Studio Community 2019, версия 16.11.5. Программа создавалась на языке программирования C++. Программа состоит из двух .cpp файлов, двух заголовочных файлов, трёх текстовых файлов для автоматической вставки блоков кода анализа сложности и построения графика и одного текстового файла для описания анализируемого алгоритма. Полный листинг программы представлен в прилегающих приложениях.

После успешного проведения трансляции у всех функций, используемых в файле с описанием алгоритма, будут изменены их идентификаторы путём добавления символа “_” перед идентификатором (необходимо для избегания пересечений идентификаторов указанных функций и функций конечной сформированной программы).

Для автоматизированного отображения графика сложности в сформированную программу будет включена часть кода, отвечающая за его построение на основе функционала библиотеки pbPlots. Взаимодействие с ней заканчивается на помещении значений параметра и числа выполняемых элементарных операций в два различных вектора и построении графика на основе значений, которые они содержат. В процессе работы с некоторым количеством дробных чисел во время анализа сложности алгоритма значения на графиках, выводимых пользователю, могут не соответствовать действительности, что обусловлено внутренними особенностями работы библиотеки. Библиотека имеет внутренний механизм, подстраивающий график под указанные размеры, тем самым создавая на некоторых осях дробные значения даже в тех случаях, если в наборе точек, используемых для построения графика, таких значений нет. Осям графика были даны наименования, отображающие значение параметра сложности (complexity parameter) и число выполняемых элементарных операций (operations).

Так как среда разработки Microsoft Visual Studio Community 2019 не предоставляет возможности автоматически провести компиляцию сгенерированного файла для анализа сложности и построения графика транслированного алгоритма прямо из ранее запущенного проекта, было принято решение подключить отдельный компилятор g++. Процесс работы данного компилятора будет запущен после успешного формирования конечного файла в командах языка C++. Сама компиляция будет запущена командой system("g++ -O3 -o output_program.exe final_output.cpp pbPlots.cpp supportLib.cpp -lm"). Указанные при вызове компилятора параметры описываются следующим образом: g++ – вызов компилятора, -O3 – включение оптимизации компилятора 3-го уровня, -o output_program.exe – определение будущего исполняемого файла, final_output.cpp pbPlots.cpp supportLib.cpp – выбор сформированного файла и файлов библиотеки pbPlots для компиляции, -lm – подключение математической библиотеки. Запуск исполняемого файла осуществляется командой system("start output_program.exe").

3.1 Описание структур и функций

Программа разбивается на несколько файлов. Будем рассматривать каждый из файлов поочередно, описывая функции, методы и структуры, содержащиеся в нём. Ввиду схожести поведения некоторых блоков, предназначенных на прохождение дерева, будем конкретизировать наиболее важные из них. Узлы в дереве указывают друг на друга с помощью “умных” указателей unique_ptr (данный вид указателей был выбран ввиду наличия у них строгого запрета на копирование, что сильно ограничивает и упрощает процесс работы с памятью).

1) AlgoliteParser.cpp – основной файл, содержащий функционал лексического анализа, синтаксического анализа и генерации синтаксического дерева;

а) readSource – функция для чтения текста из файла (возвращает текст в виде строки);

б) TokenType – тип перечисления, необходимого для определения принадлежности лексемы допустимому алфавиту;

в) error – функция вывода ошибки;

г) savePrevSymbolData, backupSymbolData – функции для сохранения и отката к предыдущей позиции в анализируемом тексте (используется для отката к предыдущей важной позиции в процессе семантического анализа);

д) nextSymbol – функция для перехода к следующей лексеме (для встреченных уникальных чисел и идентификаторов сохраняет их значения в специальные переменные для дальнейшего взаимодействия);

е) initText – функция для инициализации старта анализа текста;

ж) accept – функция, принимающая токен перечисления (тип ожидаемой лексемы), для сравнения с токеном лексемы, рассматриваемой в данный момент (необходим для отслеживания правильной постановки лексем с точки зрения синтаксических норм языка);

и) skipSpaces – функция для пропуска символов, не имеющих смысловую нагрузку для анализа текста (пробелы, переносы строк и т.д.);

к) digit, letter – функция для проверки символа рассматриваемого текста на соответствие цифре/букве;

л) readNumber, id – функции для проверки лексемы на соответствие числу/идентификатору;

м) acceptNumber – функция для проверки последовательности лексем на соответствие числу со знаком/без знака минус;

н) dataType – функция для проверки лексемы или последовательности лексем на соответствие конструкции типа данных: int, bool, double long, long long (возвращает строковое значение);

п) factor, term, sum, andExpression, orExpression, expression, variable, atom, coutPrinting, variableDeclaration, prefixIncrement, prefixDecrement, simpleCommand, ifCommand, whileCondition, whileCommand, doWhileCommand, forCommand, forCommand, caseCommand, switchCommand, functionCall, command, functionArguments, functionDefinition, functionDeclaration – функции для проверки последовательности лексем на соответствие одной из возможных конструкций анализируемого языка (в процессе синтаксического анализа помимо проверки также формируется набор данных для создания узла ожидаемой конструкции и добавления такого узла в дерево программы);

р) program – функция для формирования дерева, содержащего узлы проанализированных конструкций в классе Program;

2) Nodes.h – заголовочный файл, содержащий описания классов узлов, используемых для заполнения дерева;

а) Node – абстрактный класс, представляющий произвольный узел синтаксического дерева (каждый класс наследник содержит абстрактный или реализованный метод visit, предназначенный для анализа объектом некоторого класса, наследованного от абстрактного класса Visitor;

б) ExpressionNode, StatementNode – абстрактные классы-наследники класса Node (предназначены для разделения узлов конструкций на типы Expression и Statement);

в) FunctionDefinitionNode, ParenExpressionNode, BinaryOperationNode, UnaryOperationNode, IntegerLiteralNode, DoubleLiteralNode, BooleanLiteralNode, StringLiteralNode, VariableNode, IfNode, WhileNode, ForNode, CaseNode, SwitchNode, BlockNode, FunctionCall, FunctionCallAsStatement, CoutNode, IncDecNode, BreakNode, ContinueNode, ReturnNode, VariableDeclarationNode, AssignmentNode, Program – классы узлов, образующих синтаксическое

дерево (каждый такой класс обладает индивидуальным набором полей, что обусловлено разным подходом к каждому из них при генерации конечной программы);

г) VariableDefinitionPart – вспомогательная структура, представляющая собой часть узла VariableDeclarationNode;

д) Visitor – абстрактный класс, предназначенный для прохождения сформированного синтаксического дерева (объект такого класса не будет получать доступ к полям узлом извне – они сами будут предоставлять ему доступ, вызывая метод visit);

3) TranslatingVisitor.h – заголовочный файл, содержащий объявление методов класса TranslatingVisitor, являющегося наследников абстрактного класса Visitor;

4) TranslatingVisitor.cpp – файл, содержащий описание работы объекта класса TranslatingVisitor при обходе синтаксического дерева с последующей генерацией кода конечной программы;

а) handle(ArgumentNode& n), handle(FunctionDefinitionNode& n), handle(ParenExpressionNode& n), handle(BinaryOperationNode& n), handle(UnaryOperationNode& n), handle(IntegerLiteralNode& n), handle(DoubleLiteralNode& n), handle(BooleanLiteralNode& n), handle(StringLiteralNode& n), handle(VariableNode& n), handle(IfNode& n), handle(WhileNode& n), handle(ForNode& n), handle(CaseNode& n), handle(SwitchNode& n), handle(BlockNode& n), handle(FunctionCall& n), handle(FunctionCallAsStatement& n), handle(CoutNode& n), handle(IncDecNode& n), handle(BreakNode& n), handle(ContinueNode& n), handle(ReturnNode& n), handle(VariableDeclarationNode& n), handle(AssignmentNode& n) (или TranslatingVisitor::handle) – методы уникальной обработки объектом-посетителем каждого из возможных узлов, формирующих синтаксическое дерево, с последующей генерацией кода конечной программы;

б) handle(Program& n) (или TranslatingVisitor::handle) – метод, символизирующий старт обработки синтаксического дерева и включающий в текст сформированного файла в командах языка C++ блоки кода для анализа сложности и отображение графика из заранее заготовленных файлов;

5) ProgramInitialization.txt – файл, содержащий заготовленный блок кода для инициализации необходимого для работы функционала (подключения библиотек, введения макросов и т.д.) и содержащий блок для проведения регрессионного анализа;

а) linearRegression – функция, принимающая вектор пар как некоторый набор значений функции сложности рассматриваемого алгоритма, требующий анализа методом линейной регрессии, и возвращающая значение вычисленной средней ошибки и степени x;

б) deleteOverflowing – функция, необходимая для очищения набора пар значений параметра и числа выполняемых элементарных операций от пар, содержащих значения, мешающие процессу линейной регрессии и построению графика (такое может произойти вследствие вычислений с последующим переполнением используемых типов данных);

6) ComplexityClassSearching.txt – файл, содержащий заготовленный блок кода для поиска класса сложности рассматриваемого алгоритма;

а) findComplexityClass – функция, принимающая вектор пар как некоторый набор значений функции сложности рассматриваемого алгоритма, требующий анализа методом линейной регрессии, и определяющая класс сложности алгоритма;

7) MainFunction.txt – файл, содержащий заготовленный блок кода для запуска процесса поиска сложности алгоритма и построения графика функции (построение графика осуществляется на основе значений, содержащихся в двух векторах, предназначенных для хранения значений двух переменных соответственно);

3.2 Описание работы программы

Точкой входа программы является файл sourceAlgolite. В нём пользователю предлагается с помощью разработанного языка Algolite описать алгоритм, анализ сложности которого н хочет провести. Для успешного запуска процесса трансляции и последующего анализа сложности ему необходимо описать граничные значения и шаг изменения параметра сложности и ввести функцию main (без аргументов), откуда будет происходить вызов функций, составляющих рассматриваемый алгоритм. Ввиду максимальной автоматизации приложения пользователь заканчивает своё вмешательство в процесс работы программы.

Далее начинается сканирование файла sourceAlgolite, перевод текста из него в строчный формат и анализ данной строки синтаксическим и лексическим анализаторами. В зависимости от указанных конструкций, формируется синтаксическое дерево.

Затем проводится обход дерева с помощью программной реализации паттерна проектирования Посетитель (Visitor). Объект-посетитель просматривает каждый узел организованного дерева и в зависимости от типа узла формирует часть текста конечной программы на языке C++ (он вводит команды конечного языка и добавляет счётчики для вычисления числа элементарных операций, выполняемых в описанном алгоритме).

Если хотя бы на одном из этапов произошла некоторая ошибка, программа выводит текст ошибки пользователю и прекращает процесс работы.

После успешной трансляции программа создаёт конечный .cpp файл путём соединения 3-х заранее описанных текстовых файлов (ComplexityClassSearching, MainFunction и ProgramInitialization), содержащих блоки для анализа сложности алгоритма и построение графика, и текста, сформированного в процессе обхода синтаксического дерева. Затем

выполняется работа компилятора g++ с последующим запуском исполняемого файла output_program.

После запуска исполняемого файла описанный алгоритм выполняет свою работу на различных значениях, находящихся в области граничных значений, что задал пользователь. Данный процесс необходим для выявления зависимости числа выполняемых операций от значения параметра сложности.

На основе полученных пар значений проводится регрессионный анализ, приводящий к получению двух значений ошибки и степени x в функции сложности. В зависимости от величины каждой из них и значения степени x и определяется класс сложности рассматриваемого алгоритма.

Также на основе этих пар значений проводится построение графика функции сложности алгоритма с помощью библиотеки pbPlots и его дальнейшее отображение пользователю.

На рисунках 8, 9, 10 отображён пример вывода, что может отобразить пользователю программа.

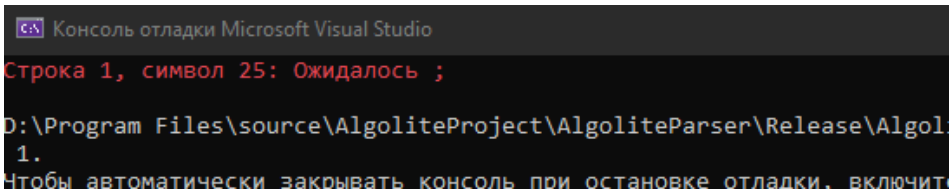


Рисунок 8 – Экстренное прекращение работы в случае ошибки

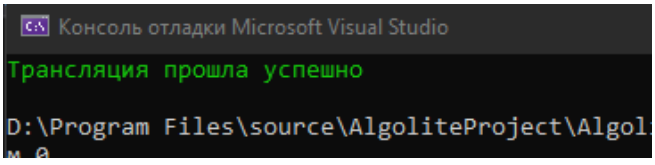


Рисунок 9 – Успешное выполнение процесса трансляции

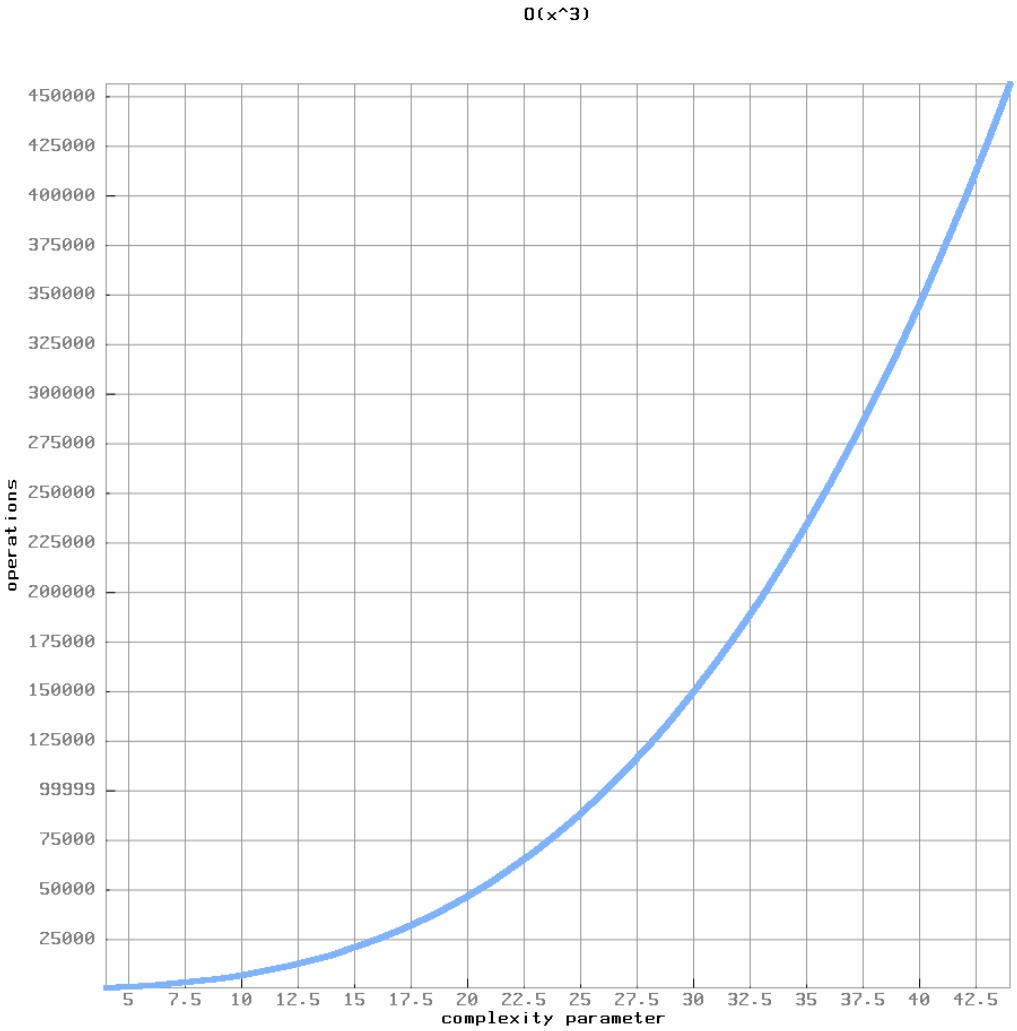


Рисунок 10 – Пример определения класса сложности некоторого алгоритма

Инструкция по работе с разработанной программой:

– открыть файл sourceAlgolite для описания алгоритма;

– объявить команду инициализации параметра сложности и ввести идентификатор параметра, граничные значения и величину шага изменения параметра при анализе;

– описать исследуемый алгоритм через набор функций;

– определить функцию main (без параметров), откуда будет запускаться весь процесс выполнения программных блоков, образующих рассматриваемый алгоритм;

– запустить процесс выполнения основной программы, описанный в файле AlgoliteParser;

Пример корректной программы на языке Algolite, описанной в соответствии с инструкцией:

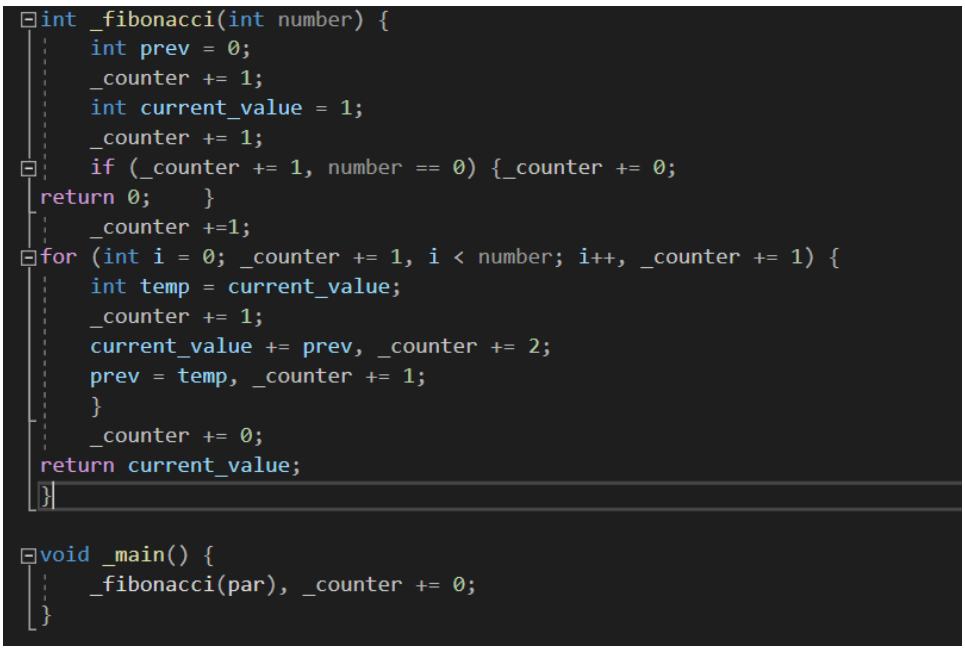
```
parameter par (4, 45, 1);
long long factorial (int n) {
    long long f =1;
    for (long long i=2; i<=n;i++)
    f*=i;
    return f;
}
void main () {
    factorial(par);
}
```

4 Результаты исследований

Для анализа эффективности работы разработанной программы достаточно пронаблюдать достоверность вставки счётчиков выполняемых операций в необходимые места (при условии корректно выбранных значений для параметра сложности). Рассмотрим следующий набор алгоритмов и классы сложности, определённые для них: нерекурсивный алгоритм поиска числа Фибоначчи, рекурсивный алгоритм поиска числа Фибоначчи, алгоритм нерекурсивный алгоритм поиска факториала, алгоритм стандартного перемножения матриц, алгоритм сортировки пузырьком (при условии худшего случая заполненности массива) и алгоритм бинарного поиска (при условии худшего случая, т.е. при отсутствии искомого элемента в массиве).

```
1   parameter par(4, 45, 1);
2
3   int fibonacci(int number)
4   {
5       int prev = 0;
6       int current_value = 1;
7
8       if (number == 0)
9           return 0;
10
11      for (int i = 0; i < number; i++) {
12          int temp = current_value;
13          current_value += prev;
14          prev = temp;
15      }
16
17      return current_value;
18  }
19
20
21  void main() {
22      fibonacci(par);
23  }
```

Рисунок 11 – Текст нерекурсивного алгоритма поиска числа Фибоначчи



```
int _fibonacci(int number) {
    int prev = 0;
    _counter += 1;
    int current_value = 1;
    _counter += 1;
    if (_counter += 1, number == 0) {_counter += 0;
return 0;    }
    _counter +=1;
for (int i = 0; _counter += 1, i < number; i++, _counter += 1) {
    int temp = current_value;
    _counter += 1;
    current_value += prev, _counter += 2;
    prev = temp, _counter += 1;
    }
    _counter += 0;
return current_value;
}

void _main() {
    _fibonacci(par), _counter += 0;
}
```

Рисунок 12 – Переработанный текст нерекурсивного алгоритма поиска числа Фибоначчи

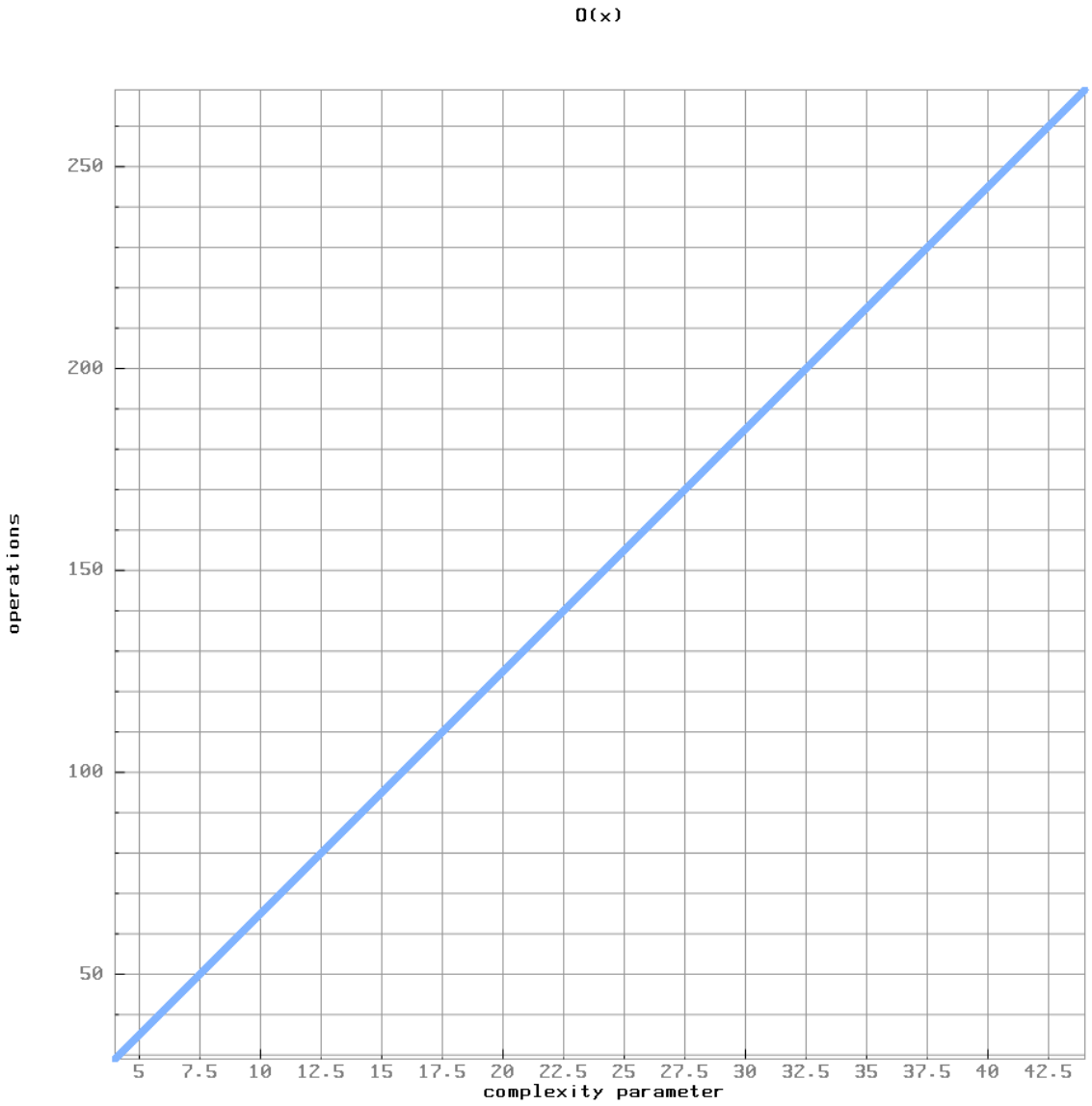


Рисунок 13 – Пример определения класса сложности нерекурсивного алгоритма поиска числа Фибоначчи

```
parameter par(4, 45, 1);

int fibonacci(int n)
{
    if (n <= 1)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

void main() {
    fibonacci(par);
}
```

Рисунок 14 – Текст рекурсивного алгоритма поиска числа Фибоначчи

```
int _fibonacci(int n) {
    if (_counter += 1, n <= 1) {_counter += 0;
return n;    }
    _counter += 3;
return _fibonacci(n - 1) + _fibonacci(n - 2);
}

void _main() {
    _fibonacci(par), _counter += 0;
}
```

Рисунок 15 – Переработанный текст рекурсивного алгоритма поиска числа Фибоначчи

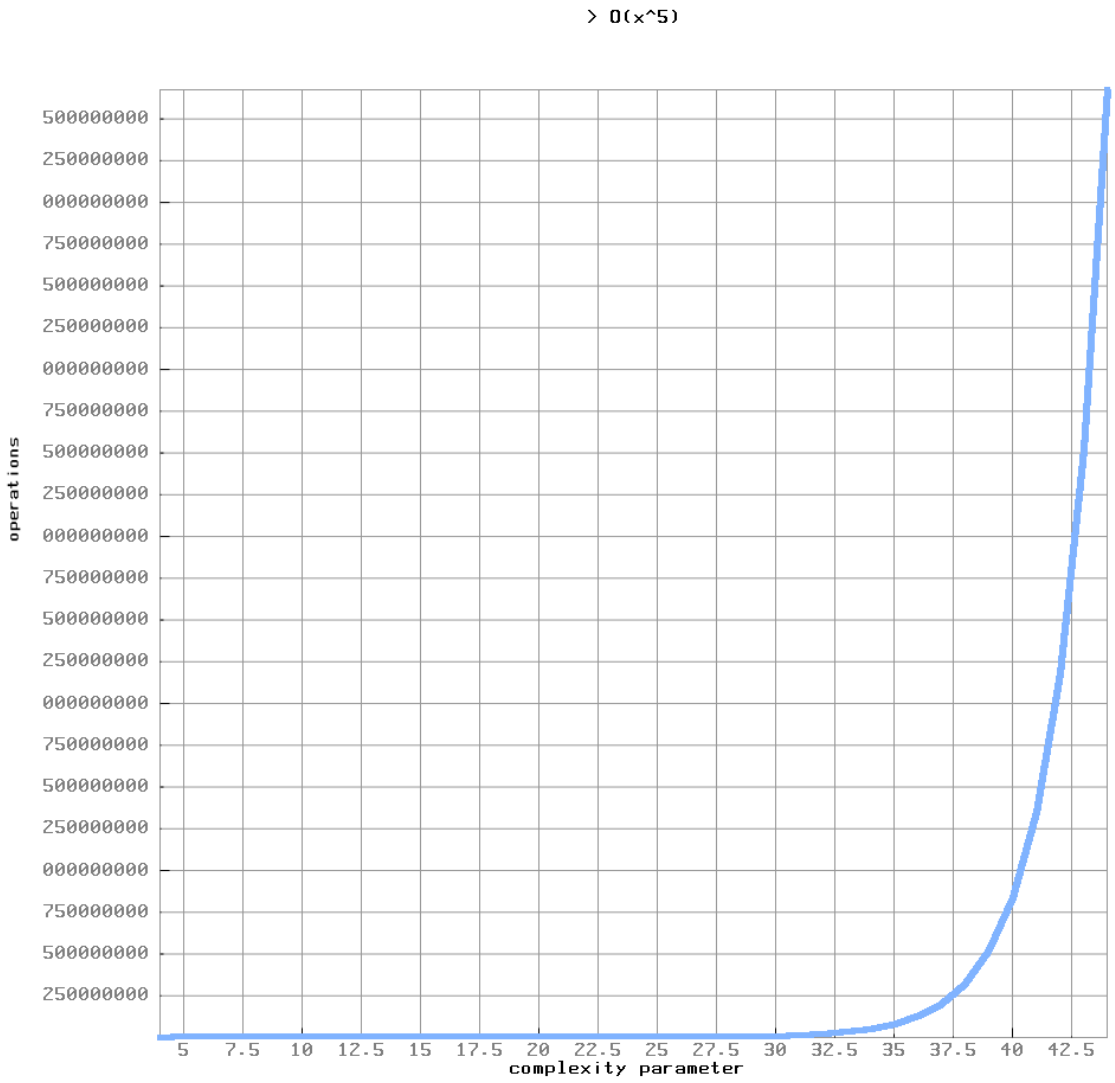


Рисунок 16 – Пример определения класса сложности рекурсивного алгоритма поиска числа Фибоначчи

```
parameter par(4, 1000, 1);

long long factorial(long long n)
{
    long long f=1;
    for(long long i=2;i<=n;i++)
        f*=i;
    return f;
}

void main() {
    factorial(par);
}
```

Рисунок 17 – Текст нерекурсивного алгоритма поиска факториала

```
long long _factorial(long long n) {
    long long f = 1;
    _counter += 1;
    _counter += 1;
for (long long i = 2; _counter += 1, i <= n; i++, _counter += 1) {f *= i, _counter += 2;    }
    _counter += 0;
return f;
}

void _main() {
    _factorial(par), _counter += 0;
}
```

Рисунок 18 – Переработанный текст нерекурсивного алгоритма поиска факториала

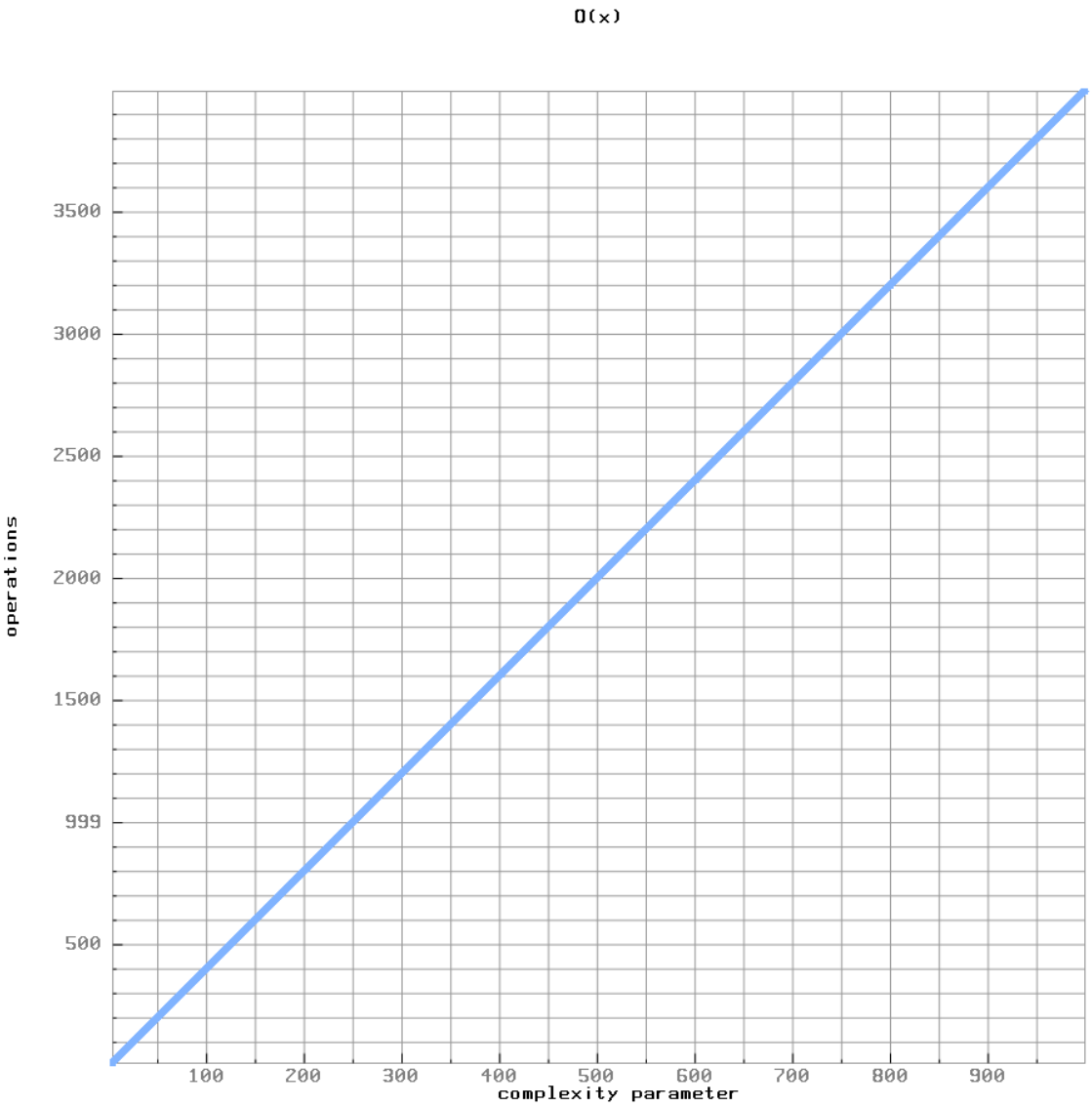
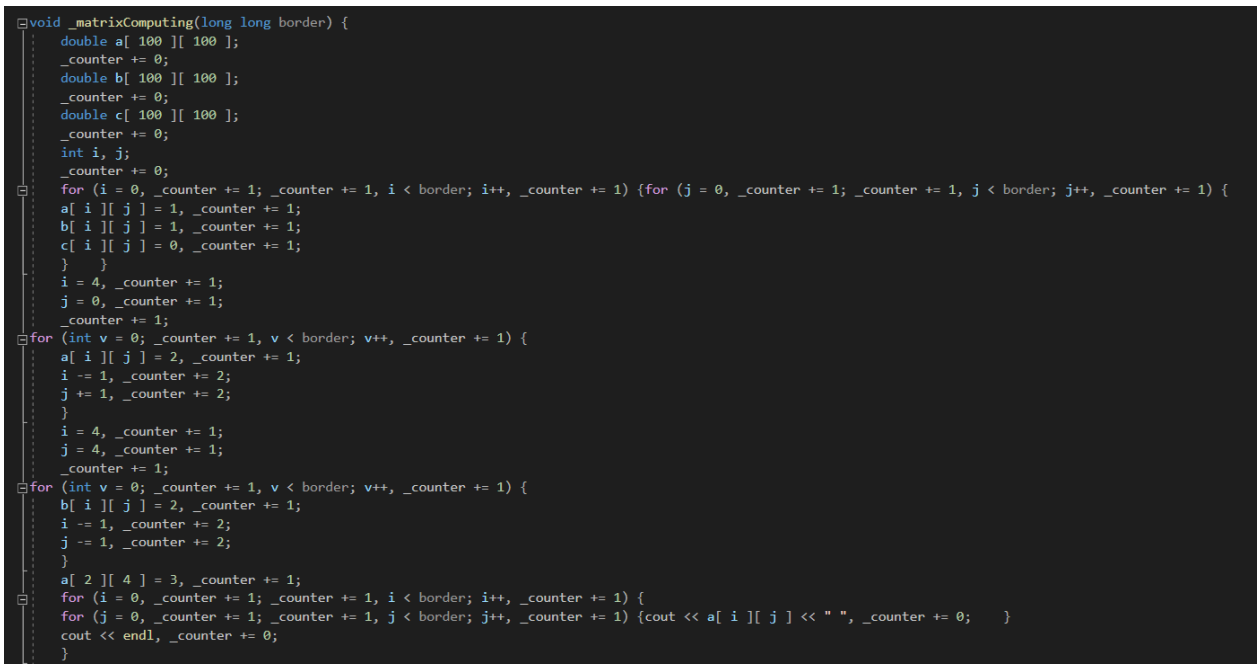


Рисунок 19 – Пример определения класса сложности нерекурсивного алгоритма поиска факториала

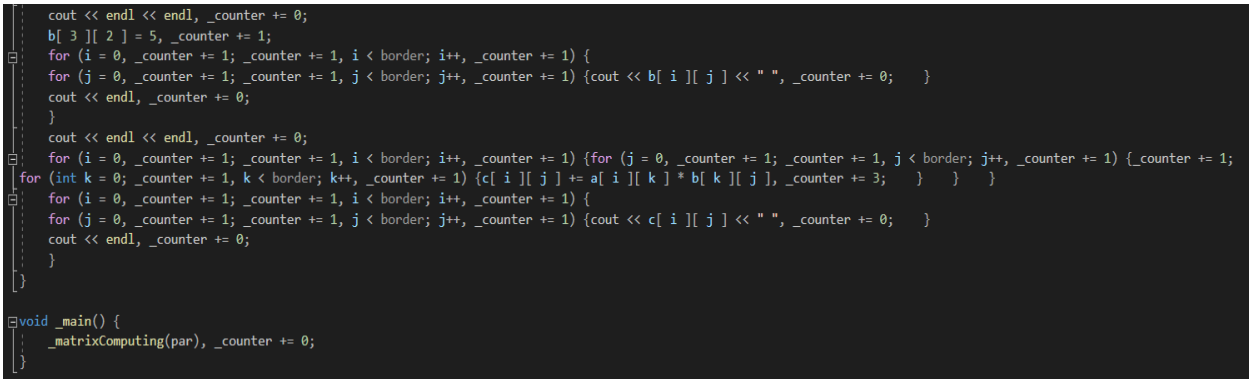
```
parameter par(4, 40, 1);

void matrixComputing(long long border) {
    double a[ 100 ][ 100 ] ; double b[ 100 ][ 100 ] ; double c[ 100 ][ 100 ] ;
        int i, j ;
        for ( i = 0 ; i < border ; ++i)
            for ( j = 0 ; j < border ; ++j) {
                a[ i ][ j ] = 1.0 ; b[ i ][ j ] = 1.0 ; c[ i ][ j ] = 0 ;
            }
        i = 4 ; j = 0 ;
        for ( int v = 0 ; v < border ; ++v) {
            a[ i ][ j ] = 2.0 ;
            i -= 1 ;
            j += 1 ;
        }
        i = 4 ; j = 4 ;
        for ( int v = 0 ; v < border ; ++v) {
            b[ i ][ j ] = 2.0 ;
            i -= 1 ;
            j -= 1 ;
        }
        a[ 2 ][ 4 ] = 3.0 ;
        for ( i = 0 ; i < border ; ++i) {
            for ( j = 0 ; j < border ; ++j)
                cout << a[ i ][ j ] << " " ;
            cout << endl ;
        }-
        cout << endl << endl ;-
        b[ 3 ][ 2 ] = 5.0 ;
        for ( i = 0 ; i < border ; ++i) {
            for ( j = 0 ; j < border ; ++j)
                cout << b[ i ][ j ] << " " ;
            cout << endl ;
        }
        cout << endl << endl ;

        for ( i = 0 ; i < border ; ++i)
            for ( j = 0 ; j < border ; ++j)
                for ( int k = 0 ; k < border ; ++k) c[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ] ;
        for ( i = 0 ; i < border ; ++i) {
            for ( j = 0 ; j < border  ; ++j)
                cout << c[ i ][ j ] << " " ;
            cout << endl ;
        }
    }

void main() {
    matrixComputing(par);
}
```

Рисунок 20 – Текст алгоритма стандартного перемножения матриц



```
void _matrixComputing(long long border) {
    double a[ 100 ][ 100 ];
    _counter += 0;
    double b[ 100 ][ 100 ];
    _counter += 0;
    double c[ 100 ][ 100 ];
    _counter += 0;
    int i, j;
    _counter += 0;
    for (i = 0, _counter += 1; _counter += 1, i < border; i++, _counter += 1) {for (j = 0, _counter += 1; _counter += 1, j < border; j++, _counter += 1) {
    a[ i ][ j ] = 1, _counter += 1;
    b[ i ][ j ] = 1, _counter += 1;
    c[ i ][ j ] = 0, _counter += 1;
    }    }
    i = 4, _counter += 1;
    j = 0, _counter += 1;
    _counter += 1;
for (int v = 0; _counter += 1, v < border; v++, _counter += 1) {
    a[ i ][ j ] = 2, _counter += 1;
    i -= 1, _counter += 2;
    j += 1, _counter += 2;
    }
    i = 4, _counter += 1;
    j = 4, _counter += 1;
    _counter += 1;
for (int v = 0; _counter += 1, v < border; v++, _counter += 1) {
    b[ i ][ j ] = 2, _counter += 1;
    i -= 1, _counter += 2;
    j -= 1, _counter += 2;
    }
    a[ 2 ][ 4 ] = 3, _counter += 1;
    for (i = 0, _counter += 1; _counter += 1, i < border; i++, _counter += 1) {
    for (j = 0, _counter += 1; _counter += 1, j < border; j++, _counter += 1) {cout << a[ i ][ j ] << " ", _counter += 0;    }
    cout << endl, _counter += 0;
    }
```

Рисунок 21 – Переработанный текст алгоритма стандартного перемножения матриц (1)



```
    cout << endl << endl, _counter += 0;
    b[ 3 ][ 2 ] = 5, _counter += 1;
    for (i = 0, _counter += 1; _counter += 1, i < border; i++, _counter += 1) {
    for (j = 0, _counter += 1; _counter += 1, j < border; j++, _counter += 1) {cout << b[ i ][ j ] << " ", _counter += 0;    }
    cout << endl, _counter += 0;
    }
    cout << endl << endl, _counter += 0;
    for (i = 0, _counter += 1; _counter += 1, i < border; i++, _counter += 1) {for (j = 0, _counter += 1; _counter += 1, j < border; j++, _counter += 1) {_counter += 1;
for (int k = 0; _counter += 1, k < border; k++, _counter += 1) {c[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ], _counter += 3;    }    }    }
    for (i = 0, _counter += 1; _counter += 1, i < border; i++, _counter += 1) {
    for (j = 0, _counter += 1; _counter += 1, j < border; j++, _counter += 1) {cout << c[ i ][ j ] << " ", _counter += 0;    }
    cout << endl, _counter += 0;
    }
}

void _main() {
    _matrixComputing(par), _counter += 0;
}
```

Рисунок 22 – Переработанный текст алгоритма стандартного перемножения матриц (2)

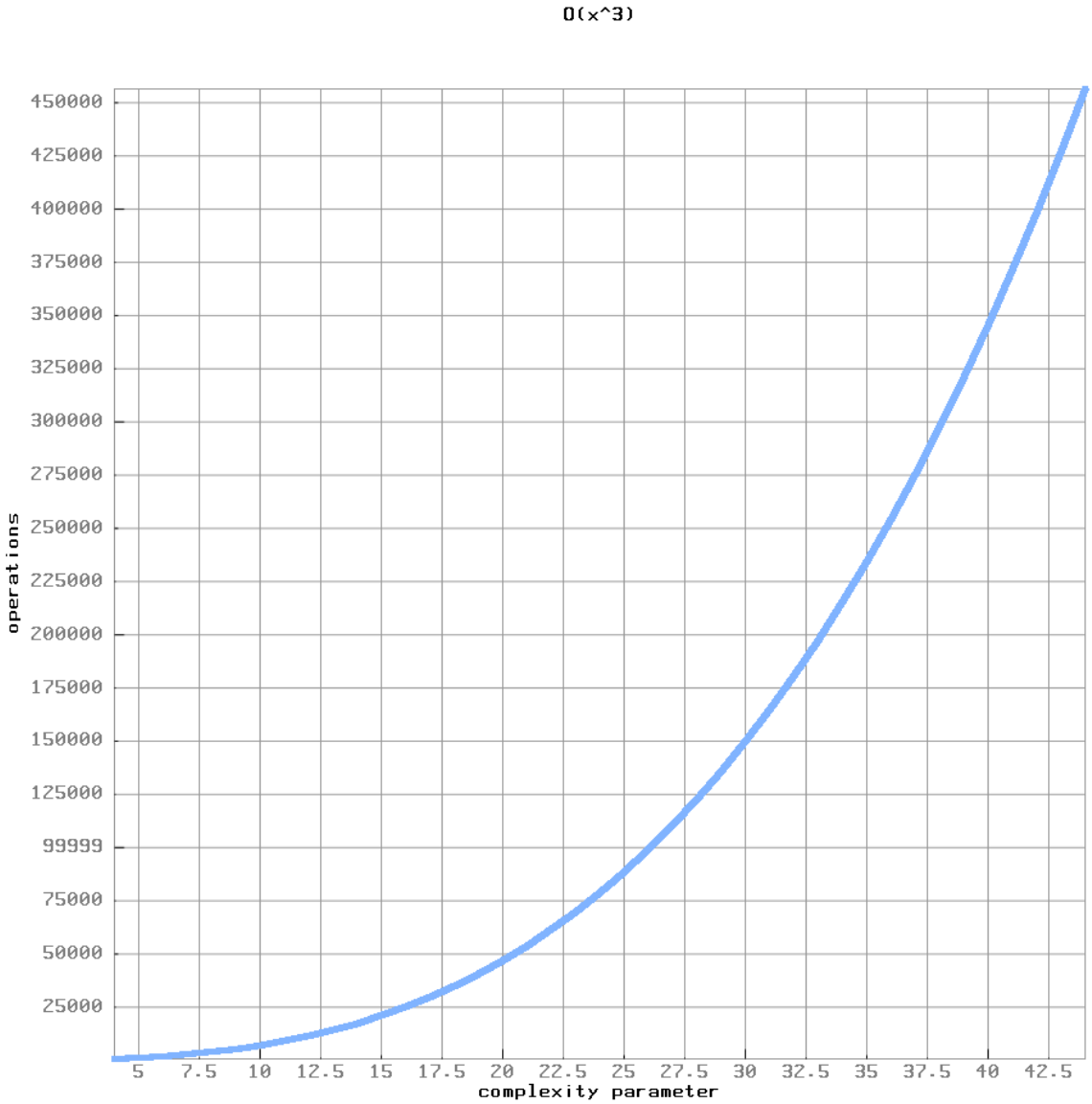
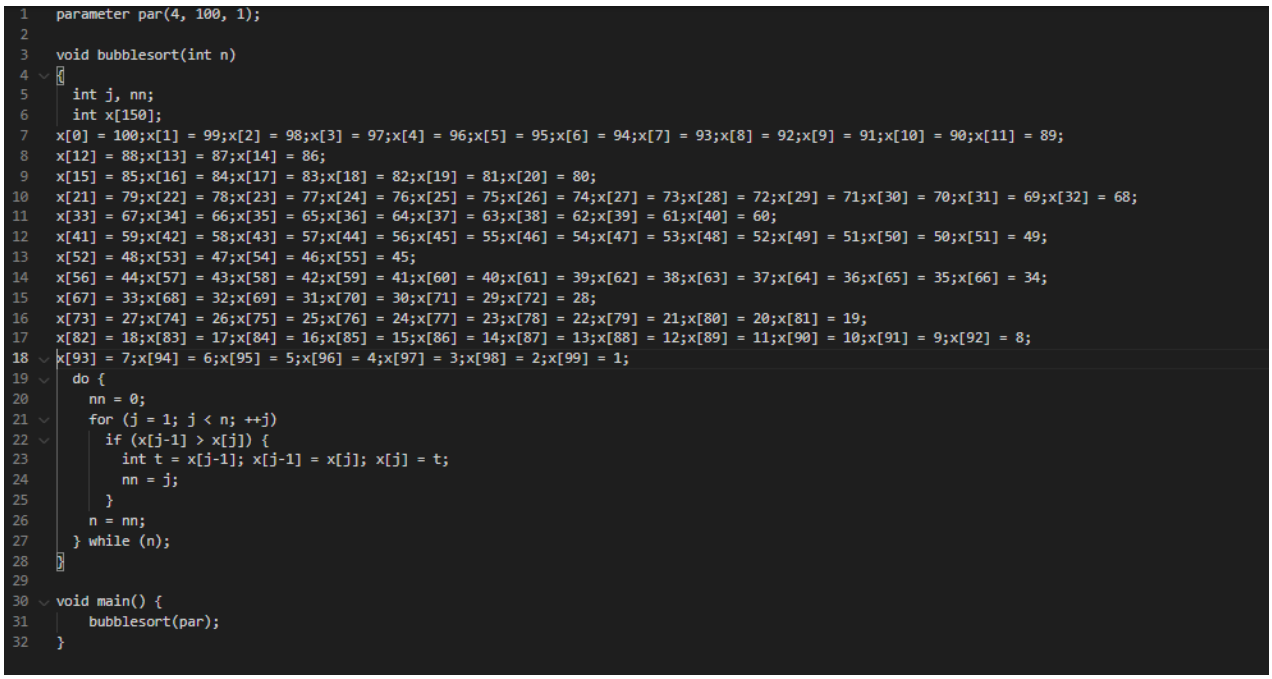
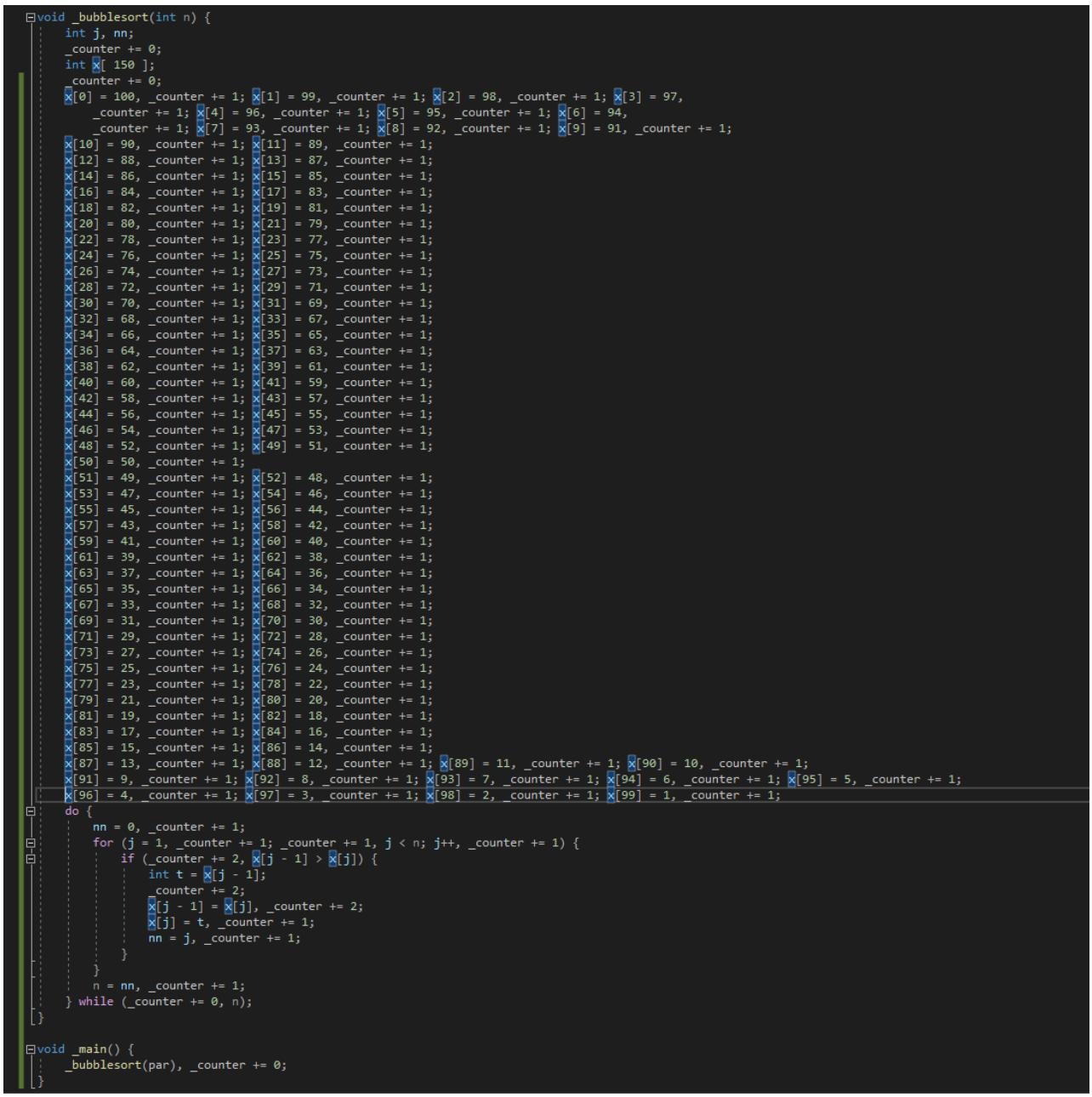


Рисунок 23 – Пример определения класса сложности алгоритма стандартного перемножения матриц



```
parameter par(4, 100, 1);

void bubblesort(int n)
{
  int j, nn;
  int x[150];
x[0] = 100;x[1] = 99;x[2] = 98;x[3] = 97;x[4] = 96;x[5] = 95;x[6] = 94;x[7] = 93;x[8] = 92;x[9] = 91;x[10] = 90;x[11] = 89;
x[12] = 88;x[13] = 87;x[14] = 86;
x[15] = 85;x[16] = 84;x[17] = 83;x[18] = 82;x[19] = 81;x[20] = 80;
x[21] = 79;x[22] = 78;x[23] = 77;x[24] = 76;x[25] = 75;x[26] = 74;x[27] = 73;x[28] = 72;x[29] = 71;x[30] = 70;x[31] = 69;x[32] = 68;
x[33] = 67;x[34] = 66;x[35] = 65;x[36] = 64;x[37] = 63;x[38] = 62;x[39] = 61;x[40] = 60;
x[41] = 59;x[42] = 58;x[43] = 57;x[44] = 56;x[45] = 55;x[46] = 54;x[47] = 53;x[48] = 52;x[49] = 51;x[50] = 50;x[51] = 49;
x[52] = 48;x[53] = 47;x[54] = 46;x[55] = 45;
x[56] = 44;x[57] = 43;x[58] = 42;x[59] = 41;x[60] = 40;x[61] = 39;x[62] = 38;x[63] = 37;x[64] = 36;x[65] = 35;x[66] = 34;
x[67] = 33;x[68] = 32;x[69] = 31;x[70] = 30;x[71] = 29;x[72] = 28;
x[73] = 27;x[74] = 26;x[75] = 25;x[76] = 24;x[77] = 23;x[78] = 22;x[79] = 21;x[80] = 20;x[81] = 19;
x[82] = 18;x[83] = 17;x[84] = 16;x[85] = 15;x[86] = 14;x[87] = 13;x[88] = 12;x[89] = 11;x[90] = 10;x[91] = 9;x[92] = 8;
x[93] = 7;x[94] = 6;x[95] = 5;x[96] = 4;x[97] = 3;x[98] = 2;x[99] = 1;
  do {
    nn = 0;
    for (j = 1; j < n; ++j)
      if (x[j-1] > x[j]) {
        int t = x[j-1]; x[j-1] = x[j]; x[j] = t;
        nn = j;
      }
    n = nn;
  } while (n);
}

void main() {
    bubblesort(par);
}
```

Рисунок 24 – Текст алгоритма сортировки пузырьком



```
void _bubblesort(int n) {
    int j, nn;
    _counter += 0;
    int x[ 150 ];
    _counter += 0;
    x[0] = 100, _counter += 1; x[1] = 99, _counter += 1; x[2] = 98, _counter += 1; x[3] = 97,
        _counter += 1; x[4] = 96, _counter += 1; x[5] = 95, _counter += 1; x[6] = 94,
        _counter += 1; x[7] = 93, _counter += 1; x[8] = 92, _counter += 1; x[9] = 91, _counter += 1;
    x[10] = 90, _counter += 1; x[11] = 89, _counter += 1;
    x[12] = 88, _counter += 1; x[13] = 87, _counter += 1;
    x[14] = 86, _counter += 1; x[15] = 85, _counter += 1;
    x[16] = 84, _counter += 1; x[17] = 83, _counter += 1;
    x[18] = 82, _counter += 1; x[19] = 81, _counter += 1;
    x[20] = 80, _counter += 1; x[21] = 79, _counter += 1;
    x[22] = 78, _counter += 1; x[23] = 77, _counter += 1;
    x[24] = 76, _counter += 1; x[25] = 75, _counter += 1;
    x[26] = 74, _counter += 1; x[27] = 73, _counter += 1;
    x[28] = 72, _counter += 1; x[29] = 71, _counter += 1;
    x[30] = 70, _counter += 1; x[31] = 69, _counter += 1;
    x[32] = 68, _counter += 1; x[33] = 67, _counter += 1;
    x[34] = 66, _counter += 1; x[35] = 65, _counter += 1;
    x[36] = 64, _counter += 1; x[37] = 63, _counter += 1;
    x[38] = 62, _counter += 1; x[39] = 61, _counter += 1;
    x[40] = 60, _counter += 1; x[41] = 59, _counter += 1;
    x[42] = 58, _counter += 1; x[43] = 57, _counter += 1;
    x[44] = 56, _counter += 1; x[45] = 55, _counter += 1;
    x[46] = 54, _counter += 1; x[47] = 53, _counter += 1;
    x[48] = 52, _counter += 1; x[49] = 51, _counter += 1;
    x[50] = 50, _counter += 1;
    x[51] = 49, _counter += 1; x[52] = 48, _counter += 1;
    x[53] = 47, _counter += 1; x[54] = 46, _counter += 1;
    x[55] = 45, _counter += 1; x[56] = 44, _counter += 1;
    x[57] = 43, _counter += 1; x[58] = 42, _counter += 1;
    x[59] = 41, _counter += 1; x[60] = 40, _counter += 1;
    x[61] = 39, _counter += 1; x[62] = 38, _counter += 1;
    x[63] = 37, _counter += 1; x[64] = 36, _counter += 1;
    x[65] = 35, _counter += 1; x[66] = 34, _counter += 1;
    x[67] = 33, _counter += 1; x[68] = 32, _counter += 1;
    x[69] = 31, _counter += 1; x[70] = 30, _counter += 1;
    x[71] = 29, _counter += 1; x[72] = 28, _counter += 1;
    x[73] = 27, _counter += 1; x[74] = 26, _counter += 1;
    x[75] = 25, _counter += 1; x[76] = 24, _counter += 1;
    x[77] = 23, _counter += 1; x[78] = 22, _counter += 1;
    x[79] = 21, _counter += 1; x[80] = 20, _counter += 1;
    x[81] = 19, _counter += 1; x[82] = 18, _counter += 1;
    x[83] = 17, _counter += 1; x[84] = 16, _counter += 1;
    x[85] = 15, _counter += 1; x[86] = 14, _counter += 1;
    x[87] = 13, _counter += 1; x[88] = 12, _counter += 1; x[89] = 11, _counter += 1; x[90] = 10, _counter += 1;
    x[91] = 9, _counter += 1; x[92] = 8, _counter += 1; x[93] = 7, _counter += 1; x[94] = 6, _counter += 1; x[95] = 5, _counter += 1;
    x[96] = 4, _counter += 1; x[97] = 3, _counter += 1; x[98] = 2, _counter += 1; x[99] = 1, _counter += 1;
    do {
        nn = 0, _counter += 1;
        for (j = 1, _counter += 1; _counter += 1, j < n; j++, _counter += 1) {
            if (_counter += 2, x[j - 1] > x[j]) {
                int t = x[j - 1];
                _counter += 2;
                x[j - 1] = x[j], _counter += 2;
                x[j] = t, _counter += 1;
                nn = j, _counter += 1;
            }
        }
        n = nn, _counter += 1;
    } while (_counter += 0, n);
}

void _main() {
    _bubblesort(par), _counter += 0;
}
```

Рисунок 25 – Переработанный текст алгоритма сортировки пузырьком

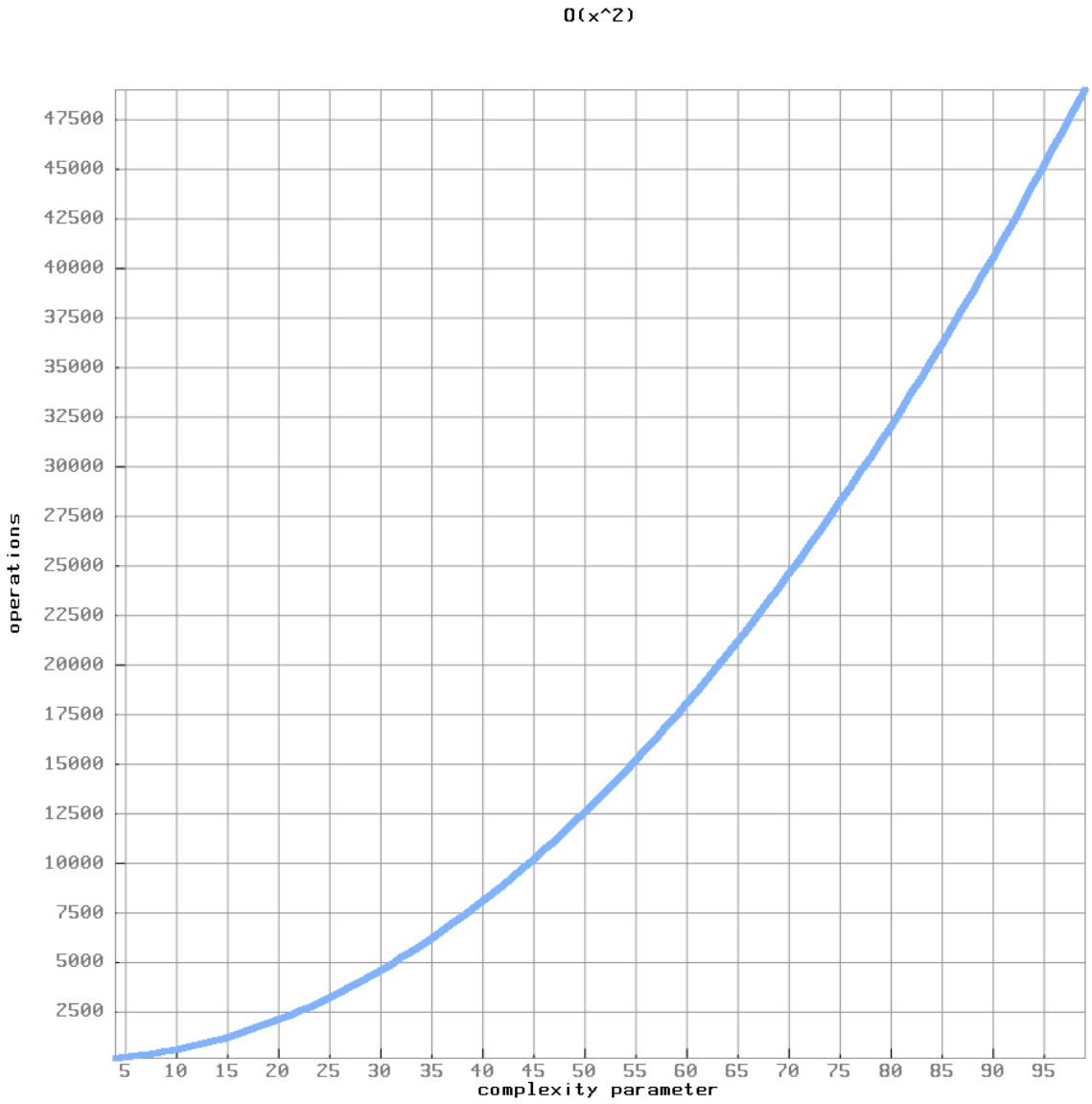


Рисунок 26 – Пример определения класса сложности алгоритма сортировки пузырьком

```
parameter par(4, 1000, 1);

void binarySearch(int border, int key)
{

    int x[1000];
    x[0] = 0;x[1] = 1;x[2] = 2;x[3] = 3;x[4] = 4;x[5] = 5;x[6] = 6;x[7] = 7;x[8] = 8;x[9] = 9;x[10] = 10;x[11] = 11;x[12] = 12;x[13] = 13;x[14] =

    int left = 0;
    int right = border;

    int midd = 0;
    while (1)
    {
        midd = (left + right) / 2;

        if (key < x[midd])
            right = midd - 1;
        else if (key > x[midd])
            left = midd + 1;
        else
            break;

        if (left > right)
            break;
    }

}

void main() {
    binarySearch(par, 1500);
}
```

Рисунок 27 – Текст алгоритма бинарного поиска (с учётом пошаговой инициализации массива)

```
void _binarySearch(int border, int key) {
    int x[ 1000 ];
    _counter += 0;
    ...
    int left = 0;
    _counter += 1;
    int right = border;
    _counter += 1;
    int midd = 0;
    _counter += 1;
    while(_counter += 0, 1) {
    midd = (left + right) / 2, _counter += 3;
    if (_counter += 1, key < x[ midd ]) {right = midd - 1, _counter += 2;    }
    else {if (_counter += 1, key > x[ midd ]) {left = midd + 1, _counter += 2;    }
    else {break;    }    }
    if (_counter += 1, left > right) {break;    }
    }
}

void _main() {
    _binarySearch(par, 1500), _counter += 0;
}
```

Рисунок 28 – Переработанный текст алгоритма бинарного поиска

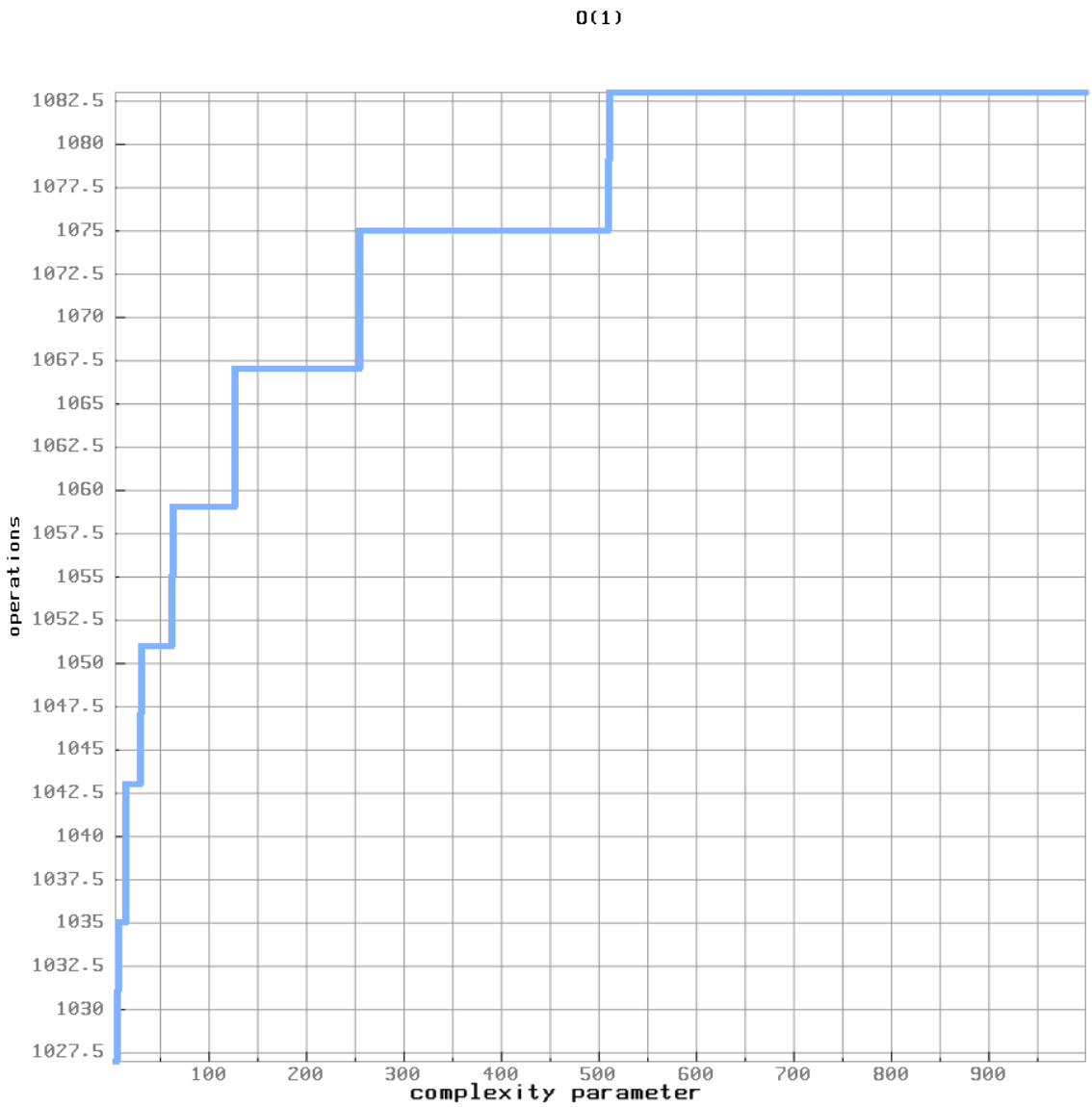


Рисунок 29 – Пример определения класса сложности алгоритма бинарного поиска (с учётом пошаговой инициализации массива)

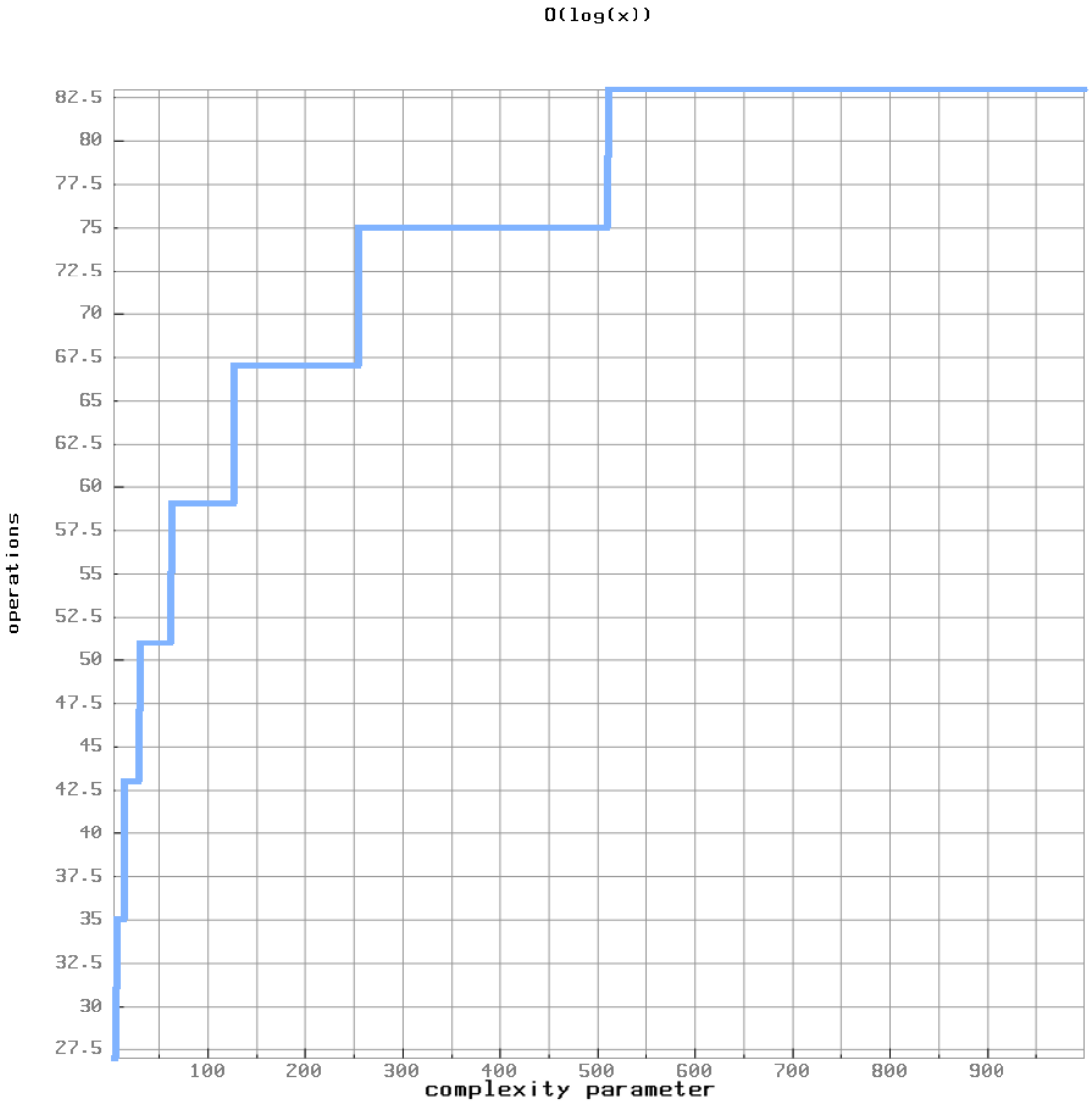


Рисунок 30 – Пример определения класса сложности алгоритма бинарного поиска (без учёта пошаговой инициализации массива)

По полученным данным можно сделать вывод, что выбор значений параметра для анализа сложности очень сильно влияет на конечный результат определения конечного класса сложности.

В примере анализа алгоритма бинарного поиска стоит сделать замечание: пошаговая инициализация массива вносит существенный вклад в количество выполнимых операций. Отбрасывая факт наличия данного этапа внутри самого алгоритма, мы получим более очевидную картину, отображенную на рисунке 30, символизирующую о медленном росте функции сложности данного алгоритма.

ЗАКЛЮЧЕНИЕ

Анализ сложности алгоритмов приходится проводить вручную, что может потребовать значительных затрат времени разработчика, пытающегося внедрить данный алгоритм в проектируемую систему. Поэтому актуальность автоматизации данного процесса и поиска оптимальных методов его программной реализации неоспорима. Дальнейшее развитие инструментария поиска сложности алгоритмов для улучшения точности определения классов сложности является важным направлением.

В результате выполнения курсовой работы была создана программа, предоставляющая возможность оценивания сложности алгоритма, написанного на разработанном языке Algolite, транслируемого в команды языка C++, с дальнейшим построением графика сложности, описывающего зависимость числа элементарных операций от заданного параметра.

С помощью созданной программы были проведены тестирования на некотором числе существующих алгоритмов, повсеместно применяемых разработчиками, отображающие высокую точность определения классов сложности, что позволяет ускорить процесс внедрения или отсеивания алгоритма при конструировании системы.

На основании полученного опыта при разработке и результатов проведённых тестов можно сделать вывод, что использование автоматизированного метода оценки сложности некоторого алгоритма на основе программной реализации с вычислениями при помощи линейной регрессии способно внести ощутимый вклад в упрощение процесса написания программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Миков, А. И. Вычислимость и сложность алгоритмов: учебное пособие / А. И. Миков, О. Н. Лапина. – Краснодар: Кубан. гос. ун-т, 2013. – 448 с.

2 Cozac, E. Big O Notation / E. Cozac // Хабр: [сайт]. – 2021. – URL: https://habr.com/ru/post/559518/ (дата обращения: 06.12.2021).

3 Conery, R. Big O / R. Conery // Хабр: [сайт]. – 2019. – URL: https://habr.com/ru/post/444594/ (дата обращения: 06.12.2021).

4 Карпов, Ю. Г. Теория и технология программирования. Основы построения трансляторов / Ю. Г. Карпов. – СПб.: БХВ-Петербург, 2005. – 272 с. – ISBN 978-5-94157-285-4.

5 Халафян, А. А. Теория вероятностей и математическая статистика: учеб. пособие / А. А. Халафян, Г. В. Калайдина, Е. Ю. Пелипенко. – Краснодар: Кубанский гос. университет, 2018. – 184 с. – ISBN 978-5-8209-1462-1.

6 Митин, И. В. Анализ и обработка экспериментальных данных: Учебно-методическое пособие для студентов младших курсов / И.В. Митин, В. С. Русаков. – М.: Физический Факультет МГУ, 2002. – 44 с. – ISBN 5-8279-0022-2.

7 Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб.: Питер, 2020. – 448 с.

ПРИЛОЖЕНИЕ А

Файл AlgoliteParser.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "Nodes.h"
#include "TranslatingVisitor.h"

#define SUCCESS_COLORIZED_COUT "\033[1m\033[32m"
#define ERROR_COLORIZED_COUT "\033[1m\033[31m"
#define STOP_COLORIZED_COUT "\033[0m"

using namespace std;
string text;
int index, line, pos;
int prev_symbol_index, prev_symbol_line, prev_symbol_pos;
string current_id;
string string_literal;
bool is_real_number;

bool has_main = false;

string readSource(string file_name) {
     ifstream f(file_name);
     f.seekg(0, ios::end);
     size_t size = f.tellg();
     string s(size, ' ');
     f.seekg(0);
     f.read(&s[0], size);
     return s;
}
```

```
enum TokenType {
     LParen = '(',
     RParen = ')',
     LBrace = '{',
     RBrace = '}',
     LBracket = '[',
     RBracket = ']',
     Comma = ',',
     Plus = '+',
     Minus = '-',
     Div = '/',
     Mod = '%',
     Mult = '*',
     Equals = '=',
     Less = '<',
     Greater = '>',
     Colon = ':',
     Semicolon = ';',
     NewLine = '\n',
     EndOfFile = '\0',
     If = 256,
     Else,
     For,
     While,
     Do,
     Switch,
     Case,
     Cout,
     LeftLeft,
     RightRight,
     And,
     Or,
```

```
    LessEq,
    GreaterEq,
    PlusEq,
    MinusEq,
    MultEq,
    DivEq,
    ModEq,
    NotEq,
    EqualsEq,
    Inc,
    Dec,
    Break,
    Continue,
    Return,
    Void,
    Int,
    Long,
    Double,
    Bool,
    Ident,
    Number,
    LogicalValue,
    StringLiteral,
    Parameter
};

TokenType symbol;
string id();
//HashTable* hashTable;
//HashIndex hashForIdent;
double number_value;
bool digit();
bool letter();
```

```
void nextSymbol();
void skipSpaces();
void readNumber();

unique_ptr<ExpressionNode> atom();
unique_ptr<FunctionCall> functionCall();
unique_ptr<StatementNode> command();

void error(string text_error) {
     cout << ERROR_COLORIZED_COUT << "Строка " <<
prev_symbol_line << ", символ " << prev_symbol_pos << ": ";
     cout << text_error << STOP_COLORIZED_COUT << "\n";
     exit(1);
}

void savePrevSymbolData() {
     prev_symbol_index = index;
     prev_symbol_pos = pos;
     prev_symbol_line = line;
}

void backupSymbolData(int backup_index, int backup_pos, int
backup_line) {
     index = backup_index;
     pos = backup_pos;
     line = backup_line;
     nextSymbol();
}

void nextSymbol() {
     is_real_number = false;
     skipSpaces();
     savePrevSymbolData();
     if (index == text.length()) {
```

```
        symbol = EndOfFile;
        return;
    }
    switch (text[index]) {
    case '(': case ')':
    case '{': case '}':
    case '[': case ']':
    case ',': case ';':
    case ':':
        symbol = (TokenType)text[index];
        index++;
        pos++;
        break;

    case '\n':
        line++;
        symbol = NewLine;
        index++;
        pos = 1;
        break;

    case '&':
        if (text[index + 1] == '&') {
            symbol = And;
            index += 2;
            pos++;
            break;
        }
        else error("Неизвестный символ");

    case '|':
        if (text[index + 1] == '|') {
            symbol = Or;
```

```
                index += 2;
                pos++;
                break;
           }
           else error("Неизвестный символ");

     case '<': case '>':
     case '-': case '+':
     case '!': case '=':
     case '/': case '%':
     case '*':
           if (text[index + 1] != '=') {
                if (text[index] == '+' && text[index + 1] == '+')
{
                     symbol = Inc;
                     index += 2;
                     pos += 2;
                     break;
                }
                else if (text[index] == '-' && text[index + 1] ==
'-') {
                     symbol = Dec;
                     index += 2;
                     pos += 2;
                     break;
                }
                else if (text[index] == '<' && text[index + 1] ==
'<') {
                     symbol = LeftLeft;
                     index += 2;
                     pos += 2;
                     break;
                }
                else if (text[index] == '>' && text[index + 1] ==
'>') {
```

```
                symbol = RightRight;
                index += 2;
                pos += 2;
                break;
            }
            else {
                symbol = (TokenType)text[index];
                index++;
                pos++;
            }
        }
        else {
            switch (text[index]) {
            case '<': symbol = LessEq; break;
            case '+': symbol = PlusEq; break;
            case '-': symbol = MinusEq; break;
            case '*': symbol = MultEq; break;
            case '/': symbol = DivEq; break;
            case '%': symbol = ModEq; break;
            case '>': symbol = GreaterEq; break;
            case '!': symbol = NotEq; break;
            case '=': symbol = EqualsEq; break;
            default: break;
            }
            index += 2;
            pos += 2;
        }
        break;

    case '"':
        symbol = StringLiteral;
        string_literal = "\"";
        index++;
```

```
            pos++;
            while (text[index] != '"') {
                  string_literal += text[index];
                  if (index == text.length())
                        error("Неизвестный символ");
                  else {
                        index++;
                        pos++;
                  }
            }
            string_literal += "\"";
            index++;
            pos++;
            break;

      default:

            if (digit()) {
                  readNumber();
            }
            else if (letter()) {
                  current_id = id();
                  if (current_id == "for") symbol = For;
                  else if (current_id == "while") symbol = While;
                  else if (current_id == "do") symbol = Do;
                  else if (current_id == "switch") symbol = Switch;
                  else if (current_id == "case") symbol = Case;
                  else if (current_id == "if") symbol = If;
                  else if (current_id == "else") symbol = Else;
                  else if (current_id == "break") symbol = Break;
                  else if (current_id == "continue") symbol =
Continue;
                  else if (current_id == "return") symbol = Return;
                  else if (current_id == "void") symbol = Void;
```

```
                else if (current_id == "int") symbol = Int;
                else if (current_id == "long") symbol = Long;
                else if (current_id == "double") symbol = Double;
                else if (current_id == "bool") symbol = Bool;
                else if (current_id == "true" || current_id ==
"false") symbol = LogicalValue;
                else if (current_id == "cout") symbol = Cout;
                else if (current_id == "parameter") symbol =
Parameter;
                else {
                     symbol = Ident;
                }
          }
          else error("Неизвестный символ");
     }
}

void initText() {
     text = readSource("sourceAlgolite.txt");
     index = 0;
     line = 1;
     pos = 1;
     nextSymbol();
}

void accept(TokenType expected) {
     if (symbol != expected) {
          string str = string(1, (char)expected);
          switch (expected) {
          case NewLine: str = "новая строка"; break;
          case EndOfFile: str = "конец файла"; break;
          case Parameter: str = "parameter"; break;
          case If: str = "if"; break;
          case Else: str = "else"; break;
```

```
case For: str = "for"; break;
case While: str = "while"; break;
case Do: str = "do"; break;
case Switch: str = "switch"; break;
case Case: str = "case"; break;
case Cout: str = "cout"; break;
case LeftLeft: str = "<<"; break;
case RightRight: str = ">>"; break;
case And: str = "&&"; break;
case Or: str = "||"; break;
case LessEq: str = "<="; break;
case GreaterEq: str = ">="; break;
case PlusEq: str = "+="; break;
case MinusEq: str = "-="; break;
case MultEq: str = "*="; break;
case DivEq: str = "/="; break;
case ModEq: str = "%="; break;
case NotEq: str = "!="; break;
case EqualsEq: str = "=="; break;
case Inc: str = "++"; break;
case Dec: str = "--"; break;
case Break: str = "break"; break;
case Continue: str = "continue"; break;
case Return: str = "return"; break;
case Void: str = "void"; break;
case Int: str = "int"; break;
case Long: str = "long "; break;
case Double: str = "double"; break;
case Bool: str = "bool"; break;
case Ident: str = "имя"; break;
case Number: str = "число"; break;
}
error("Ожидалось " + str);
```

```
     }
     nextSymbol();
}

void skipSpaces() {
     while (text[index] == ' ' || text[index] == '\t' ||
text[index] == '\r' || text[index] == '\n') {
          if (text[index] == '\n') {
               line++;
               pos = 1;
          }
          else
               pos++;
          index++;
     }
}

bool digit() {
     if (text[index] >= '0' && text[index] <= '9') return true;
     return false;
}

void readNumber() {
     symbol = Number;
     number_value = 0;
     while (digit()) {
          number_value = number_value * 10 + text[index] - '0';
          index++;
          pos++;
     }
     if (text[index] == '.') {
          is_real_number = true;
          index++;
          pos++;
```

```
          if (!digit()) {
               index--;
               pos--;
               return;
          }

          double weight = 0.1;
          while (digit()) {
               number_value += weight * (text[index] - '0');
               weight /= 10;
               index++;
               pos++;
          }
     }
}

bool letter() {
     if (text[index] == '_') return true;
     if (text[index] >= 'a' && text[index] <= 'z') return true;
     if (text[index] >= 'A' && text[index] <= 'Z') return true;
     return false;
}

string id() {
     int startIndex = index;
     if (!letter()) error("Это не буква");
     index++;
     pos++;

     while (letter() || digit()) {
          index++;
          pos++;
     }
```

```
    return text.substr(startIndex, index - startIndex);
}

double acceptNumber() {
    double result = 1;
    if (symbol == Minus) {
        nextSymbol();
        result = -1;
    }
    result *= number_value;
    accept(Number);
    return result;
}

string dataType() {
    string result_type = current_id;
    switch (symbol) {
    case Int:
    case Bool:
    case Double:
        nextSymbol();
        break;
    case Long:
        nextSymbol();
        if (symbol == Long) {
            result_type += " long";
            nextSymbol();
        }
        break;
    default:
        error("Ожидался тип данных");
    }
    return result_type;
```

```
}

unique_ptr<ExpressionNode> factor() {
     if (symbol == Minus) {
          nextSymbol();
          unique_ptr<UnaryOperationNode> result_factor =
make_unique<UnaryOperationNode>();
          result_factor->child = atom();
          result_factor->operation = '-';
          return result_factor;
     }
     else
          return atom();
}

unique_ptr<ExpressionNode> term() {
     unique_ptr<ExpressionNode> temp_factor = factor();

     if (symbol != Mult && symbol != Div && symbol != Mod)
          return temp_factor;

     else {
          unique_ptr<BinaryOperationNode> result_term =
make_unique<BinaryOperationNode>();
          result_term->children.push_back(move(temp_factor));

          switch (symbol) {
          case Mult:
               result_term->operation = "*";
               break;
          case Div:
               result_term->operation = "/";
               break;
          case Mod:
```

```
            result_term->operation = "%";
            break;
        }

        while (true) {
            if (symbol != Mult && symbol != Div && symbol != Mod)
                break;

            string current_operation;

            switch (symbol) {
            case Mult:
                current_operation = "*";
                break;
            case Div:
                current_operation = "/";
                break;
            case Mod:
                current_operation = "%";
                break;
            }

            if (current_operation != result_term->operation) {
                unique_ptr<BinaryOperationNode> new_term = make_unique<BinaryOperationNode>();
                new_term->operation = current_operation;
                new_term->children.push_back(move(result_term));
                result_term = move(new_term);
            }

            nextSymbol();
            result_term->children.push_back(factor());
```

```
          }
          return result_term;
     }
}

unique_ptr<ExpressionNode> sum() {
     unique_ptr<ExpressionNode> temp_term = term();

     if (symbol != Plus && symbol != Minus)
          return temp_term;

     else {
          unique_ptr<BinaryOperationNode> result_sum =
make_unique<BinaryOperationNode>();
          result_sum->children.push_back(move(temp_term));

          switch (symbol) {
          case Plus:
               result_sum->operation = "+";
               break;
          case Minus:
               result_sum->operation = "-";
               break;
          }

          while (true) {
               if (symbol != Plus && symbol != Minus)
                    break;

               string current_operation;

               switch (symbol) {
               case Plus:
                    current_operation = "+";
```

```
                    break;
               case Minus:
                    current_operation = "-";
                    break;
               }

               if (current_operation != result_sum->operation) {
                    unique_ptr<BinaryOperationNode> new_sum =
make_unique<BinaryOperationNode>();
                    new_sum->operation = current_operation;
                    new_sum-
>children.push_back(move(result_sum));
                    result_sum = move(new_sum);
               }

               nextSymbol();
               result_sum->children.push_back(term());
          }
          return result_sum;
     }
}

…

unique_ptr<StatementNode> simpleCommand() {
     unique_ptr<StatementNode> result_command;

     int command_start_index = prev_symbol_index;
     int command_start_pos = prev_symbol_pos;
     int command_start_line = prev_symbol_line;

     switch (symbol) {
     case Cout:
```

```
          backupSymbolData(command_start_index,
command_start_pos, command_start_line);
          result_command = coutPrinting();
          break;

     case Return: {
          unique_ptr<ReturnNode> temp_return =
make_unique<ReturnNode>();
          nextSymbol();
          temp_return->returning_expression = expression();
          result_command = move(temp_return);
          break;
     }

     case Break: {
          result_command = make_unique<BreakNode>();
          nextSymbol();
          break;
     }

     case Continue: {
          result_command = make_unique<ContinueNode>();
          nextSymbol();
          break;
     }

     case Inc:
          backupSymbolData(command_start_index,
command_start_pos, command_start_line);
          result_command = prefixIncrement();
          break;

     case Dec:
          backupSymbolData(command_start_index,
command_start_pos, command_start_line);
```

```
            result_command = prefixDecrement();
            break;

        case Int: case Bool:
        case Double: case Long:
            backupSymbolData(command_start_index,
command_start_pos, command_start_line);
            result_command = variableDeclaration();
            break;

        case Ident:
            nextSymbol();

            if (symbol == LParen) {
                backupSymbolData(command_start_index,
command_start_pos, command_start_line);
                unique_ptr<FunctionCallAsStatement> temp_func_call
= make_unique<FunctionCallAsStatement>();
                temp_func_call->function_call = functionCall();
                result_command = move(temp_func_call);
                break;
            }

            else {
                backupSymbolData(command_start_index,
command_start_pos, command_start_line);
                unique_ptr<VariableNode> temp_var = variable();

                if (symbol == Inc || symbol == Dec) {
                    unique_ptr<IncDecNode> temp_inc_dec =
make_unique<IncDecNode>();
                    temp_inc_dec->variable = move(temp_var);
                    if (symbol == Inc)
                        temp_inc_dec->is_inc = true;
                    result_command = move(temp_inc_dec);
```

```
                    nextSymbol();
                    break;
               }

               else if (symbol == PlusEq || symbol == MinusEq ||
symbol == MultEq || symbol == DivEq || symbol == ModEq || symbol
== Equals) {
                    unique_ptr<AssignmentNode> temp_assignment =
make_unique<AssignmentNode>();
                    temp_assignment->variable = move(temp_var);

                    switch (symbol) {
                    case PlusEq:
                         temp_assignment->operation = "+=";
                         break;
                    case MinusEq:
                         temp_assignment->operation = "-=";
                         break;
                    case MultEq:
                         temp_assignment->operation = "*=";
                         break;
                    case DivEq:
                         temp_assignment->operation = "/=";
                         break;
                    case ModEq:
                         temp_assignment->operation = "%=";
                         break;
                    case Equals:
                         temp_assignment->operation = "=";
                         break;
                    }

                    nextSymbol();
                    temp_assignment->value = expression();
```

```
                    result_command = move(temp_assignment);
                    break;
               }

               else
                    error("Ожидалось присваивание");
          }

     default:
          error("Ожидалась простая команда");
     }
     return result_command;
}

…

unique_ptr<StatementNode> command() {
     unique_ptr<StatementNode> result_command;
     switch (symbol) {
     case If:
          result_command = ifCommand();
          break;

     case For:
          result_command = forCommand();
          break;

     case Switch:
          result_command = switchCommand();
          break;

     case While:
          result_command = whileCommand();
```

```
            break;

        case LBrace: {
            unique_ptr<BlockNode> temp_block =
make_unique<BlockNode>();
            nextSymbol();
            while (symbol != RBrace)
                temp_block->commands.push_back(command());
            result_command = move(temp_block);
            nextSymbol();
            break;
        }

        default:
            if (symbol == Do)
                result_command = doWhileCommand();
            else
                result_command = simpleCommand();
            accept(Semicolon);
            break;
        }
        return result_command;
}

…

vector<unique_ptr<StatementNode>> functionDefinition() {
        vector<unique_ptr<StatementNode>> result_statements;

        accept(LBrace);
        while (symbol != RBrace)
            result_statements.push_back(command());

        nextSymbol();
```

```
    return result_statements;
}

unique_ptr<FunctionDefinitionNode> functionDeclaration() {
    unique_ptr<FunctionDefinitionNode> result_func_declaration =
make_unique<FunctionDefinitionNode>();
    if (symbol != Void)
        result_func_declaration->type = dataType();
    else {
        result_func_declaration->type = "void";
        nextSymbol();
    }
    result_func_declaration->name = current_id;

    accept(Ident);
    accept(LParen);

    if (symbol != RParen) {
        result_func_declaration->arguments =
functionArguments();
        accept(RParen);
    }
    else
        nextSymbol();

    if (symbol != Semicolon) {
        result_func_declaration->hasDefinition = true;
        result_func_declaration->commands =
functionDefinition();
    }
    else {
        result_func_declaration->hasDefinition = false;
        nextSymbol();
```

```
    }

    if (result_func_declaration->name == "main" &&
result_func_declaration->hasDefinition)
        has_main = true;

    return result_func_declaration;
}

Program program() {
    Program result_program;

    accept(Parameter);
    if (symbol == Ident) {
        result_program.parameter_name = current_id;
        nextSymbol();
        accept(LParen);

        result_program.parameter_min = number_value;
        accept(Number);

        accept(Comma);

        result_program.parameter_max = number_value;
        accept(Number);

        accept(Comma);

        result_program.parameter_step = number_value;
        accept(Number);

        accept(RParen);
        accept(Semicolon);
    }
```

```
    else
        error("Ожидалось имя параметра сложности");

    while (symbol != EndOfFile) {
    result_program.functions.push_back(functionDeclaration());
    }

    if (!has_main)
        error("Отсутствует начальная точка для выполнения программы в виде описанной функции main");

    return result_program;
}

int main() {
    setlocale(LC_ALL, "ru");

    initText();
    TranslatingVisitor visitor;
    Program current_program = program();
    current_program.visit(visitor);

    cout << SUCCESS_COLORIZED_COUT << "Трансляция прошла успешно" << STOP_COLORIZED_COUT << "\n";

    system("g++ -O3 -o output_program.exe final_output.cpp pbPlots.cpp supportLib.cpp -lm");
    system("start output_program.exe");
}
```

ПРИЛОЖЕНИЕ Б

Файл TranslatingVisitor.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include "Nodes.h"
#include "TranslatingVisitor.h"

#define ERROR_COLORIZED_COUT "\033[1m\033[31m"
#define STOP_COLORIZED_COUT "\033[0m"

bool in_for_translating = false;
bool writing_mode = true;
stringstream analysing_program;

int _expression_counter;

void TranslatingVisitor::handle(ArgumentNode& n) {
     if (writing_mode) {
          analysing_program << n.type << " " << n.name;

          if (n.is_array) {
               analysing_program << "[]";
          }
     }

     for (const unique_ptr<ExpressionNode>& expression :
n.arrays_sizes) {
          if (writing_mode)
               analysing_program << "[ ";
          expression->visit(*this);
          if (writing_mode)
               analysing_program << " ]";
     }
}

void TranslatingVisitor::handle(FunctionDefinitionNode& n) {
     if (writing_mode) {
          analysing_program << n.type << " _" << n.name;

          analysing_program << "(";
     }

     int arguments_size = n.arguments.size();

     if (n.name == "main" && arguments_size != 0) {
          cout << ERROR_COLORIZED_COUT << "Функция main не должна
иметь аргументов" << STOP_COLORIZED_COUT << "\n";
```

```
            exit(1);
        }

        for (int i = 0; i < arguments_size; i++) {
            n.arguments[i]->visit(*this);
            if (i < arguments_size - 1 && writing_mode)
                analysing_program << ", ";
        }
        if (writing_mode)
            analysing_program << ")";

        if (n.hasDefinition) {
            if (writing_mode)
                analysing_program << " {" << "\n";
            for   (const   unique_ptr<StatementNode>&   command   :
n.commands) {
                if (writing_mode)
                    analysing_program << "    ";
                command->visit(*this);
                if (writing_mode)
                    analysing_program << "\n";
            }
            if (writing_mode)
                analysing_program << "}";
        }
        else if (writing_mode)
            analysing_program << ";" << "\n";
}

void TranslatingVisitor::handle(ParenExpressionNode& n) {
        if (writing_mode)
            analysing_program << "(";
        n.child->visit(*this);
        if (writing_mode)
            analysing_program << ")";
}

…

void TranslatingVisitor::handle(ForNode& n) {
        if
(dynamic_cast<VariableDeclarationNode*>(n.initialization.get())
!= nullptr) {
            _expression_counter = 0;
            writing_mode = false;

            n.initialization->visit(*this);

            writing_mode = true;

            analysing_program    <<    "_counter    +=    "    <<
_expression_counter << ";\n";
        }
```

```
     analysing_program << "for " << "(";
     in_for_translating = true;

     if (n.initialization != nullptr)
          n.initialization->visit(*this);
     if (writing_mode)
          analysing_program << "; ";

     if (n.condition != nullptr) {
          _expression_counter = 0;
          writing_mode = false;

          n.condition->visit(*this);

          writing_mode = true;

          analysing_program    <<    "_counter    +=    "    <<
_expression_counter << ", ";
          n.condition->visit(*this);

     }
     analysing_program << "; ";

     if (n.step != nullptr)
          n.step->visit(*this);

     analysing_program << ") ";
     in_for_translating = false;


     if (dynamic_cast<BlockNode*>(n.command.get()) != nullptr) {
          n.command->visit(*this);
     }
     else {
          analysing_program << "{";
          n.command->visit(*this);
          analysing_program << "    }";
     }
}

…

void TranslatingVisitor::handle(Program& n) {
     ifstream
initialization_file("ProgramPartFiles/ProgramInitialization.txt"
);

     if (initialization_file && writing_mode) {
          analysing_program << initialization_file.rdbuf();
          initialization_file.close();
     }
```

```
    if (writing_mode)
        analysing_program << "long long" << " " << n.parameter_name << ", _counter;" << "\n\n";

    for (const unique_ptr<FunctionDefinitionNode>& func_definition : n.functions) {
        func_definition->visit(*this);
        if (writing_mode)
            analysing_program << "\n\n";
    }

    if (writing_mode) {
        analysing_program << "vector<pair<long long, double>> createComplexityFunc() {" << "\n";
        analysing_program << "   vector<pair<long long, double>> result_func;" << "\n";
        analysing_program << " for (" << n.parameter_name << " = " << n.parameter_min << "; " << n.parameter_name << " < " << n.parameter_max << "; " << n.parameter_name << " += " << n.parameter_step << ") {" << "\n";
        analysing_program << "         _counter = 0;" << "\n";
        analysing_program << "         _main();" << "\n";
        analysing_program << "                 result_func.push_back(make_pair(" << n.parameter_name << ", _counter)); " << "\n";
        analysing_program << "     }" << "\n";
        analysing_program << "     return result_func;" << "\n";
        analysing_program << "}" << "\n\n";

        analysing_program << "vector<pair<long long, double>> createLogarithmicComplexityFunc() {" << "\n";
        analysing_program << "   vector<pair<long long, double>> result_func;" << "\n";
        analysing_program << " for (" << n.parameter_name << " = " << n.parameter_min << "; " << n.parameter_name << " < " << n.parameter_max << "; " << n.parameter_name << " += " << n.parameter_step << ") {" << "\n";
        analysing_program << "         _counter = 0;" << "\n";
        analysing_program << "         _main();" << "\n";
        analysing_program << "                 result_func.push_back(make_pair(" << n.parameter_name << ", _counter / log10(" << n.parameter_name << "))); " << "\n";
        analysing_program << "     }" << "\n";
        analysing_program << "     return result_func;" << "\n";
        analysing_program << "}" << "\n\n";
    }

    ifstream complexity_class_file("ProgramPartFiles/ComplexityClassSearching.txt");
    if (complexity_class_file) {
        if (writing_mode)
```

```
               analysing_program                                    <<
complexity_class_file.rdbuf();
          complexity_class_file.close();
     }

     ifstream
main_func_file("ProgramPartFiles/MainFunction.txt");
     if (main_func_file) {
          if (writing_mode)
               analysing_program << main_func_file.rdbuf();
          main_func_file.close();
     }

     ofstream f("final_output.cpp", ios::out);
     f << analysing_program.str();
     f.close();
}
```

ПРИЛОЖЕНИЕ В

Файл Nodes.h

```
#pragma once
#include <string>
#include <vector>
using namespace std;

struct Node;
struct ExpressionNode;
struct StatementNode;
struct ArgumentNode;
struct FunctionDefinitionNode;
struct ParenExpressionNode;
struct BinaryOperationNode;
struct UnaryOperationNode;
struct IntegerLiteralNode;
struct DoubleLiteralNode;
struct BooleanLiteralNode;
struct StringLiteralNode;
struct VariableNode;
struct IfNode;
struct WhileNode;
struct ForNode;
struct CaseNode;
struct SwitchNode;
struct BlockNode;
struct FunctionCall;
struct FunctionCallAsStatement;
struct CoutNode;
struct IncDecNode;
struct BreakNode;
struct ContinueNode;
struct ReturnNode;
struct VariableDefinitionPart;
struct VariableDeclarationNode;
struct AssignmentNode;
struct Program;

struct Visitor {
     virtual void handle(ArgumentNode& n) = 0;
     virtual void handle(FunctionDefinitionNode& n) = 0;
     virtual void handle(ParenExpressionNode& n) = 0;
     virtual void handle(BinaryOperationNode& n) = 0;
     virtual void handle(UnaryOperationNode& n) = 0;
     virtual void handle(IntegerLiteralNode& n) = 0;
     virtual void handle(DoubleLiteralNode& n) = 0;
     virtual void handle(BooleanLiteralNode& n) = 0;
     virtual void handle(StringLiteralNode& n) = 0;
     virtual void handle(VariableNode& n) = 0;
     virtual void handle(IfNode& n) = 0;
```

```
    virtual void handle(WhileNode& n) = 0;
    virtual void handle(ForNode& n) = 0;
    virtual void handle(CaseNode& n) = 0;
    virtual void handle(SwitchNode& n) = 0;
    virtual void handle(BlockNode& n) = 0;
    virtual void handle(FunctionCall& n) = 0;
    virtual void handle(FunctionCallAsStatement& n) = 0;
    virtual void handle(CoutNode& n) = 0;
    virtual void handle(IncDecNode& n) = 0;
    virtual void handle(BreakNode& n) = 0;
    virtual void handle(ContinueNode& n) = 0;
    virtual void handle(ReturnNode& n) = 0;
    virtual void handle(VariableDeclarationNode& n) = 0;
    virtual void handle(AssignmentNode& n) = 0;
    virtual void handle(Program& n) = 0;
};

struct Node {
    virtual void visit(Visitor& v) = 0;
};

struct ExpressionNode : public Node {};
struct StatementNode : public Node {};

struct ArgumentNode : public Node {
    string type;
    string name;
    bool is_array;
    vector<unique_ptr<ExpressionNode>> arrays_sizes;

    void visit(Visitor& v) override {
        v.handle(*this);
    }
};

struct FunctionDefinitionNode : public Node {
    string type;
    string name;
    bool hasDefinition;
    vector<unique_ptr<ArgumentNode>> arguments;
    vector<unique_ptr<StatementNode>> commands;

    void visit(Visitor& v) override {
        v.handle(*this);
    }
};

struct ParenExpressionNode : public ExpressionNode {
    unique_ptr<ExpressionNode> child;

    void visit(Visitor& v) override {
        v.handle(*this);
    }
```

```
};


struct BinaryOperationNode : public ExpressionNode {
     vector<unique_ptr<ExpressionNode>> children;
     string operation;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct UnaryOperationNode : public ExpressionNode {
     unique_ptr<ExpressionNode> child;
     string operation;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct IntegerLiteralNode : public ExpressionNode {
     long long value;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct DoubleLiteralNode : public ExpressionNode {
     double value;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct BooleanLiteralNode : public ExpressionNode {
     bool value;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct StringLiteralNode : public ExpressionNode {
     string value;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};
```

```
struct VariableNode : public ExpressionNode {
     string name;
     vector<unique_ptr<ExpressionNode>> index_expressions;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct IfNode : public StatementNode {
     unique_ptr<ExpressionNode> condition;
     unique_ptr<StatementNode> command;
     unique_ptr<StatementNode> else_command;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct WhileNode : public StatementNode {
     bool is_do_while;
     unique_ptr<ExpressionNode> condition;
     unique_ptr<StatementNode> command;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct ForNode : public StatementNode {
     unique_ptr<StatementNode> initialization;
     unique_ptr<ExpressionNode> condition;
     unique_ptr<StatementNode> step;
     unique_ptr<StatementNode> command;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct CaseNode : public StatementNode {
     unique_ptr<ExpressionNode> condition;
     vector<unique_ptr<StatementNode>> commands;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct SwitchNode : public StatementNode {
     unique_ptr<ExpressionNode> expression_for_switch;
     vector<unique_ptr<CaseNode>> cases;
```

```
     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct BlockNode : public StatementNode {
     vector<unique_ptr<StatementNode>> commands;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct FunctionCall : public ExpressionNode {
     string name;
     vector<unique_ptr<ExpressionNode>> argument_expressions;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct FunctionCallAsStatement : public StatementNode {
     unique_ptr<FunctionCall> function_call;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct CoutNode : public StatementNode {
     vector<unique_ptr<ExpressionNode>> printing_expressions;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct IncDecNode : public StatementNode {
     bool is_inc;
     unique_ptr<VariableNode> variable;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct BreakNode : public StatementNode {
     void visit(Visitor& v) override {
          v.handle(*this);
     }
};
```

```
struct ContinueNode : public StatementNode {
     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct ReturnNode : public StatementNode {
     unique_ptr<ExpressionNode> returning_expression;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct VariableDefinitionPart {
     string name;
     unique_ptr<ExpressionNode> value;
     vector<unique_ptr<ExpressionNode>> arrays_sizes;
};

struct VariableDeclarationNode : public StatementNode {
     string variable_type;
     vector<VariableDefinitionPart> declarations;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct AssignmentNode : public StatementNode {
     unique_ptr<VariableNode> variable;
     string operation;
     unique_ptr<ExpressionNode> value;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};

struct Program : Node {
     vector<unique_ptr<FunctionDefinitionNode>> functions;
     string parameter_name;
     long long parameter_min;
     long long parameter_max;
     long long parameter_step;

     void visit(Visitor& v) override {
          v.handle(*this);
     }
};
```

ПРИЛОЖЕНИЕ Г

Файл ComplexityClassSearching.txt

```
void findComplexityClass(vector<pair<long long, double>>& func)
{
    vector<pair<long long, double>> logarithmic_func;
    pair<double, double> params;
    int first_degree, second_degree;
    double first_error, second_error;

    func = createComplexityFunc();

    deleteOverflowing(func);

    params = linearRegression(func);
    first_error = params.first;
    first_degree = (int) round(abs(params.second));

    logarithmic_func = createLogarithmicComplexityFunc();

    deleteOverflowing(logarithmic_func);

    params = linearRegression(logarithmic_func);
    second_error = params.first;
    second_degree = (int) round(abs(params.second));

    complexity_class = "O(";

    if (first_error < second_error) {
        if (first_degree != 0) {
            complexity_class += "x";
            if (first_degree > 1 && first_degree <= 5)
                    complexity_class += "^" +
to_string(first_degree);
               else if (first_degree > 5)
                        complexity_class = "> " +
complexity_class + "^5";
        }
        else
            complexity_class += "1";
    }
    else {
        if (second_degree != 0) {
            complexity_class += "x";
            if (second_degree > 1 && second_degree <= 5)
                    complexity_class += "^" +
to_string(second_degree);
               else if (second_degree > 5)
                        complexity_class = "> " +
complexity_class + "^5";
            complexity_class += " * ";
```

```
        }
        complexity_class += "log(x)";
    }

    complexity_class += ")";

    w = wstring(complexity_class.begin(),
complexity_class.end());
    graph_title = w.c_str();

}
```

ПРИЛОЖЕНИЕ Д

Файл MainFunction.txt

```
int main() {
    vector<pair<long long, double>> func;
    findComplexityClass(func);

    RGBABitmapImageReference* imageReference =
CreateRGBABitmapImageReference();

    for (auto points_pair = func.begin(); points_pair !=
func.end(); points_pair++) {
        xs.push_back((*points_pair).first);
        ys.push_back((*points_pair).second);
    }

    ScatterPlotSeries* series =
GetDefaultScatterPlotSeriesSettings();
    series->xs = &xs;
    series->ys = &ys;
    series->linearInterpolation = true;
    series->lineThickness = 5;
    series->color = CreateRGBColor(0.5, 0.7, 1);

    ScatterPlotSettings* settings =
GetDefaultScatterPlotSettings();
    settings->width = 1000;
    settings->height = 1000;
    settings->autoBoundaries = true;
    settings->autoPadding = true;
    settings->title = toVector(graph_title);
    settings->xLabel = toVector(L"parameter");
    settings->yLabel = toVector(L"operations");
    settings->scatterPlotSeries->push_back(series);
    settings->gridColor = CreateRGBColor(0.6, 0.6, 0.6);
    settings->showGrid = true;
    DrawScatterPlotFromSettings(imageReference, settings);

    vector<double>* pngdata = ConvertToPNG(imageReference-
>image);

    WriteToFile(pngdata, "func.png");
    DeleteImage(imageReference->image);
    system("start func.png");
}
```

ПРИЛОЖЕНИЕ Е

Файл ProgramInitialization.txt

```
#include "pbPlots.hpp"
#include "supportLib.hpp"
#include <vector>
#include <cmath>
#include <iostream>
#include <limits>
#include <string>
#define LLONG_MAX numeric_limits<long long>().max()

using namespace std;

vector<double> xs;
vector<double> ys;
vector<double> xs1;
vector<double> ys1;
string complexity_class;
const wchar_t* graph_title;
wstring w;

pair<double, double> linearRegression(vector<pair<long long,
double>> func) {
    double error = 0;
    long long n = func.size();
    double a;
    double b;

    double v_sum = 0;
    double u_sum = 0;
    double vu_sum = 0;
    double vv_sum = 0;

    for (long long i = 0; i < n; i++) {

        double v = log10(func[i].first);
        double u = log10(func[i].second);

        v_sum += v;
        u_sum += u;
        vu_sum += v * u;
        vv_sum += v * v;
    }

    b = (n * vu_sum - v_sum * u_sum) / (n * vv_sum - v_sum *
v_sum);
    a = (u_sum - b * v_sum) / n;

    for (long long i = 0; i < n; i++) {
        double v = log10(func[i].first);
```

```
        double u = log10(func[i].second);

        double estimated_u = a + b * v;
        error += (estimated_u - u) * (estimated_u - u);
    }

    error /= n;
    return make_pair(error, b);
}

void deleteOverflowing(vector<pair<long long, double>>& func) {
    for (auto points_pair = func.begin(); points_pair !=
func.end(); points_pair++) {
        if ((*points_pair).second > 0.6 * LLONG_MAX ||
(*points_pair).second < 0.0) {
            func.erase(points_pair, func.end());
            break;
        }
    }
}
```