Министерство науки и образования РФ

Федеральное государственное бюджетное учреждение

высшего образования

**«Тверской государственный технический университет»**

(ТвГТУ)

Кафедра программного обеспечения

**Отчет по курсовой работе**

по дисциплине: «Теория алгоритмов»

Тема: «Реализация задач полиномиальной сложности»

Выполнил:

студент группы

Б.ПИН.РИС-22.06

Невзоров П. Е.

Проверила:

старший преподаватель кафедры ПО

Корнеева Е.И.

Тверь 2024

**Содержание**

[Введение 3](#_Toc194279209)

[Глава 1. Теоретические аспекты по теме 6](#_Toc194279210)

[1.1. Определение полиномиальной сложности 6](#_Toc194279211)

[1.2. Примеры задач полиномиальной сложности 6](#_Toc194279212)

[1.3. Реализация интерфейса на PyQt5 6](#_Toc194279213)

[Глава 2. Практическая часть 8](#_Toc194279214)

[2.1. Структура проекта 8](#_Toc194279215)

[2.2 Реализация модуля sudoku\_solver.py 9](#_Toc194279216)

[2.3 Реализация модуля sudoku\_solver\_gui.py 10](#_Toc194279217)

[2.4 Реализация модуля test\_sudoku\_solver.py 11](#_Toc194279218)

[Глава 3. Тестирование 14](#_Toc194279219)

[3.1. Unit-тесты 14](#_Toc194279220)

[3.2. Интеграционное тестирование 14](#_Toc194279221)

[Результаты интеграционного тестирования: 20](#_Toc194279222)

[Заключение 21](#_Toc194279223)

[Список литературы 23](#_Toc194279224)

[Листинг кода 24](#_Toc194279225)

[Ссылка на GitHub 36](#_Toc194279226)

# **Введение**

Современные вычислительные системы играют ключевую роль в решении широкого спектра задач — от обработки данных до оптимизации процессов в различных областях науки и техники. Одним из фундаментальных аспектов разработки таких систем является анализ вычислительной сложности алгоритмов, который позволяет оценить их эффективность и применимость. Особое внимание в этом контексте уделяется задачам полиномиальной сложности, относящимся к классу P, поскольку они характеризуются приемлемым временем выполнения даже при увеличении объёма входных данных. Такие задачи составляют основу многих практических приложений, включая сортировку, поиск и решение логических головоломок.

**Актуальность темы** обусловлена возрастающей потребностью в эффективных алгоритмах, способных решать сложные задачи в реальном времени. В условиях роста объёмов данных и развития технологий искусственного интеллекта умение выделять и реализовывать полиномиальные подзадачи становится важным навыком для разработчиков. Судоку, как классическая логическая головоломка, представляет собой интересный пример задачи, которая в общем случае относится к классу NP, но включает полиномиальные подзадачи, такие как проверка корректности размещения чисел. Изучение и реализация подобных задач позволяют глубже понять принципы оптимизации алгоритмов и их практического применения, что делает тему исследования востребованной как в образовательных, так и в прикладных целях.

**Целью данной курсовой работы** является изучение теоретических основ задач полиномиальной сложности и разработка программного приложения для решения судоку, демонстрирующего применение таких задач на практике.

**Для достижения поставленной цели были сформулированы следующие задачи:**

1. Провести анализ теоретических аспектов вычислительной сложности, включая определение и классификацию задач полиномиального времени.
2. Разработать алгоритм решения судоку с использованием метода возврата и оценить его сложность.
3. Реализовать графический интерфейс пользователя с применением библиотеки PyQt5 для удобного взаимодействия с алгоритмом.
4. Создать систему тестирования с использованием фреймворка pytest для проверки корректности и надёжности разработанного решения.
5. Оформить результаты исследования, подведя итоги и выявив перспективы дальнейшего развития проекта.

**Объект исследования** — задачи полиномиальной сложности в контексте вычислительных алгоритмов.  
**Предмет исследования** — реализация алгоритма решения судоку как примера задачи, включающей полиномиальные подзадачи.

Работа состоит из введения, теоретической главы, описания практической реализации, тестирования, заключения и списка литературы. В первой главе рассматриваются основы теории сложности и примеры задач с полиномиальным временем выполнения. Вторая часть посвящена разработке алгоритма и интерфейса, включая программный код и его визуальное оформление. Третья часть описывает процесс тестирования, а в заключении подводятся итоги и определяются направления дальнейших исследований.

Разработка решателя судоку с удобным интерфейсом и системой тестирования позволяет не только продемонстрировать теоретические знания, но и создать практический инструмент, который может быть использован для обучения или развлечения. Таким образом, данная работа вносит вклад в изучение и популяризацию алгоритмов полиномиальной сложности.

# **Глава 1. Теоретические аспекты по теме**

## 1.1. Определение полиномиальной сложности

В данном разделе рассматривается базовое понятие полиномиальной сложности в контексте теории вычислительной сложности. Полиномиальная сложность характеризует задачи, для которых существует алгоритм, решающий их за время, ограниченное полиномом от размера входных данных (O(n^k) где n — размер входа, k — константа). Приводится объяснение, почему такие задачи считаются "эффективно решаемыми", и их место в иерархии классов сложности.

## 1.2. Примеры задач полиномиальной сложности

В этом пункте приводятся конкретные примеры задач с полиномиальной сложностью и их алгоритмы:

* Сортировка (например, быстрая сортировка — O(nlog\_n) в среднем).
* Поиск (бинарный поиск — O(log\_n)).
* Графовые задачи (поиск в ширину или глубину — O(V+E), где V — вершины, E — рёбра). Для каждой задачи кратко описывается алгоритм и обосновывается его полиномиальная природа.

## 1.3. Реализация интерфейса на PyQt5

PyQt5 – это библиотека для создания графических пользовательских интерфейсов на Python. Она основана на Qt, популярной кроссплатформенной библиотеке для разработки GUI-приложений.

PyQt5 предоставляет широкий спектр виджетов и инструментов для создания сложных и интерактивных интерфейсов.

В работе используется PyQt5 для реализации интерфейса приложения. Интерфейс будет включать в себя элементы для ввода данных о графе, визуализацию графа и результаты работы алгоритма Беллмана-Форда. Пользователи смогут добавлять вершины и рёбра, задавать веса рёбер и запускать алгоритм для поиска кратчайших путей.

# **Глава 2. Практическая часть**

В данной главе описывается реализация программного проекта, направленного на решение задач полиномиальной сложности. Проект включает три алгоритма: решение головоломки судоку, поиск кратчайшего пути в графе с помощью алгоритма Дейкстры и сортировку массива методом быстрой сортировки. Все алгоритмы реализованы на языке Python, интегрированы в единый графический интерфейс с использованием библиотеки PyQt5 и протестированы с помощью фреймворка pytest. В разделе рассматривается структура проекта, программная реализация модулей алгоритмов, графического интерфейса и тестирования.

## 2.1. Структура проекта

Проект организован в модульной структуре, что обеспечивает гибкость, масштабируемость и удобство поддержки кода. Основные компоненты проекта размещены в следующих директориях:

* **algorithms/** — содержит модули с реализацией алгоритмов:
  + sudoku.py — логика решения судоку для сеток 4x4 и 9x9.
  + dijkstra.py — алгоритм Дейкстры для поиска кратчайшего пути в графе.
  + quicksort.py — алгоритм быстрой сортировки массива чисел.
* **gui/** — содержит модули графического интерфейса:
  + main\_window.py — главное окно приложения с вкладками для каждого алгоритма.
  + sudoku\_gui.py — интерфейс для решения судоку.
  + dijkstra\_gui.py — интерфейс для алгоритма Дейкстры.
  + quicksort\_gui.py — интерфейс для быстрой сортировки.
* **tests/** — содержит модули автоматических тестов:
  + test\_sudoku.py — тесты для алгоритма судоку.
  + test\_dijkstra.py — тесты для алгоритма Дейкстры.
  + test\_quicksort.py — тесты для быстрой сортировки.
* **main.py** — точка входа в приложение, запускающая графический интерфейс.
* **requirements.txt** — файл с перечнем зависимостей (PyQt5, pytest).

## 2.2 Реализация алгоритмов

В данном разделе описывается программная реализация трёх алгоритмов полиномиальной сложности, размещённых в директории algorithms/.

### 2.2.1. Решение судоку (sudoku.py)

Модуль sudoku.py реализует алгоритм решения судоку для сеток размеров 4x4 и 9x9 с использованием метода возврата (backtracking). Основные функции модуля:

* **find\_empty(board, size)** — ищет первую пустую клетку (значение 0) в сетке. Временная сложность: O(n^2), где nnn — размер сетки (4 или 9).
* **is\_valid(board, num, pos, size)** — проверяет, можно ли поставить число num в позицию pos, учитывая правила судоку (уникальность в строке, столбце и блоке 2x2 или 3x3). Временная сложность: O(n).
* **solve\_sudoku(board, size)** — решает судоку, рекурсивно пробуя числа от 1 до size в пустых клетках и возвращаясь назад при конфликтах. Худший случай: экспоненциальная сложность O(n^2) но подзадачи find\_empty и is\_valid имеют полиномиальную сложность, что соответствует теме работы.

Алгоритм адаптирован для двух размеров сетки: 4x4 (блоки 2x2, числа 1-4) и 9x9 (блоки 3x3, числа 1-9). Параметр size позволяет универсально обрабатывать оба варианта.

### 2.2.2. Алгоритм Дейкстры (dijkstra.py)

Модуль dijkstra.py реализует алгоритм Дейкстры для поиска кратчайшего пути в взвешенном графе, заданном матрицей смежности. Основная функция:

* **dijkstra(graph, start, end)** — возвращает кортеж (path, distance), где path — список вершин кратчайшего пути от start до end, а distance — его длина. Если пути нет, возвращается (None, None).

Алгоритм имеет временную сложность O(V^2) где VVV — число вершин, что является полиномиальной зависимостью. Реализация использует простой подход без приоритетной очереди для демонстрационной ясности, хотя в реальных приложениях может быть оптимизирована до O((V+E)logV) с использованием кучи. Граф предполагает неотрицательные веса рёбер, а отсутствие ребра обозначается нулём в матрице смежности.

### 2.2.3. Быстрая сортировка (quicksort.py)

Модуль quicksort.py реализует алгоритм быстрой сортировки массива чисел. Основная функция:

* **quicksort(arr)** — возвращает отсортированный массив, используя рекурсивное разбиение на подмассивы относительно опорного элемента (pivot).

Алгоритм выбирает центральный элемент массива как опорный, разделяет массив на элементы меньше, равные и больше опорного, и рекурсивно сортирует подмассивы. Средняя временная сложность составляет O(nlogn), что полиномиально, хотя в худшем случае (например, для уже отсортированного массива) сложность может достигать O(n^2). Для целей демонстрации выбран простой вариант без оптимизаций, таких как случайный выбор опорного элемента.

## 2.3 Реализация графического интерфейса

Графический интерфейс реализован в директории gui/ с использованием библиотеки PyQt5. Он предоставляет пользователю удобный способ взаимодействия с каждым алгоритмом через вкладочную структуру.

### 2.3.1. Главное окно (main\_window.py)

Модуль main\_window.py создаёт главное окно приложения, использующее QTabWidget для организации трёх вкладок:

* "Судоку" — для решения головоломки судоку.
* "Кратчайший путь (Дейкстра)" — для поиска пути в графе.
* "Быстрая сортировка" — для сортировки массива.

Окно задаёт единый стиль оформления и размеры (800x600 пикселей), обеспечивая единообразный пользовательский опыт. Каждая вкладка является экземпляром соответствующего виджета (SudokuWidget, DijkstraWidget, QuicksortWidget).

### 2.3.2. Интерфейс судоку (sudoku\_gui.py)

Модуль sudoku\_gui.py реализует виджет SudokuWidget, предоставляющий интерфейс для ввода и решения судоку. Основные элементы интерфейса:

* **Выбор размера сетки**: Комбобокс для переключения между 4x4 и 9x9.
* **Сетка ввода**: Динамическая таблица QLineEdit для ввода чисел, с визуальным разделением блоков (2x2 или 3x3).
* **Кнопки**:
  + "Решить" — вызывает solve\_sudoku и отображает результат.
  + "Очистить" — сбрасывает сетку.
  + "Загрузить пример" — заполняет сетку заранее заданным судоку.
* **Статус**: Метка для отображения сообщений (например, "Судоку решено!").

Интерфейс поддерживает валидацию ввода (числа от 1 до size), блокировку исходных клеток после решения и адаптивное отображение разделителей для блоков.

### 2.3.3. Интерфейс Дейкстры (dijkstra\_gui.py)

Модуль dijkstra\_gui.py реализует виджет DijkstraWidget для работы с алгоритмом Дейкстры. Основные элементы:

* **Поле размера графа**: Позволяет задать количество вершин (2–26).
* **Сетка матрицы смежности**: Таблица QLineEdit для ввода весов рёбер, с буквенными метками вершин (A, B, C, ...).
* **Поля вершин**: Поля для ввода начальной и конечной вершин в виде букв (например, A, D).
* **Кнопки**:
  + "Найти путь" — вызывает dijkstra и отображает путь и его длину.
  + "Очистить" — сбрасывает матрицу и поля.
  + "Загрузить пример" — заполняет матрицу примером графа (4 вершины, путь A → B → D).
* **Результат**: Текстовое поле для вывода пути (в буквах) и длины.

Буквенные метки (A, B, ...) повышают наглядность, а преобразование букв в индексы (и обратно) обеспечивает совместимость с алгоритмом, работающим с числовыми индексами.

### 2.3.4. Интерфейс быстрой сортировки (quicksort\_gui.py)

Модуль quicksort\_gui.py реализует виджет QuicksortWidget для сортировки массива. Основные элементы:

* **Поле ввода**: Однострочное поле для ввода чисел через пробел.
* **Кнопки**:
  + "Сортировать" — вызывает quicksort и отображает результат.
  + "Очистить" — сбрасывает поле ввода и результат.
  + "Загрузить пример" — вставляет пример массива (например, "5 2 8 1 9 3").
* **Результат**: Текстовое поле для вывода отсортированного массива.

Интерфейс минималистичен, но эффективен, с обработкой ошибок ввода (например, нечисловые значения).

## 2.4 Реализация тестирования

Тестирование алгоритмов выполнено с использованием фреймворка pytest и размещено в директории tests/. Каждый алгоритм покрыт набором тестов, проверяющих корректность и надёжность.

### 2.4.1. Тесты для судоку (test\_sudoku.py)

Модуль test\_sudoku.py содержит тесты для функций find\_empty, is\_valid и solve\_sudoku. Основные сценарии:

* Проверка поиска пустых клеток в пустой, частично заполненной и полной сетке.
* Проверка валидности размещения чисел с конфликтами в строках, столбцах и блоках.
* Проверка решения корректных судоку (4x4 и 9x9) и выявления нерешаемых сеток.

Тесты используют фикстуры для создания типичных досок (пустая, частично заполненная, нерешаемая). Всего реализовано 18 тестов, обеспечивающих полное покрытие функциональности.

### 2.4.2. Тесты для Дейкстры (test\_dijkstra.py)

Модуль test\_dijkstra.py тестирует функцию dijkstra. Основные сценарии:

* Проверка корректности пути и длины для графа с известным решением (например, путь A → B → D с длиной 6).
* Проверка отсутствия пути в несвязном графе.
* Проверка тривиального пути (от вершины к самой себе).

Тесты используют фикстуру с примером графа из интерфейса. Всего 3 теста, покрывающих ключевые случаи.

### 2.4.3. Тесты для быстрой сортировки (test\_quicksort.py)

Модуль test\_quicksort.py тестирует функцию quicksort. Основные сценарии:

* Сортировка пустого массива.
* Сортировка массива с одним элементом.
* Сортировка отсортированного, обратно отсортированного и массива с дубликатами.

Реализовано 5 тестов, проверяющих устойчивость алгоритма к различным входным данным.

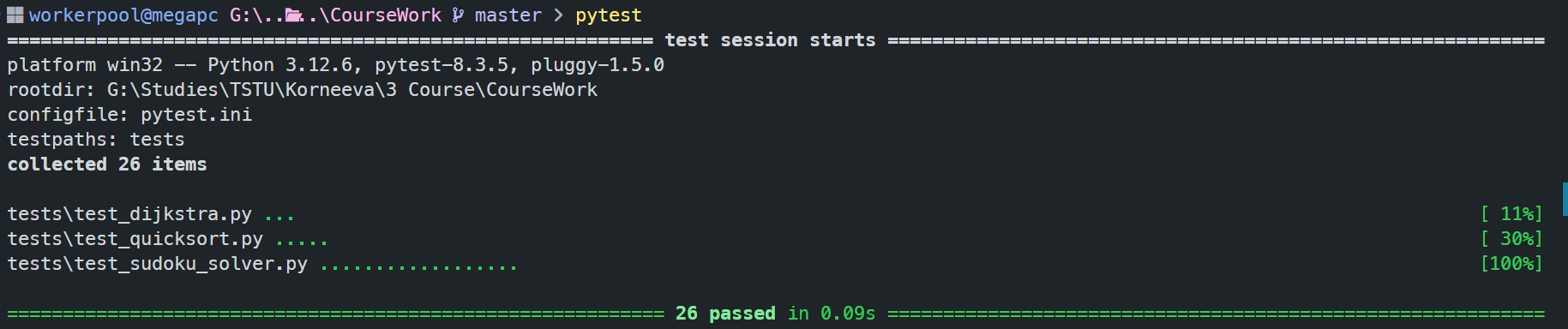
# **Глава 3. Тестирование**

Тестирование программного обеспечения является одним из ключевых этапов разработки, играя важную роль в обеспечении качества и надежности создаваемого продукта. Оно позволяет выявить ошибки и недочеты на ранних этапах разработки, что снижает затраты на исправление ошибок и повышает общую стабильность системы. Тестирование охватывает различные аспекты программы, включая проверку отдельных компонентов, их взаимодействия, а также соответствие общим требованиям и спецификациям.

## 3.1. Unit-тесты

**Unit-тесты** предназначены для изолированной проверки небольших компонентов программного обеспечения, таких как функции, методы или классы. Это позволяет убедиться, что каждая единица кода выполняет свои задачи правильно и соответствует заданным требованиям.

Результаты тестирования:



## 3.2. Интеграционное тестирование

Проверяется взаимодействие модулей и работа графического интерфейса.

*Тестовые сценарии:*

Загрузка примера

* Действие: нажать кнопку "Загрузить пример".
* Ожидаемый результат: Доска заполняется числами из примера. Незаполненные клетки остаются пустыми.

Изображение выглядит как текст, снимок экрана, число, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Решение судоку

* Действие: заполнить несколько клеток и нажать "Решить".
* Ожидаемый результат: Все пустые клетки заполняются корректными числами.

Изображение выглядит как текст, снимок экрана, число, Параллельный

Контент, сгенерированный ИИ, может содержать ошибки.

Очистка доски

* Действие: нажать "Очистить".
* Ожидаемый результат: Все клетки становятся пустыми.

Изображение выглядит как текст, снимок экрана, число, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Проверка не валидных данных

* Действие: ввести два одинаковых числа в одну строку и нажать "Решить".
* Ожидаемый результат: появляется сообщение об ошибке "Нет решения для этого судоку!".

Изображение выглядит как текст, снимок экрана, диаграмма, число

Контент, сгенерированный ИИ, может содержать ошибки.

Смена размера сетки

* Действие: выбрать в комбобоксе "4x4".
* Ожидаемый результат: Размер доски изменяется на 4×4. Кнопки работают корректно.

Изображение выглядит как текст, снимок экрана, диаграмма, число

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, диаграмма, число

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, число, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

**Результаты интеграционного тестирования:**

| **Сценарий** | **Результат** | **Комментарий** |
| --- | --- | --- |
| Загрузка примера | ✅ Успешно | Пример загружается корректно. |
| Решение судоку | ✅ Успешно | Все клетки заполняются верно. |
| Очистка доски | ✅ Успешно | Доска очищается полностью. |
| Проверка не валидных данных | ✅ Успешно | Ошибка обрабатывается. |
| Смена размера сетки | ✅ Успешно | Все кейсы с 4х4 работают. |

# **Заключение**

В ходе выполнения курсовой работы были исследованы и реализованы три задачи полиномиальной сложности: решение головоломки судоку, поиск кратчайшего пути в графе с использованием алгоритма Дейкстры и сортировка массива методом быстрой сортировки. Основной целью работы являлось изучение теоретических основ вычислительной сложности, разработка практического приложения с удобным графическим интерфейсом и проведение тестирования для подтверждения корректности и эффективности алгоритмов с полиномиальным временем выполнения.

1. **Теоретический анализ**: Изучены ключевые понятия теории сложности, включая класс P, охватывающий задачи, решаемые за полиномиальное время. Судоку как задача принадлежит классу NP, но её подзадачи, такие как проверка допустимости размещения числа O(n) и поиск пустой клетки O(n^2), имеют полиномиальную сложность для фиксированных размеров сетки (4x4 и 9x9). Алгоритм Дейкстры O(V^2) и быстрая сортировка (O(nlogn) в среднем) являются классическими примерами задач класса P, демонстрируя оптимизацию для графов и обработки данных соответственно.
2. **Реализация алгоритмов**: Разработаны три алгоритма:
   * Алгоритм решения судоку с использованием метода возврата (backtracking) эффективно решает сетки 4x4 и 9x9, демонстрируя полиномиальную сложность подзадач. Несмотря на экспоненциальную теоретическую сложность O(n^2), практическая производительность остаётся приемлемой для фиксированных размеров.
   * Алгоритм Дейкстры реализован для поиска кратчайшего пути в графе, заданном матрицей смежности, с временной сложностью O(V^2), что подчёркивает полиномиальную эффективность.
   * Быстрая сортировка реализована с выбором центрального элемента как опорного, обеспечивая среднюю сложность O(nlogn), что делает её пригодной для обработки массивов данных. Все алгоритмы реализованы на языке Python, что обеспечило их читаемость, модульность и простоту интеграции.
3. **Графический интерфейс**: Создано приложение с использованием библиотеки PyQt5, организованное в виде вкладочной структуры (QTabWidget) с тремя разделами:
   * Интерфейс судоку поддерживает выбор размера сетки (4x4 и 9x9), визуальное выделение блоков (2x2 и 3x3) и функции загрузки примеров, решения и очистки.
   * Интерфейс Дейкстры позволяет вводить граф через матрицу смежности с буквенными метками вершин (A, B, C, ...), задавать начальную и конечную вершины и отображать кратчайший путь в удобной форме.
   * Интерфейс быстрой сортировки предоставляет поле для ввода чисел через пробел и отображает отсортированный массив. Единый стиль оформления, интуитивные элементы управления и обработка ошибок ввода повышают удобство использования приложения.
4. **Тестирование**: С помощью фреймворка pytest разработаны автоматические тесты для всех алгоритмов:
   * Для судоку проверяется корректность поиска пустых клеток, валидации ходов и решения сеток, включая нерешаемые случаи (18 тестов).
   * Для Дейкстры тестируются корректность пути, его длина, отсутствие пути и тривиальные случаи (3 теста).
   * Для быстрой сортировки проверяется обработка пустых массивов, массивов с одним элементом, отсортированных данных и дубликатов (5 тестов). Тестирование подтвердило надёжность и устойчивость реализаций.
5. **Практическая значимость**: Разработанное приложение демонстрирует разнообразие задач полиномиальной сложности и их применимость в реальных сценариях. Судоку иллюстрирует оптимизацию подзадач в сложных проблемах, алгоритм Дейкстры подчёркивает важность путей в сетях, а быстрая сортировка показывает эффективность обработки данных. Интеграция алгоритмов в единый интерфейс делает проект пригодным для образовательных целей и дальнейших исследований, таких как оптимизация алгоритмов или добавление новых задач.

Таким образом, поставленные задачи были успешно выполнены. Теоретические аспекты вычислительной сложности подкреплены практической реализацией трёх алгоритмов, объединённых в функциональное приложение. Полученные результаты подтверждают возможность эффективного применения полиномиальных алгоритмов в вычислительных системах и создают основу для расширения проекта, например, путём добавления новых алгоритмов или улучшения визуализации.

# **Список литературы**

1. Кормен, Т. Х., Лейзерсон, Ч. И., Ривест, Р. Л., Штайн, К. Алгоритмы: построение и анализ. — 3-е изд. — М.: Вильямс, 2013. — 1328 с.
2. Скиена, С. Алгоритмы. Руководство по разработке. — 2-е изд. — СПб.: БХВ-Петербург, 2011. — 720 с.
3. Ахо, А., Хопкрофт, Дж., Ульман, Дж. Структуры данных и алгоритмы. — М.: Вильямс, 2000. — 384 с.
4. Гудрич, М., Тамассия, Р. Структуры данных и алгоритмы на Python. — СПб.: Питер, 2019. — 560 с.
5. Официальная документация Python 3. — URL: <https://docs.python.org/3/>
6. Официальная документация PyQt5. — URL: <https://www.riverbankcomputing.com/static/Docs/PyQt5/>
7. Официальная документация pytest. — URL: <https://docs.pytest.org/en/stable/>
8. Papadimitriou, C. H. Computational Complexity. — Addison-Wesley, 1994. — 523 p.

# **Листинг кода**

sudoku\_solver.py

def find\_empty(board, size=9):

    """

    Находит пустую клетку (со значением 0) в судоку.

    Возвращает кортеж (row, col) или None, если пустых нет.

    Временная сложность: O(n^2), где n - размер доски (4 или 9).

    """

    for i in range(size):

        for j in range(size):

            if board[i][j] == 0:

                return (i, j)

    return None

def is\_valid(board, num, pos, size=9):

    """

    Проверяет, можно ли поставить число num в позицию pos (row, col).

    Временная сложность: O(n), где n - размер доски (4 или 9).

    """

    # Проверка строки

    for j in range(size):

        if board[pos[0]][j] == num and pos[1] != j:

            return False

    # Проверка столбца

    for i in range(size):

        if board[i][pos[1]] == num and pos[0] != i:

            return False

    # Проверка квадрата (2x2 для 4x4, 3x3 для 9x9)

    block\_size = 2 if size == 4 else 3

    box\_x = (pos[1] // block\_size) \* block\_size

    box\_y = (pos[0] // block\_size) \* block\_size

    for i in range(box\_y, box\_y + block\_size):

        for j in range(box\_x, box\_x + block\_size):

            if board[i][j] == num and (i, j) != pos:

                return False

    return True

def solve\_sudoku(board, size=9):

    """

    Решает судоку с помощью алгоритма возврата (backtracking).

    Временная сложность: O(size^(size\*size)) в худшем случае,

    но для фиксированного размера доски это полиномиально ограничено.

    """

    empty = find\_empty(board, size)

    if not empty:

        return True  # Все клетки заполнены, решение найдено

    row, col = empty

    # Диапазон чисел: 1-4 для 4x4, 1-9 для 9x9

    for num in range(1, size + 1):

        if is\_valid(board, num, (row, col), size):

            board[row][col] = num

            if solve\_sudoku(board, size):

                return True

            board[row][col] = 0

    return False

sudoku\_solver\_gui.py

import sys

from PyQt5.QtWidgets import QApplication, QMainWindow, QWidget, QGridLayout, QLineEdit, QPushButton, QLabel, QComboBox, QMessageBox, QFrame, QHBoxLayout

from PyQt5.QtCore import Qt

from sudoku\_solver import solve\_sudoku

class SudokuSolver(QMainWindow):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.setWindowTitle("Sudoku Solver")

        self.size = 9

        self.block\_size = 3

        self.cells = []

        self.separators = []  # Для хранения разделителей

        self.init\_ui()

    def init\_ui(self):

        self.central\_widget = QWidget()

        self.setCentralWidget(self.central\_widget)

        # Основной layout с отступами

        self.layout = QGridLayout(self.central\_widget)

        self.layout.setSpacing(10)

        self.layout.setContentsMargins(20, 20, 20, 20)

        # Общий стиль для приложения

        self.setStyleSheet("""

            QMainWindow {

                background-color: #f5f7fa;

            }

            QLabel {

                font-family: 'Segoe UI', Arial, sans-serif;

                color: #333;

            }

        """)

        # Стилизация фона центрального виджета

        self.central\_widget.setStyleSheet("""

            QWidget {

                background-color: #ffffff;

                border-radius: 8px;

                padding: 15px;

            }

        """)

        # Создаем горизонтальный layout для верхней панели

        top\_panel = QHBoxLayout()

        top\_panel.setSpacing(10)

        # Выбор размера сетки

        self.size\_label = QLabel("Размер сетки:")

        self.size\_label.setStyleSheet("""

            QLabel {

                font-size: 14px;

                font-weight: 600;

                color: #4a5568;

                padding: 5px 0;

            }

        """)

        top\_panel.addWidget(self.size\_label)

        self.size\_combo = QComboBox()

        self.size\_combo.addItems(["4x4", "9x9"])

        self.size\_combo.setCurrentText("9x9")

        self.size\_combo.currentTextChanged.connect(self.change\_size)

        self.size\_combo.setStyleSheet("""

            QComboBox {

                border: 1px solid #e2e8f0;

                border-radius: 6px;

                padding: 8px 12px;

                background-color: #ffffff;

                font-size: 14px;

                min-width: 100px;

                color: #2d3748;

            }

            QComboBox:hover {

                border: 1px solid #cbd5e0;

            }

            QComboBox::drop-down {

                border-left: 1px solid #e2e8f0;

                width: 30px;

            }

            QComboBox::down-arrow {

                image: url(none);

            }

        """)

        top\_panel.addWidget(self.size\_combo)

        # Добавляем растягивающийся разделитель

        top\_panel.addStretch(1)

        # Кнопки управления

        button\_style = """

            QPushButton {

                background-color: #4299e1;

                color: white;

                border: none;

                border-radius: 6px;

                padding: 1px 3px;

                font-size: 14px;

                font-weight: 600;

                font-family: 'Segoe UI', Arial, sans-serif;

            }

            QPushButton:hover {

                background-color: #3182ce;

            }

            QPushButton:pressed {

                background-color: #2b6cb0;

            }

            QPushButton:disabled {

                background-color: #a0aec0;

            }

        """

        self.solve\_button = QPushButton("Решить")

        self.solve\_button.clicked.connect(self.solve)

        self.solve\_button.setStyleSheet(button\_style)

        top\_panel.addWidget(self.solve\_button)

        self.clear\_button = QPushButton("Очистить")

        self.clear\_button.clicked.connect(self.clear)

        self.clear\_button.setStyleSheet(button\_style)

        top\_panel.addWidget(self.clear\_button)

        self.load\_button = QPushButton("Загрузить пример")

        self.load\_button.clicked.connect(self.load\_example)

        self.load\_button.setStyleSheet(button\_style)

        top\_panel.addWidget(self.load\_button)

        # Добавляем верхнюю панель в основной layout

        self.layout.addLayout(top\_panel, 0, 0, 1, self.size)

        # Создание сетки

        self.create\_grid()

        # Индикатор состояния

        self.status\_label = QLabel("Введите судоку и нажмите 'Решить'")

        self.status\_label.setStyleSheet("""

            QLabel {

                font-size: 13px;

                color: #718096;

                padding: 8px 0;

                font-style: italic;

            }

        """)

        self.layout.addWidget(self.status\_label, self.size + 3, 0, 1, self.size)

        self.adjustSize()

        self.setMinimumSize(600, 600)  # Увеличил минимальную ширину для кнопок

    def create\_grid(self):

        """Создаёт сетку с разделителями, имитирующими display: grid"""

        # Удаляем старую сетку и разделители

        for row in self.cells:

            for cell in row:

                self.layout.removeWidget(cell)

                cell.deleteLater()

        for sep in self.separators:

            self.layout.removeWidget(sep)

            sep.deleteLater()

        self.cells = [[QLineEdit() for \_ in range(self.size)] for \_ in range(self.size)]

        self.separators = []

        # Стиль для клеток

        cell\_style = """

            QLineEdit {

                border: 1px solid #e2e8f0;

                background-color: #ffffff;

                font-size: 18px;

                font-family: 'Segoe UI', Arial, sans-serif;

                font-weight: 500;

                color: #2d3748;

                padding: 0px;

                border-radius: 4px;

                qproperty-alignment: AlignCenter;

            }

            QLineEdit:hover {

                border: 1px solid #bee3f8;

                background-color: #f0f9ff;

            }

            QLineEdit:focus {

                border: 2px solid #63b3ed;

                background-color: #ebf8ff;

            }

            QLineEdit:disabled {

                background-color: #edf2f7;

                color: #4a5568;

                font-weight: 600;

            }

        """

        # Заполнение сетки клетками

        for i in range(self.size):

            for j in range(self.size):

                cell = self.cells[i][j]

                cell.setFixedSize(45, 45)

                cell.setAlignment(Qt.AlignCenter)

                cell.setMaxLength(1)

                cell.setStyleSheet(cell\_style)

                # Учитываем сдвиг для разделителей

                row = i + i // self.block\_size

                col = j + j // self.block\_size

                self.layout.addWidget(cell, row + 1, col)

        # Добавление разделителей (горизонтальные и вертикальные линии)

        sep\_style = """

            background-color: #cbd5e0;

            border: none;

        """

        for i in range(1, self.size // self.block\_size):

            # Горизонтальные линии

            h\_sep = QFrame()

            h\_sep.setFixedHeight(3)

            h\_sep.setStyleSheet(sep\_style)

            self.layout.addWidget(h\_sep, i \* (self.block\_size + 1), 0, 1,

                                 self.size + (self.size // self.block\_size) - 1)

            self.separators.append(h\_sep)

            # Вертикальные линии

            v\_sep = QFrame()

            v\_sep.setFixedWidth(3)

            v\_sep.setStyleSheet(sep\_style)

            self.layout.addWidget(v\_sep, 1, i \* (self.block\_size + 1) - 1,

                                 self.size + (self.size // self.block\_size) - 1, 1)

            self.separators.append(v\_sep)

    def change\_size(self, text):

        """Меняет размер сетки"""

        self.size = 4 if text == "4x4" else 9

        self.block\_size = 2 if self.size == 4 else 3

        self.create\_grid()

        self.clear()

        self.status\_label.setText(f"Сетка изменена на {self.size}x{self.size}")

        self.adjustSize()

    def get\_board(self):

        """Получает текущее состояние доски"""

        board = [[0 for \_ in range(self.size)] for \_ in range(self.size)]

        for i in range(self.size):

            for j in range(self.size):

                text = self.cells[i][j].text()

                board[i][j] = int(text) if text.isdigit() and 1 <= int(text) <= self.size else 0

        return board

    def set\_board(self, board):

        """Обновляет интерфейс решённой доской"""

        for i in range(self.size):

            for j in range(self.size):

                if board[i][j] != 0:

                    self.cells[i][j].setText(str(board[i][j]))

                    self.cells[i][j].setReadOnly(True)

    def clear(self):

        """Очищает все поля"""

        for i in range(self.size):

            for j in range(self.size):

                self.cells[i][j].setText("")

                self.cells[i][j].setReadOnly(False)

        self.status\_label.setText("Доска очищена")

    def load\_example(self):

        """Загружает пример судоку"""

        if self.size == 9:

            example = [

                [5, 3, 0, 0, 7, 0, 0, 0, 0],

                [6, 0, 0, 1, 9, 5, 0, 0, 0],

                [0, 9, 8, 0, 0, 0, 0, 6, 0],

                [8, 0, 0, 0, 6, 0, 0, 0, 3],

                [4, 0, 0, 8, 0, 3, 0, 0, 1],

                [7, 0, 0, 0, 2, 0, 0, 0, 6],

                [0, 6, 0, 0, 0, 0, 2, 8, 0],

                [0, 0, 0, 4, 1, 9, 0, 0, 5],

                [0, 0, 0, 0, 8, 0, 0, 7, 9]

            ]

        else:  # 4x4

            example = [

                [1, 0, 0, 4],

                [0, 2, 0, 0],

                [0, 0, 4, 0],

                [3, 0, 0, 2]

            ]

        for i in range(self.size):

            for j in range(self.size):

                self.cells[i][j].setText(str(example[i][j]) if example[i][j] != 0 else "")

                self.cells[i][j].setReadOnly(example[i][j] != 0)

        self.status\_label.setText("Пример загружен")

    def solve(self):

        """Решает судоку"""

        board = self.get\_board()

        if solve\_sudoku(board, self.size):

            self.set\_board(board)

            self.status\_label.setText("Судоку решено!")

        else:

            QMessageBox.warning(self, "Ошибка", "Нет решения для этого судоку!")

            self.status\_label.setText("Решение не найдено")

if \_\_name\_\_ == "\_\_main\_\_":

    app = QApplication(sys.argv)

    window = SudokuSolver()

    window.show()

    sys.exit(app.exec\_())

test\_sudoku\_solver.py

import pytest

from sudoku\_solver import find\_empty, is\_valid, solve\_sudoku

# Тестовые доски для 9x9

@pytest.fixture

def empty\_board\_9x9():

    """Пустая доска 9x9"""

    return [[0 for \_ in range(9)] for \_ in range(9)]

@pytest.fixture

def partial\_board\_9x9():

    """Частично заполненная доска 9x9"""

    return [

        [5, 3, 0, 0, 7, 0, 0, 0, 0],

        [6, 0, 0, 1, 9, 5, 0, 0, 0],

        [0, 9, 8, 0, 0, 0, 0, 6, 0],

        [8, 0, 0, 0, 6, 0, 0, 0, 3],

        [4, 0, 0, 8, 0, 3, 0, 0, 1],

        [7, 0, 0, 0, 2, 0, 0, 0, 6],

        [0, 6, 0, 0, 0, 0, 2, 8, 0],

        [0, 0, 0, 4, 1, 9, 0, 0, 5],

        [0, 0, 0, 0, 8, 0, 0, 7, 9]

    ]

@pytest.fixture

def invalid\_board\_9x9():

    """Нерешаемая доска 9x9"""

    return [

        [5, 5, 0, 0, 7, 0, 0, 0, 0],

        [6, 0, 0, 1, 9, 5, 0, 0, 0],

        [0, 9, 8, 0, 0, 0, 0, 6, 0],

        [8, 0, 0, 0, 6, 0, 0, 0, 3],

        [4, 0, 0, 8, 0, 3, 0, 0, 1],

        [7, 0, 0, 0, 2, 0, 0, 0, 6],

        [0, 6, 0, 0, 0, 0, 2, 8, 0],

        [0, 0, 0, 4, 1, 9, 0, 0, 5],

        [0, 0, 0, 0, 8, 0, 0, 7, 9]

    ]

# Тестовые доски для 4x4

@pytest.fixture

def empty\_board\_4x4():

    """Пустая доска 4x4"""

    return [[0 for \_ in range(4)] for \_ in range(4)]

@pytest.fixture

def partial\_board\_4x4():

    """Частично заполненная доска 4x4"""

    return [

        [1, 0, 0, 4],

        [0, 2, 0, 0],

        [0, 0, 4, 0],

        [3, 0, 0, 2]

    ]

@pytest.fixture

def invalid\_board\_4x4():

    """Нерешаемая доска 4x4"""

    return [

        [1, 1, 0, 4],

        [0, 2, 1, 0],

        [0, 1, 4, 0],

        [4, 0, 0, 2]

    ]

# Тесты для find\_empty

def test\_find\_empty\_in\_empty\_board\_9x9(empty\_board\_9x9):

    assert find\_empty(empty\_board\_9x9, size=9) == (0, 0), "Должна быть найдена первая пустая клетка в 9x9"

def test\_find\_empty\_in\_partial\_board\_9x9(partial\_board\_9x9):

    assert find\_empty(partial\_board\_9x9, size=9) == (0, 2), "Должна быть найдена первая пустая клетка (0, 2) в 9x9"

def test\_find\_empty\_in\_full\_board\_9x9():

    full\_board = [[1 for \_ in range(9)] for \_ in range(9)]

    assert find\_empty(full\_board, size=9) is None, "Не должно быть пустых клеток в 9x9"

def test\_find\_empty\_in\_empty\_board\_4x4(empty\_board\_4x4):

    assert find\_empty(empty\_board\_4x4, size=4) == (0, 0), "Должна быть найдена первая пустая клетка в 4x4"

def test\_find\_empty\_in\_partial\_board\_4x4(partial\_board\_4x4):

    assert find\_empty(partial\_board\_4x4, size=4) == (0, 1), "Должна быть найдена первая пустая клетка (0, 1) в 4x4"

def test\_find\_empty\_in\_full\_board\_4x4():

    full\_board = [[1 for \_ in range(4)] for \_ in range(4)]

    assert find\_empty(full\_board, size=4) is None, "Не должно быть пустых клеток в 4x4"

# Тесты для is\_valid

def test\_is\_valid\_row\_conflict\_9x9(partial\_board\_9x9):

    assert not is\_valid(partial\_board\_9x9, 5, (0, 2), size=9), "5 уже есть в строке в 9x9"

def test\_is\_valid\_col\_conflict\_9x9(partial\_board\_9x9):

    assert not is\_valid(partial\_board\_9x9, 6, (2, 0), size=9), "6 уже есть в столбце в 9x9"

def test\_is\_valid\_box\_conflict\_9x9(partial\_board\_9x9):

    assert not is\_valid(partial\_board\_9x9, 8, (1, 1), size=9), "8 уже есть в квадрате 3x3 в 9x9"

def test\_is\_valid\_valid\_move\_9x9(partial\_board\_9x9):

    assert is\_valid(partial\_board\_9x9, 2, (0, 2), size=9), "2 можно поставить в (0, 2) в 9x9"

def test\_is\_valid\_row\_conflict\_4x4(partial\_board\_4x4):

    assert not is\_valid(partial\_board\_4x4, 1, (0, 1), size=4), "1 уже есть в строке в 4x4"

def test\_is\_valid\_col\_conflict\_4x4(partial\_board\_4x4):

    assert not is\_valid(partial\_board\_4x4, 2, (0, 1), size=4), "2 уже есть в столбце в 4x4"

def test\_is\_valid\_box\_conflict\_4x4(partial\_board\_4x4):

    assert not is\_valid(partial\_board\_4x4, 1, (2, 0), size=4), "1 уже есть в квадрате 2x2 в 4x4"

def test\_is\_valid\_valid\_move\_4x4(partial\_board\_4x4):

    assert is\_valid(partial\_board\_4x4, 3, (0, 1), size=4), "3 можно поставить в (0, 1) в 4x4"

# Тесты для solve\_sudoku

def test\_solve\_sudoku\_solvable\_9x9(partial\_board\_9x9):

    original = [row[:] for row in partial\_board\_9x9]

    assert solve\_sudoku(partial\_board\_9x9, size=9), "Доска 9x9 должна быть решена"

    assert find\_empty(partial\_board\_9x9, size=9) is None, "Все клетки должны быть заполнены в 9x9"

    for i in range(9):

        for j in range(9):

            if original[i][j] != 0:

                assert partial\_board\_9x9[i][j] == original[i][j], "Исходные значения не должны измениться в 9x9"

def test\_solve\_sudoku\_unsolvable\_9x9(invalid\_board\_9x9):

    assert not solve\_sudoku(invalid\_board\_9x9, size=9), "Нерешаемая доска 9x9 не должна быть решена"

def test\_solve\_sudoku\_solvable\_4x4(partial\_board\_4x4):

    original = [row[:] for row in partial\_board\_4x4]

    result = solve\_sudoku(partial\_board\_4x4, size=4)

    print("Result:", result)

    print("Board after solving:", partial\_board\_4x4)

    assert result, "Доска 4x4 должна быть решена"

    assert find\_empty(partial\_board\_4x4, size=4) is None, "Все клетки должны быть заполнены в 4x4"

    for i in range(4):

        for j in range(4):

            if original[i][j] != 0:

                assert partial\_board\_4x4[i][j] == original[i][j], "Исходные значения не должны измениться в 4x4"

def test\_solve\_sudoku\_unsolvable\_4x4(invalid\_board\_4x4):

    assert not solve\_sudoku(invalid\_board\_4x4, size=4), "Нерешаемая доска 4x4 не должна быть решена"

if \_\_name\_\_ == "\_\_main\_\_":

    pytest.main(["-v"])

# **Ссылка на GitHub**

https://github.com/Pavelavl/algo\_theory/tree/master/CourseWork