Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

«Поиск абелевых строк наибольшей длины»

Автор: Збань Илья Константинович _____
Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика
Квалификация: Бакалавр
Руководитель: Аксенов В.Е., магистр _____

К защите допустить
Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. _____

«__» _____ 20__ г.

Санкт-Петербург, 2017 г.

Студент Збань И.К. **Группа** М3439 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Математические модели и алгоритмы разработки программного обеспечения

Квалификационная работа выполнена с оценкой _____

Дата защиты «20» июня 2015 г.

Секретарь ГЭК *Павлова О.Н.* Принято: «___» _____ 20___ г.

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

УТВЕРЖДАЮ
Зав. каф. компьютерных технологий
докт. техн. наук, проф.
_____ Васильев В.Н.
«__» _____ 20__ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Студент Збань И.К. **Группа** М3439 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования **Руководитель** Аксенов Виталий Евгеньевич, магистр, аспирант Университета ИТМО/INRIA PARIS

1 Наименование темы: Поиск абелевых строк наибольшей длины

Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика

Направленность (профиль): Математические модели и алгоритмы разработки программного обеспечения

Квалификация: Бакалавр

2 Срок сдачи студентом законченной работы: «31» мая 2017 г.

3 Техническое задание и исходные данные к работе.

Улучшить существующие алгоритмы поиска наибольшей Абелевой подстроки, проанализировать существующие решения

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

Тестирование на практике оптимального на данный момент алгоритма поиска абелевых подквадратов и работа над алгоритмами поиска НОАП

5 Перечень графического материала (с указанием обязательного материала)

Не предусмотрено

6 Исходные материалы и пособия

Опубликованные за последние годы публикации об абелевых строках

7 Календарный план

№№ пп.	Наименование этапов выпускной квалификационной работы	Срок выполнения этапов работы	Отметка о выполнении, подпись руков.
1	Изучение предметной области	09.2016	
2	Проверка известных теоретических методов на компьютере	11.2016	
3	Постановка конкретных задач	12.2016	
4	Разработка алгоритмов для решения поставленных задач	03.2017	
5	Написать пояснительную записку	05.2017	

8 Дата выдачи задания: «01» сентября 2016 г.

Руководитель _____

Задание принял к исполнению _____ «01» сентября 2016 г.

Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Студент: Збань Илья Константинович
Наименование темы работы: Поиск абелевых строк наибольшей длины
Наименование организации, где выполнена работа: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Получить новый алгоритм поиска НОАП, улучшающий существующие результаты
2 Задачи, решаемые в работе: Тестирование существующих алгоритмов, теоретическая и практическая оценка матожидания НОАП случайных строк, решение задачи о поиске НОАП в общем случае
3 Число источников, использованных при составлении обзора: _____
4 Полное число источников, использованных в работе: 9
5 В том числе источников по годам

Отечественных			Иностранных		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет

6 Использование информационных ресурсов Internet: _____
7 Использование современных пакетов компьютерных программ и технологий: c++, python, gnuplot, git, etc
8 Краткая характеристика полученных результатов: Главным итогом работы является новый алгоритм поиска НОАП, на данный момент являющийся оптимальным по затратам времени и памяти
9 Гранты, полученные при выполнении работы: Грантов при выполнении работы получено не было
10 Наличие публикаций и выступлений на конференциях по теме работы: Публикаций и выступлений на конференциях по теме выпускной работы не было

Выпусник: Збань И.К. _____

Руководитель: Аксенов В.Е. _____

«___» _____ 20___ г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В последнее время тема Абелевой эквивалентности строк стала активно обсуждаться в научном сообществе. Первые определения, такие как вектор числа встреч каждого символа в подстроке, называемый вектором Парея, и первые сформулированные задачи на эту тему, такие как нахождение подстроки с заданным вектором Парея, называемой *jumbled indexing*, были предложены ещё в 60-х годах прошлого века. Но только около двадцати лет назад стали появляться первые результаты в этой области. С тех пор результатов с каждым годом становилось всё больше, и, в конце концов, Абелева эквивалентность выделилась в самостоятельную подобласть.

Задачи, связанные с Абелевой эквивалентностью, встречаются в большом числе различных областей, связанных с информатикой. Например, представленная выше задача *jumbled indexing* находит применение в бионформатике при решении задач *mass spectrometry* и *gene clusters*. Кроме того, Абелево совпадение слов, иначе, совпадение векторов Парея, может применяться как критерий эвристического фильтра для поиска шаблона в тексте, а также поиска шаблона с ошибками.

В главе 1 мы вводим основные определения, используемые в работе, ставим решаемые задачи и представляем известные на сегодняшний день результаты.

В главе 2 мы предлагаем новые алгоритмы для решения задачи о поиске числа Абелевых подквадратов и задачи о нахождении наибольшей общей Абелевой подстроки. В этой же главе мы приводим теоретические оценки к поставленным задачам.

В главе 3 мы приводим экспериментальные результаты алгоритмов, предложенных в главе 2.

ГЛАВА 1. ОБЗОР

1.1. Используемые определения

Введем набор определений, которые будут использоваться по ходу работы.

Определение 1. Алфавит $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ — конечное множество символов. Число символов алфавита $|\Sigma| = \sigma$.

Определение 2. Обозначим за $|s|_c$ количество символов c в строке s.

Определение 3. Вектор Парея $\mathcal{P}(s) = (|s|_{c_1}, |s|_{c_2}, \ldots, |s|_{c_\sigma})$ — вектор частот символов строки s.

Определение 4. Две строки a и b Абелево эквивалентны, если существует перестановка π, переводящая одну из строк в другую, $\pi(a) = b$.

Определение 5. Строка s является Абелевым квадратом, если существуют две Абелево эквивалентные строки $a \equiv b$, что $s = ab$.

Определение 6. Абелевым подквадратом строки s назовем ее подстроку строки s, являющуюся Абелевым квадратом.

Определение 7. Событие является `w.h.p.` — *with high probability*, если вероятность такого исхода зависит от какого-то параметра n, и эта вероятность стремится к единицы при стремлении n к бесконечности.

Определение 8. Будем говорить, что алгоритм работает за $\langle \mathcal{O}(f(n)), \mathcal{O}(g(n)) \rangle$, если он работает, используя $\mathcal{O}(f(n))$ времени и $\mathcal{O}(g(n))$ памяти.

Определение 9. Строка t имеет Абелево вхождение в строку s, если у строки s есть подстрока x, Абелево эквивалентная t.

Определение 10. Хешмап — ассоциативный массив, обрабатывающий запросы *get* и *put* за $\mathcal{O}(1)$.

Далее сформулируем некоторые задачи, которые используются в работе.

Задача 1. $3SUM^+$. Дано три множества кортежей целых чисел A, B, C. Нужно найти три элемента $a \in A, b \in B, c \in C$ такие, что $a + b = c$.

Задача 2. Поиск НОАП. Даны две строки a, b над алфавитом Σ. Нужно найти подстроку x строки a и подстроку y строки b, что $x \equiv y$, а длина подстрок максимальна.

Задача 3. Подсчет числа Абелевых подквадратов. Дана строка $s = s_0 s_1 \dots s_{n-1}$ над алфавитом мощности 2, $\Sigma = \{c_0, c_1\}$. Нужно найти количество различных ее подстрок $s_{i \dots j}$, являющихся Абелевыми квадратами.

Задача 4. *Jumbled indexing*, *Histogram indexing*. Задача проверки того, содержит ли данный текст T подстроку, Абелево эквивалентную заданному шаблону S.

1.2. Предыдущие результаты

1.2.1. Наибольшая общая Абелева подстрока

Постановка задачи о поиске наибольшей общей Абелевой подстроке схожа с задачей поиска наидлиннейшей общей подстроки — очень важной задачи, исследовавшейся в 70-е года прошлого века. В частности, суффиксное дерево и линейный алгоритм его построения были разработаны в процессе работы над несколькими задачами, одной из которых являлся линейный поиск наидлиннейшей общей подстроки.

Задача поиска наибольшей общей Абелевой подстроки ($LCAS$) была сформулирована в 2013 году на семинаре StringMasters. Там она была поставлена как открытая задача.

После этого в 2015 году A.Attabi et al [1] предложили два алгоритма решения этой задачи.

Первый алгоритм решает задачу поиска НОАП за $\langle \mathcal{O}(n^2\sigma), \mathcal{O}(n\sigma) \rangle$. В этом алгоритме авторы предлагают для каждой длины посчитать для обеих строк векторы Парея всех подстрок такой длины за $\langle \mathcal{O}(n\sigma), \mathcal{O}(n\sigma) \rangle$, отсортировать их сортировкой подсчетом, и после этого двумя указателями проверить, есть ли две Абелево эквивалентные подстроки.

Второй алгоритм решает задачу поиска НОАП для строк на бинарном алфавите. Он использует два факта: что для фиксированной длины подстроки достаточно сравнивать лишь количество одного из символов c_1, и что если у строки есть подстрока длины l содержащая x символов c_1, и подстрока длины l содержащая y символов c_1, то найдется и подстрока содержащая z символов c_1 для любого $x \leqslant z \leqslant y$. Используя два этих факта, в алгоритме для обеих строк считается максимальное и минимальное количество символов c_1, которые могут быть у подстроки длины l для каждой длины. Если эти отрезки для длины l у строк пересекаются, то l — кандидат на НОАП. На алгоритм подсчета этих отрезков за $\mathcal{O}(n^2/\log n)$ авторы ссылаются как на

уже известный. Один из вариантов это сделать — применить метод четырех русских, предподсчитав матрицу размера $\frac{\log n}{2}$ на $\frac{\log n}{2}$, в которой содержится изменение количества единиц и максимальное число единиц, полученное на префиксе, если отрезать такие первые $\frac{\log n}{2}$ символов и дописать такие последние $\frac{\log n}{2}$ символов. Прыжков получится $2 \cdot n / \log n$, что и дает искомую асимптотику в $n^2 / \log n$.

В 2016 году на конференции SPIRE S.Grabowski et al [2] улучшили алгоритм из [1] 2015 года, уменьшив требование памяти до $\mathcal{O}(n)$, и предложили алгоритм, решающий задачу для случая алфавитов большого размера, работающий за $\langle \mathcal{O}(n^2 \log^2 n \log^* n), \mathcal{O}(n \log^2 n) \rangle$ времени и памяти.

Первый результат, улучшение памяти алгоритма A.Attabi et al основано на двух оптимизациях. Во-первых, во время сортировки не хранятся все векторы Парея, а на каждой из σ итераций количество вхождений очередного символа вычисляется заново. Во-вторых, для того, чтобы все еще можно было сравнивать векторы Парея, явно хранятся векторы Парея для всех позиций, кратных σ, т.е. для подстрок $s_0 \dots s_{l-1}$, $s_\sigma \dots s_{\sigma+l-1}$, и так далее. Для сравнения векторов Парея двух произвольных подстрок, нужно взять ближайший посчитанный вектор Парея и изменить в нем $\mathcal{O}(\sigma)$ элементов, таким образом сравнение все еще работает за $\mathcal{O}(\sigma)$.

Второй алгоритм использует технику для поддержания изменяющегося множества строк с операциями split, join и equality testing, опубликованной в [3].

Сравнительное время работы этих алгоритмов можно увидеть в таблице 1.

Таблица 1 – Существующие детерминированные алгоритмы поиска НОАП

Год	Авторы	Время	Память
2015	A. Alattabi et al	$\mathcal{O}(n^2\sigma)$	$\mathcal{O}(n\sigma)$
2016	S. Grabowski et al	$\mathcal{O}(n^2\sigma)$	$\mathcal{O}(n)$
2016	S. Grabowski et al	$\mathcal{O}(n^2 \log^2 n \log^* n)$	$\mathcal{O}(n \log^2 n)$
2017	Данная работа	$\mathcal{O}(n^2 \log \sigma)$	$\mathcal{O}(n)$

Кроме того, нельзя не упомянуть о недетерминированных версиях решения этой задачи. Публикации, посвященные этому алгоритму не были найдены, так что назовем это фольклором.

Определим полиномиальный хеш последовательности $s = s_0 s_1 \dots s_{|s|-1}$ как $h(s) = (\sum_{i=0}^{|s|-1} s_i \cdot p^i) \mod m$ для выбранных параметров p и m, где в качестве m обычно берется простое число, а в качестве p произвольное.

Фиксировав длину l, можно скользящим окном посчитать полиномиальный хеш последовательностей $\mathcal{P}(s_i \dots s_{i+l-1})$ для всех подстрок длины l, начинающихся в $i \in [0, n-l]$, за $\mathcal{O}(n)$ времени.

Посчитав полиномиальные хеши векторов Парея всех подстрок обеих строк, можно проверить, есть ли совпадающие, используя хешмап. В случае совпадения хешей нужно произвести дополнительную проверку подстрок на Абелеву эквивалентность, поскольку совпадение полиномиальных хешей еще не гарантирует эквивалентность строк, хотя коллизия хешей векторов Парея очень маловероятна.

Оценить математическое ожидание времени работы такого алгоритма довольно сложно, но на практике он работает очень быстро, и есть все основания полагать, что математическое ожидание его времени работы $\mathcal{O}(n^2)$. На этот алгоритм была предложена задача на интернет-олимпиаду ИТМО в 2015 году [4].

1.2.2. Наибольший Абелев подквадрат и количество Абелевых подквадратов строки

Задачи о нахождении наидлиннейшего Абелево подквадрата и их количества были поставлены в 2016 году, с указанием на метод, опубликованный в том же году, позволяющий решать некоторые задачи, связанные с Абелевой эквивалетностью, за субквадратичное время [5].

Определение 11. Подстрока $s_i s_{i+1} \dots s_{i+k-1}$ является Абелевым периодом строки s, если существует такое j, что $s_i s_{i+1} \dots s_{i+k-1} \equiv s_{i+k} s_{i+k+1} \cdots s_{i+2k-1} \equiv \dots \equiv s_{i+(j-1)k} s_{i+(j-1)k+1} \dots s_{i+jk-1}$, и как $s_0 \dots s_{i-1}$, так и $s_{i+jk} \dots s_{|s|-1}$ являются Абелевыми подстроками подстроки $s_i s_{i+1} \dots s_{i+k-1}$.

Определение 12. Строка t является Абелевым бордером строки s, если префикс и суффикс длины $|t|$ строки s Абелево эквивалентны t.

Определение 13. Абелев бордер t является Абелевым покрытием строки s, если Абелевы вхождения строки t в s покрывают полностью всю строку s.

Авторы этой статьи предложили алгоритмы поиска длиннейшего/кратчайшего Абелевого подквадрата за $\mathcal{O}(n^2/\log^2 n)$, поиска кратчайшего Абелевого периода за $\mathcal{O}(n^2/\sqrt{\log n})$, поиска Абелевых бордеров строки за $\mathcal{O}(n^2/\log^2 n)$ и поиска всех Абелевых покрытий строки за $\mathcal{O}(n^2/\log n)$.

Так же недавно был опубликован новый алгоритм для решения задачи $3SUM^+$, используя методы аддитивной комбинаторики, решающий частный случай задачи значительно быстрее, чем за квадратичное время — за $\mathcal{O}(n^{1.86})$. С помощью этого алгоритма они показали, как отвечать на запросы *histogram queries* за $\mathcal{O}(1)$ после предподсчета за $\mathcal{O}(n^{1.86})$.

1.2.3. Обзор алгоритма решения 3SUM+

Кратко рассмотрим алгоритм решения задачи $3SUM^+$, предложенный в [6].

В упомянутой статье было предложено решение задачи $3SUM^+$ для монотонных линейно ограниченных множеств. Ограничимся случаем, когда множества A, B, C состоят из точек в двумерном пространстве.

Сначала вводится функция $cell(a)$ — все точки разбиваются по принадлежности к клеткам со стороной длины l, получая $(n/l)^2$ различных клеток, в которых могут находиться исходные точки. Кроме того, точки нормализуются так, что для любых точек $a \in A, b \in B$ верно, что $cell(a+b) = cell(a)+cell(b)$, это достигается путем решения четырех подзадач подзадач.

Ко множеству непустых клеток $A^* = \{cell(a) \ : \ a \in A\}$, $B^* = \{cell(b) \ : \ b \in B\}$, $C^* = \{cell(c) \ : \ c \in C\}$ применяется *BSG Corollary*, получая набор множеств $A_1^*, \ldots, A_k^*, B_1^*, \ldots, B_k^*$ и остаток R^*.

Для каждой клетки остатка $(a^*, b^*) \in R^*$ задача решается рекурсивно для множеств $\{a \in A \ : \ cell(a) = a^*\}$, $\{b \in B \ : \ cell(b) = b^*\}$ и $\{c \in C \ : \ cell(c) = a^* + b^*\}$.

Затем, для каждого $i = 1, \ldots, k$ применяется алгоритм, работающий на основе быстрого преобразования Фурье, чтобы получить все $\{a \in A \ : \ cell(a) \in A_i^*\} + \{b \in B \ : \ cell(b) \in B_i^*\}$, которые содержатся в надмножестве $T_i = \{s \in \mathcal{Z}^d \ : \ cell(s) \in A_i^* + B_i^*\}$.

Основная идея алгоритма в том, что после кластеризации точек, используя *BSG Corollary* можно довольно быстро выделить набор пар множеств с малой декартовой суммой, которые своей суммой плотно покрывают кластеризованное точек, оставляя небольшой остаток. Для каждой непокрытой

пары из остатка задача решается рекурсивно, а для каждой пары множеств, из-за того, что их декартова сумма невелика, используя быстрое преобразование Фурье, она считается быстрее, чем за квадрат.

Предложенный алгоритм является довольно большим продвижением в изучении задачи $3SUM^+$, являясь первым строго субквадратичным $(\mathcal{O}(n^\alpha), \alpha < 2)$ алгоритмом для задач, основанных на ограниченной монотонной $(\min,+)$ свертке.

В алгоритме есть пространство для дальнейшего исследования. Так, авторами поставлена задача для улучшения детерменированной версии алгоритма с целью избавления от перемножения матриц с предположением, что можно так же использовать преобразование Фурье. Помимо этого, поиск применений мощных методов аддитивной комбинаторики в задачах дискретной математики выглядит очень перспективной областью исследования.

ГЛАВА 2. ТЕОРЕТИЧЕСКИЕ ИССЛЕДОВАНИЯ

2.1. Абелевы квадраты

2.1.1. Поиск числа Абелевых подквадратов

2.1.1.1. Сведение к 3SUM+

Задачу о поиске числа Абелевых подквадратов в бинарной строке можно свести к задаче $3SUM^+$.

Пусть дана строка $s = s_0 s_1 \dots s_{n-1}$. Рассмотрим следующие множества:

$$A = \{(cnt_{c_1}(i), cnt_{c_2}(i)) \mid 0 \leqslant i \leqslant n\}, \tag{1}$$

$$B = \{(cnt_{c_1}(i), cnt_{c_2}(i)) \mid 0 \leqslant i \leqslant n\}, \tag{2}$$

$$C = \{(2 \cdot cnt_{c_1}(i), 2 \cdot cnt_{c_2}(i)) \mid 0 \leqslant i \leqslant n\}, \tag{3}$$

где $cnt_x(i)$ — число символов типа x на префиксе длины i строки s. Мотивация для этого сведения в том, что для Абелево подквадрата мы хотим, чтобы по обоим символам число вхождений этого символа в первой половине подстроки было равно числу вхождений во второй. Если рассматривать подстроку $[i; j)$, середина которой $k = \frac{i+j}{2}$, то должно быть выполнено $cnt_{c_1}(k) - cnt_{c_1}(i) = cnt_{c_1}(j) - cnt_{c_1}(k)$ и $cnt_{c_2}(k) - cnt_{c_2}(i) = cnt_{c_2}(j) - cnt_{c_2}(k)$, или $cnt_x(i) + cnt_x(j) = 2cnt_x(k)$. Поскольку $cnt_{c_1}(i) + cnt_{c_2}(i) = i$, становится понятно, что число абелевых подквадратов можно найти по формуле

$$(\#3SUM^+(A, B, C) - (n + 1)) \ / \ 2, \tag{4}$$

где $n + 1$ приходится вычитать, потому что нам неинтересны решения длины 0, а делить на два, потому что каждая подстрока будет посчитана дважды, с $i < j$ и $i > j$.

Поскольку лучшее известное решение задачи $3SUM^+$ для монотонных ограниченных множеств с элементами $\mathcal{O}(n)$, работает за $\mathcal{O}(n^{1.86})$, а наше сведение работает за линию, получаем решение задачи о количестве Абелевых подквадратов за ту же асимптотику.

На самом деле, предполагая, что средний компьютер выполняет порядка 10^9 операций в секунду, и предположив, что мы будем проверять решение на ограничениях порядка $n = 10^5$, асимптотическое ускорение в $n^{0.14}$, переводя в

числа, ускоряет всего в $(10^5)^{0.14} \approx 5$ раз, что может оказаться незаметным, а вспоминая о большой константе алгоритма и нескольких логарифмах можно предположить о его практической неэффективности. Все же, представляет интерес для изучения работа алгоритма на специфичных тестах или в среднем случае, вдруг он работает достаточно быстро с какой-то стороны.

Результаты экспериментальных запусков мы приведем в главе 3.

2.2. Наибольшая общая Абелева подстрока

2.2.1. Общий алгоритм

В качестве решения задачи нас будут интересовать только детерминированные алгоритмы решения задачи. Известно несколько недетерминированных решений, на практике работающих достаточно быстро, но они не являются темой исследования данной работы.

Будем подходить к лучшему решению по шагам от самого простого, на каждом шаге оптимизируя какую-то часть алгоритма, для лучшего понимания.

2.2.1.1. $\langle \mathcal{O}(n^2 \log \sigma), \mathcal{O}(n \log \sigma) \rangle$ w.h.p

Будем перебирать длину l и проверять, есть ли общая Абелева подстрока длины l. Сформулируем план. Мы будем строить $\mathcal{P}(t)$ для всех подстрок t длины l от 1 до n строк a и b, потом проверять, есть ли две строки с одинаковым $\mathcal{P}(t)$.

Будем строить векторы $\mathcal{P}(t)$ для всех подстрок длины l строк a и b по очереди, переходя от одной подстроки к следующей. Для этого нужно уметь удалять первый символ текущей подстроки и дописывать в конец новый символ. Расширим алфавит на один символ, добавив разделитель \$, нигде ранее не встречавшийся, и будем идти скользящим окном длины l по строке $a\$b$.

Хранить векторы $\mathcal{P}(t)$ будем в персистентном массиве длины σ, реализованном на персистентном дереве отрезков. Для того, чтобы перейти к следующей подстроке, нужно уменьшить число в ячейке, соответствующей удаляемому символу, на 1, и увеличить значение в ячейке, соответствующей добавляемому символу, на 1.

Для того, чтобы научиться сравнивать на равенство два корня дерева, соответствующие двум векторам $\mathcal{P}(t)$, будем при построении считать некоторое число $h(v)$ — класс эквивалентности всех вершин дерева. Этот класс

эквивалентности будет соответствовать набору значений на подотрезке символов, соответствующему этой вершине дерева отрезков — две вершины в одном классе эквивалентности, если они соответствуют одному и тому же подотрезку символов, и число вхождений каждого символа у них одинаково.

Будем поддерживать хешмап, в котором для пары чисел $\langle h_1, h_2 \rangle$ хранится класс эквивалетности этой пары, если она уже встречалась. Чтобы посчитать хеш для листа, получим класс эквивалентности у пары $\langle -pos, val \rangle$, где pos — номер символа, соответствующего этому листу, а val — значение, записанное в этой вершине. Такая пара, с отрицательным $-pos$, выбирается для того, чтобы избежать коллизии со внутренними вершинами, характеристиками которых являются пары неотрицательных чисел — пара уже посчитанных классов эквивалентности сыновей $\langle h(v_l), h(v_r) \rangle$, где v_l — левый сын вершины, а v_r — правый сын вершины. Когда нам нужно узнать хеш пары $\langle h_1, h_2 \rangle$, смотрим в хешмап: если там есть элемент с таким ключом, то соответствующий класс эквивалентности уже посчитан, иначе кладем туда новый элемент с таким ключом и значением, равным размеру хешмапа. Значения всех хешей таким образом будут принимать значения от 0 до $MapSize - 1$, где $MapSize$ — текущий размер хешмапа, не превышающий $\mathcal{O}(n \log \sigma)$.

Таким образом, после подсчета класса эквивалентности каждой вершины, для всех подстрок длины l первой и второй строки можно выписать их классы эквивалентности, а именно, значения в корнях персистентного дерева. Теперь нужно проверить, есть ли в двух массивах одинаковое число. Поскольку все значения имеют порядок $\mathcal{O}(n \log \sigma)$, это можно сделать, используя сортировку подсчетом.

Время работы — n итераций по l, и $\mathcal{O}(n \log \sigma)$ операций для каждой длины: вычисление класса эквивалентности каждой вершины дерева отрезков работает за $\mathcal{O}(1)$ w.h.p. используя обращение хешмапу. Расходуемая память $\mathcal{O}(n \log \sigma)$ на хранение дерева отрезков и хешмапа.

2.2.1.2. $\langle \mathcal{O}(n^2 \log \sigma), \mathcal{O}(n^2) \rangle$ deterministic

Посмотрим внимательнее на персистентное дерево отрезков из предыдущего решения. Это ациклический ориентированный граф, в котором каждая вершина имеет свой уровень (глубину) от 1 до $\log \sigma$, при чем на каждой глубине находится $\mathcal{O}(n)$ вершин.

Будем считать классы эквивалентности всех вершин, поднимаясь по уровням от листьев к корням, используя один хешмап размера $\mathcal{O}(n^2)$, который умеем очищать за $\mathcal{O}(1)$. В этом подразделе под хешмапом я подразумеваю просто массив на $\mathcal{O}(n^2)$ элементов с пометкой последнего изменения для возможности обнуления за $\mathcal{O}(1)$.

В этом решении класс эквивалентности будет иметь не сквозную нумерацию среди всех вершин дерева, как в предыдущем решении, а иметь отдельную нумерацию для каждого уровня вершин дерева.

Начнем с самого низкого уровня. Посчитаем классы эквивалентности листьев. Класс эквивалентности листа, как и в прошлом пункте, $h(\langle -pos, val\rangle)$, где и pos, и val принимают значения порядка $\mathcal{O}(n)$. Поэтому можно пройти по всем листьям в дереве, и посчитать классы, обращаясь к хешмапу напрямую и спрашивая, был ли уже такой же лист, и какой у него класс эквивалентности. Для того, чтобы обойти все листы за их число, при изменении значения в листе можно в лист складывать ссылку на новый лист, который появляется в следующей версии дерева отрезков.

Опишем переход от одного уровня к следующему. Допустим, уже посчитан класс эквивалентности всех вершин более глубокого уровня. Обратим внимание, что поскольку на каждой глубине $\mathcal{O}(n)$ вершин, классы этих вершин так же будут принимать значения $\mathcal{O}(n)$. Поэтому мы можем очистить хешмап и точно так же, как и для листьев, считать значение класса, к которому относится вершина, проверяя, была ли уже такая пара $\langle h(v_l), h(v_r)\rangle$.

Таким образом, мы построили дерево и посчитали хеши всех вершин за $\langle \mathcal{O}(n^2 \log \sigma), \mathcal{O}(n^2)\rangle$ полностью детерменированно.

2.2.1.3. $\langle \mathcal{O}(n^2 \log \sigma), \mathcal{O}(n)\rangle$ deterministic

Начнем с того, что на хранение дерева отрезков у нас сейчас уходит $n \log \sigma$ памяти, и это много. Чтобы уменьшить потребление памяти, можно использовать технику **limited node copying**. Кратко опишем эту технику, поскольку это будет необходимо для дальнейшего понимания алгоритма.

Вместо того, чтобы после пересоздания очередного листа пересоздавать весь путь до корня, будем хранить в каждой вершине дополнительный указатель, изначально нулевой. При изменении значения в листе будем подниматься по предкам, пока у предка дополнительный указатель уже занят, и создавать в этом случае новую вершину. Когда мы стоим в вершине и зна-

ем, что один из ее сыновей был изменен, а дополнительный указатель еще не занят, просто установим этот дополнительный указатель на новую версию этого сына и подпишем текущим глобальным временем. После такого изменения все еще несложно обратиться к какой-то версии дерева отрезков: нужно просто при переходе к сыновьям при выборе, куда спускаться, посмотреть, не нужно ли идти по дополнительному указателю.

Можно доказать, что таким образом построенное дерево занимает $\mathcal{O}(n)$ памяти, используя амортизационный анализ, но не будем об этом.

Подсчет классов эквивалентности для листьев и внутренних вершин в этом решении отличается.

Начнем с самого низкого уровня, а именно построим классы эквивалентности для листьев. Будем считать классы для листьев группами, для каждой позиции все листья, соответствующие этой позиции в массиве, вместе. Будем поддерживать счетчик ch — первый еще не использованный номер класса эквивалентности. Фиксировав позицию, которую мы сейчас обрабатываем, обойдем все листья с этой позицией (для этого можно хранить в каждом листе ссылку на предыдущий лист этой позиции), и листу со значением val присвоим класс $ch + val$. После этого увеличим ch на $maxValue_{pos} + 1$, где $maxValue_{pos}$ — наибольшее значение, которое было в ячейке pos в одной из версий дерева отрезков. Мы знаем $maxValue_{pos}$ для каждой позиции, и можно заметить, что число классов эквивалентности на этом уровне $\mathcal{O}(n)$, поскольку $\sum\limits_{pos=1}^{\sigma} maxValue_{pos} = \mathcal{O}(n)$.

Опишем переход от одного уровня к следующему. Допустим, мы посчитали классы для всех (даже больше, чем для всех вершин, об этом далее) вершин на предыдущем уровне. После завершения очередного уровня, будем передавать на уровень выше не только посчитанные классы эквивалентности всех вершин, но и все не созданные явно из-за **limited node copying** копии вершин. Этот момент стоит описать более подробно, потому что сделать это не совсем тривиально.

Рассмотрим момент, когда мы развернули очередной уровень дерева, создав все несозданные явно вершины этого слоя.

Лемма: каждая вершина в сжатом персистентном дереве соответствует набору вершин, отвечающих за такой же отрезок символов в несжатом дереве, созданных в некоторый промежуток времени $[t_1; t_2)$, причем для всех вершин

несжатого дерева, отвечающих за этот отрезок, эти промежутки времен не пересекаются.

Доказательство. Рассмотрим «срез» персистентного дерева в некоторый момент времени. Срез дерева в момент времени — дерево без дополнительных ссылок, в котором один из сыновей заменен на вершину по дополнительной ссылке, если на этой ссылке написано время не большее, чем момент среза. Поскольку именно такой срез дерева поддерживался при построении, легко понять, что вершина «живет» в сжатом дереве с момента создания до момента замены на свою новую версию, и отвечает за отрезок времен, соответствующих всем изменениям в ее поддереве, произошедших за ее время жизни.

Теперь опишем, как передавать отрезки версий вершин предкам в алгоритме. За время жизни вершины у нее несколько раз меняются предки в текущем срезе персистентного дерева (предок меняется, если произошло изменение в другом сыне предка, и у предка не осталось дополнительного указателя). В алгоритме, когда мы развернули каждую вершину, создав все ее явные копии, нужно для каждого ее предка отправить такой подотрезок этих копий, который соответствует моментам времени, когда данный предок являлся предком текущей неявной вершины. Поскольку все эти моменты времени не пересекаются, можно явно передать всем предкам информацию о том, что в такие времена у вершины сыном была такая-то вершина, суммарно мы передадим все на следующий уровень за $\mathcal{O}(n)$ памяти и времени. Потом нужно будет в вершине-предке отсортировать все пришедшие данные из потомков, но поскольку это объединение не больше трех отсортированных массивов (информация приходила только из непосредственного сына, а их всего три с учетом дополнительного указателя), это можно так же сделать за линейное время.

Поскольку без оптимизаций на каждом уровне линейное количество вершин, после создания всех неявных вершин их останется линейное число. При переходе на послеследующий уровень все эти вершины можно удалить, чтобы сохранить линейную память. Чтобы получить для вершины список всех интересных времен, и понять, сколько ее копий нужно сделать, нужно объединить список всех времен у сыновей и у дополнительного указателя, если он есть.

Следующее, что нужно сделать — сгруппировать все вершины с одинаковым $h(v_l)$ в одну группу, и обработать их вместе, чтобы назначить им соответствующие классы эквивалентности. Пусть у очередной вершины (для каждого варианта глобального времени, которое в ней интересно) классы сыновей $h(v_l)$ и $h(v_r)$. Запишем в вектор с номером $h(v_l)$ напоминание: нужно посчитать $h(\langle h_1, h_2 \rangle)$ и записать его в текущую вершину v. После того, как мы сделали это для всех вершин текущего уровня, нужно перебрать $h(v_l)$, создать хешмап размера $\mathcal{O}(n)$ (на самом деле, можно использовать один и тот же, просто очищая его за $\mathcal{O}(1)$), и перебрать соответствующее ему $h(v_r)$, назначая вершинам текущего уровня соответствующие классы. Как обычно, если $h(v_r)$ есть в хешмапе, достаем оттуда посчитанный класс, иначе сопоставляем ему новый.

После того, как классы эквивалентности всех вершин посчитаны, освободим память с предыдущего уровня и перейдем к следующему. В конце получим посчитанные классы для всех корней.

Используемое время так и осталось $\mathcal{O}(n^2 \log \sigma)$, а вот требуемая память стала всего $\mathcal{O}(n)$.

Остается открытым вопрос существования более быстрых детерменированных алгоритмов, работающих за $o(n^2 \log \sigma)$, в частности, $\mathcal{O}(n^2)$. К существованию такого алгоритма есть такие предпосылки, как недетерменированные алгоритмы, работающие за $\langle \mathcal{O}(n^2), \mathcal{O}(n) \rangle$.

2.2.2. Случай бинарных строк

2.2.2.1. Оценка математического ожидания НОАП бинарных строк

Отдельный интерес представляет случай $\sigma = 2$. В предыдущих работах был описан алгоритм, работающий за время $O(n^2 / \log n)$ [1].

В этой же статье рассмотрено матожидание длины НОАП двух случайных бинарных строк длины n и сделано предположение 5.1 [1] о том, что $LCAF_{avg} \geqslant n - O(\log n)$, где $LCAF_{avg}$ — матожидание НОАП двух случайных бинарных строк. Это преположение выглядит слишком смелым, рассмотрим его подробнее.

Теорема 14. Для любой функции $f(n) = o(n)$ верно что для двух случайных бинарных строк длины n: $LCAF_{avg} < n - f(n)$.

Доказательство.

Лемма: для любой функции $f(n) = o(n)$ с фиксированной вероятностью $P > 0$, не зависящей от n, верно $LCAF_{avg} < n - f(n)$.

Доказательство. Обратим внимание, что центральная подстрока длины $n - 2f(n)$, полученная отрезанием суффикса и префикса длины $f(n)$, является подстрокой любой строки длины $n - f(n)$ этой же строки.

Рассмотрим задачу как задачу случайного блуждания: пусть $x_i = A_i - B_i$, где A, B — наши случайные строки. От стандартной задачи случайного блуждания она отличается тем, что кроме переходов $|x_{i+1} - x_i| = 1$ разрешены переходы $x_{i+1} = x_i$. Абелево равенство двух подстрок A и B эквивалентно тому, что подпуть блуждания x возвращается в свое начало, $x_r = x_l$.

Наше случайное блуждание имеет следующие вероятности:

Δx	$p(\Delta x)$
-1	0.25
0	0.5
1	0.25

Обозначим за $P(n, k)$ вероятность после n испытаний получить сумму k.

Лемма: $P(n, k) = C(2n, n + k)$. Доказательство: Если посмотреть на геометрический смысл этого распределения, то $P(n,k)$ — вероятность за $2n$ равновероятных шагов вправо или вверх дойти до диагонали $x + y = 2n$ и остановиться на диагонали $y - x = k$, что сходится с формулой $P(n,k) = P(n-1,k-1) + 2P(n-1,k) + 2P(n-1,k+1)$.

При больших n будем приближать наше биномиальное распределение нормальным, $P(n, k) = \sqrt{n}N(0,1)$

Вспомним о правиле трех сигм.

Поскольку у нормального распределения $\sqrt{n}N(0,1)$ среднеквадратичное отклонение $\sigma = \sqrt{n}$, по правилу трех сигм можно сказать, что у центрального пути длины $n - 2f(n)$ вероятность остановиться в промежутке $[-3\sqrt{n - 2f(n)}; 3\sqrt{n - 2f(n)}]$ около 0.9973, а поскольку $f(n) = o(n)$, вероятность того, что изменение координаты окажется вне промежутка $[-2.5\sqrt{n}; 2.5\sqrt{n}]$, хотя бы 0.0026.

Для того, чтобы получить отрезок длины $n - f(n)$ с нулевой суммой, нужно взять какой-то суффикс префикса длины $f(n)$ и какой-то префикс суф-

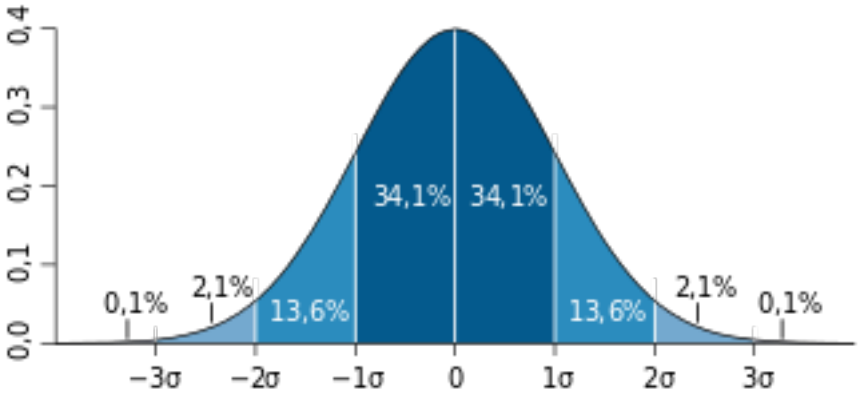

Рисунок 1 – правило трех сигм [7]

фикса длины $f(n)$. Покажем, что с достаточной вероятностью мы не сможем приблизиться к нулю за $f(n)$ шагов.

Скажем, что наше блуждание сейчас будет обычным случайным, с двумя переходами $+1$ и -1. Этот переход является корректным, так как если в обычном случайном блуждании утверждение выполнено, то оно верно и для нашего случая. И действительно, можно продлить все испытания, в которых был переход по 0 до ровно k (общее число испытаний) ненулевых переходов и прийти к случайному блужданию, при чем максимум модуля отклонения на префиксе мог только увеличиться.

Поскольку $2f(n) = o(n)$, докажем более сильное условие. За n шагов с вероятностью $C > 0$, не зависящей от n, случайное блуждание не попадет в область с координатой меньше $-\sqrt{n}$.

Переформулируем эту задачу как задачу о разорении: игрок имеет $\sqrt{n}$ денег и играет n раундов против бесконечно богатого казино, и нужно найти вероятность разорения игрока. В [8] и в [9] можно найти следующую формулу: вероятность проигрыша игрока со стартовым капиталом a за n раундов равна $y_{a,n} = 1 - \frac{2}{\sqrt{\pi}} \int_0^t e^{-u^2} du + \Delta$, где Δ — малый остаточный член, а $t = \frac{a}{\sqrt{2(n+\frac{2}{3})}}$.

В нашем случае, $a = \sqrt{n}$, $t \approx \frac{1}{\sqrt{2}}$, и вероятность проигрыша в пределе равна $1 - \frac{2}{\sqrt{\pi}} \int_0^{\frac{1}{\sqrt{2}}} e^{-u^2} du \approx 0.395$.

Таким образом, с вероятностью хотя бы 0.0026 центральный подпуть будет иметь отклонение от нуля хотя бы в $2.5\sqrt{n}$, и с вероятностью хотя бы 0.605^2 и префикс, и суффикс, который мы допишем к этой строке, будут иметь отклонение не больше, чем на $\sqrt{n}$, то есть, с вероятностью $P \geqslant 0.026 \cdot 0.605^2$

у двух случайных строк наибольшая Абелева подстрока будет меньше, чем $n - f(n)$.

Вернемся к доказательству теоремы. Будем доказывать ее от противного — пусть есть $f(n) = o(n)$ такое, что $LCAF_{avg} \geqslant n - f(n)$.

Оценим $LCAF_{avg}$. По лемме, с вероятностью $0.026 \cdot 0.605^2 = 0.0095$ $LCAF$ будет не больше, чем $g(n) = \sqrt{nf(n)}$. Тогда

$LCAF_{avg} \leqslant 0.0095 \cdot (n - g(n)) + (1 - 0.0095) \cdot n = n - 0.0095 \cdot g(n) < n - f(n)$, т.к. $f = o(g)$. Противоречие.

Кроме того, докажем грубую оценку снизу:

Теорема 15. Для двух случайных бинарных строк длины n: $LCAF_{avg} \geqslant 0.05n$.

Доказательство. Снова приблизим наше случайное блуждание нормальным распределением и воспользуемся правилом трех сигм.

С вероятностью $2 \cdot 0.34$ за первые $n/2$ шагов мы остановимся в зоне $[-\sqrt{\frac{n}{2}}; \sqrt{\frac{n}{2}}]$. После этого, с вероятностью хотя бы $0.136 + 0.021 + 0.001 \geqslant 0.15$ (вероятность пройти в нужную сторону больше, чем $\sqrt{\frac{n}{2}}$ шагов, см. рисунок 1) мы за следующие $n/2$ шагов пройдем в другую сторону хотя бы $\sqrt{\frac{n}{2}}$ шагов, обязательно перейдя через точку старта. Таким образом, с вероятностью хотя бы $2 \cdot 0.34 \cdot 0.15$ НОАП будет хотя бы $n/2$, или $LCAF_{avg} \geqslant 0.05$.

Итого, получаем, что матожидание НОАП у двух случайных бинарных строк сверху и снизу ограничено линейными функциями, но более точная оценка ее поведения остается нерешенной задачей.

2.2.2.2. Сведение к 3SUM+

Можно получить асимптотически хорошее решение, использовав алгоритм решения $3SUM^+$, описанный в [6].

В упомянутой статье было предложено **Corollary 3.6**, позволяющее решать задачу *histogram indexing* для случая бинарной строки за предподсчет $\mathcal{O}(n^{1.86})$, и отвечать после этого на запрос за $\mathcal{O}(1)$. В процессе предподсчета оказываются посчитаны значения минимального и максимального количества вхождений c_0 в строку длины l для каждой длины l. Используя эти массивы можно найти НОАП, перебрав длину l общей подстроки, проверив, пересекаются ли множества подстрок длины l с фиксированным количеством вхождений у обеих строк, и выбрав наибольшее подходящее l.

Используя это сведение, получаем решение задачи поиска НОАП для бинарного алфавита за $\mathcal{O}(n^{1.86})$, лучшее на данный момент.

ГЛАВА 3. ПРАКТИЧЕСКИЕ РЕЗУЛЬТАТЫ

3.1. Параметры компьютера, производящего вычисления

Все вычисления были выполнены на ноутбуке Acer с процессором Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz.

Реализации всех алгоритмов были написаны на языке C++, и были скомпилированы с флагами $-O2$.

3.2. Наибольшая общая Абелева подстрока

3.2.1. Случай бинарного алфавита

Первое, что мы сделаем — посмотрим, как себя ведет на практике матожидание наибольшей общей Абелевой подстроки двух случайных бинарных строк.

Мы выполнили 10^4 запусков поиска НОАП для различных значений n до 10^4. Такого количества запусков оказалось вполне достаточно, чтобы среднее значение НОАП стабилизировалось. Полученный результат можно увидеть на рисунке 2.

Видно, что функция ведет себя практически как прямая $y = 0.83x$, что подтверждает полученные теоретические линейные оценки как сверху, так и снизу.

3.2.2. Случай большого алфавита

Будем сравнивать время работы предложенного в этой статье алгоритма за $\mathcal{O}(n^2 \log \sigma)$ с алгоритмом, предложенным A. Attabi et al [1]. Главными достоинствами этого алгоритма является необычайная простота, котороая приводит к очень маленькой константе, скрытой во временной оценке, потому что в нем не используется никаких тяжелых алгоритмов. Алгоритм S. Grabowski et al [2] за $\mathcal{O}(n^2 \log \sigma)$ имеет такую же временную асимптотику, поэтому с ним сравнение не проводилось: есть все основания полагать, что он работает в несколько (как минимум, в два) раз медленнее из-за эвристик, приводящих к уменьшению потребления памяти.

Так же в процессе анализа было принято решение не сравнивать с алгоритмом S. Grabowski et al [2] за $\mathcal{O}(n^2 \log^2 n \log^* n)$, потому что в процессе ознакомления с ним было выяснено, что у него слишком большая скрытая константа — алгоритм получен в результате деамортизации недетерминированного, используя для этого довольно ресурсоемкие операции и структуры данных.

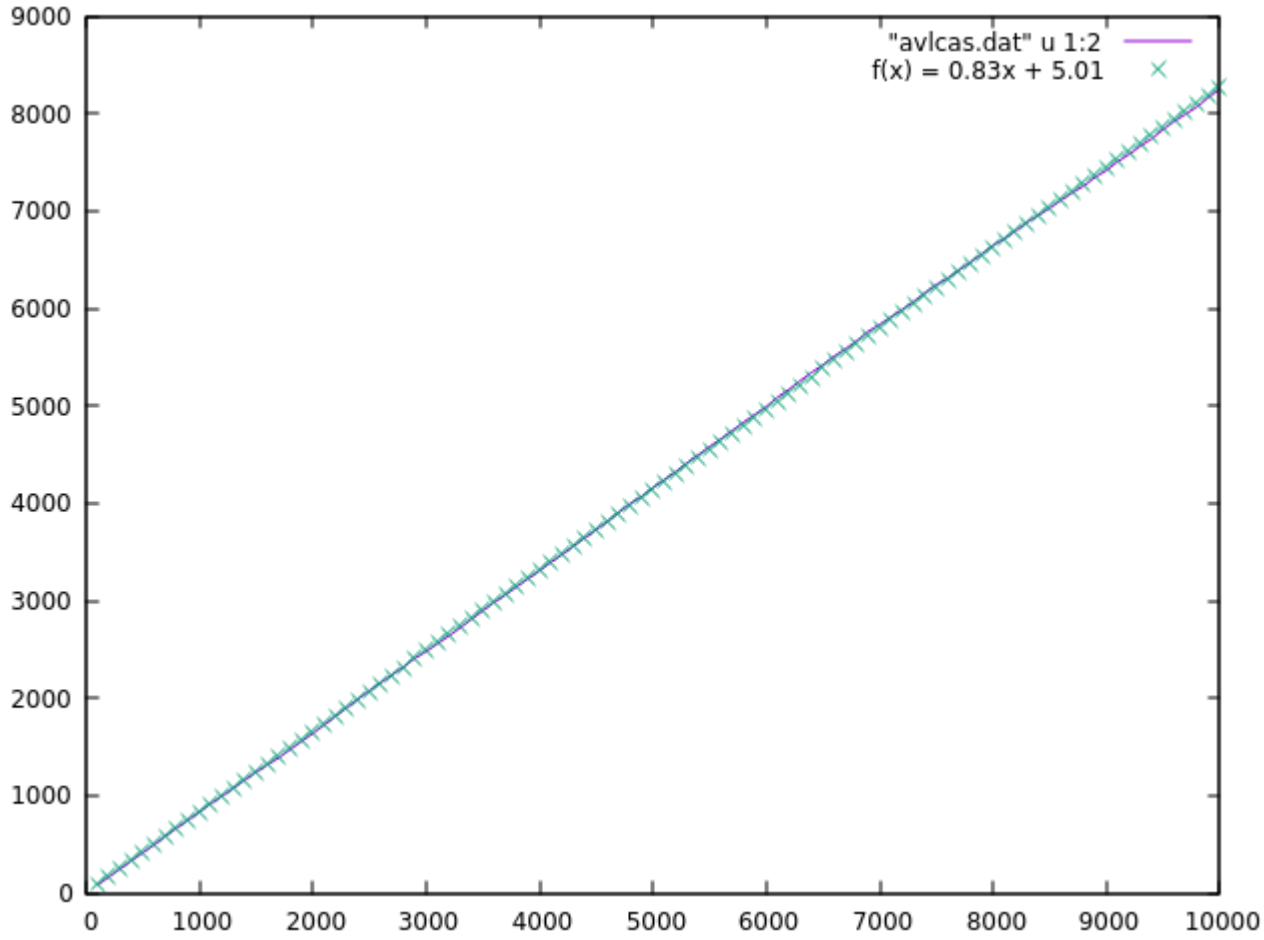

Рисунок 2 – зависимость матожидания НОАП от длин строк

В свою очередь, алгоритм, предложенный в настоящей работе, достаточно тяжелый как в плане написания кода, так и имеет сложно оцениваемую скрытую константу времени работы из-за необходимости в аккуратной работе с памятью. Представляет большой интерес понимание того, является ли данный алгоритм лишь теоретическим улучшением существующих, или все же он дает ощутимое ускорение на практике.

Сравнение времени работы предложенного алгоритма в зависимости от n на тесте из двух строк длины n, состоящих из случайных символов из алфавита мощности n, представлено на рисунке 3.

Таким образом, можно с уверенностью сказать, что начиная с не очень большого n предложенный алгоритм начинает выигрывать у стандартной реализации известного алгоритма, и этот выигрыш будет только увеличиваться из-за более хорошей асимптотики.

К сожалению, несмотря на строго более хорошую асимптотику, на маленьких тестовых данных предложенный алгоритм работает несколько мед-

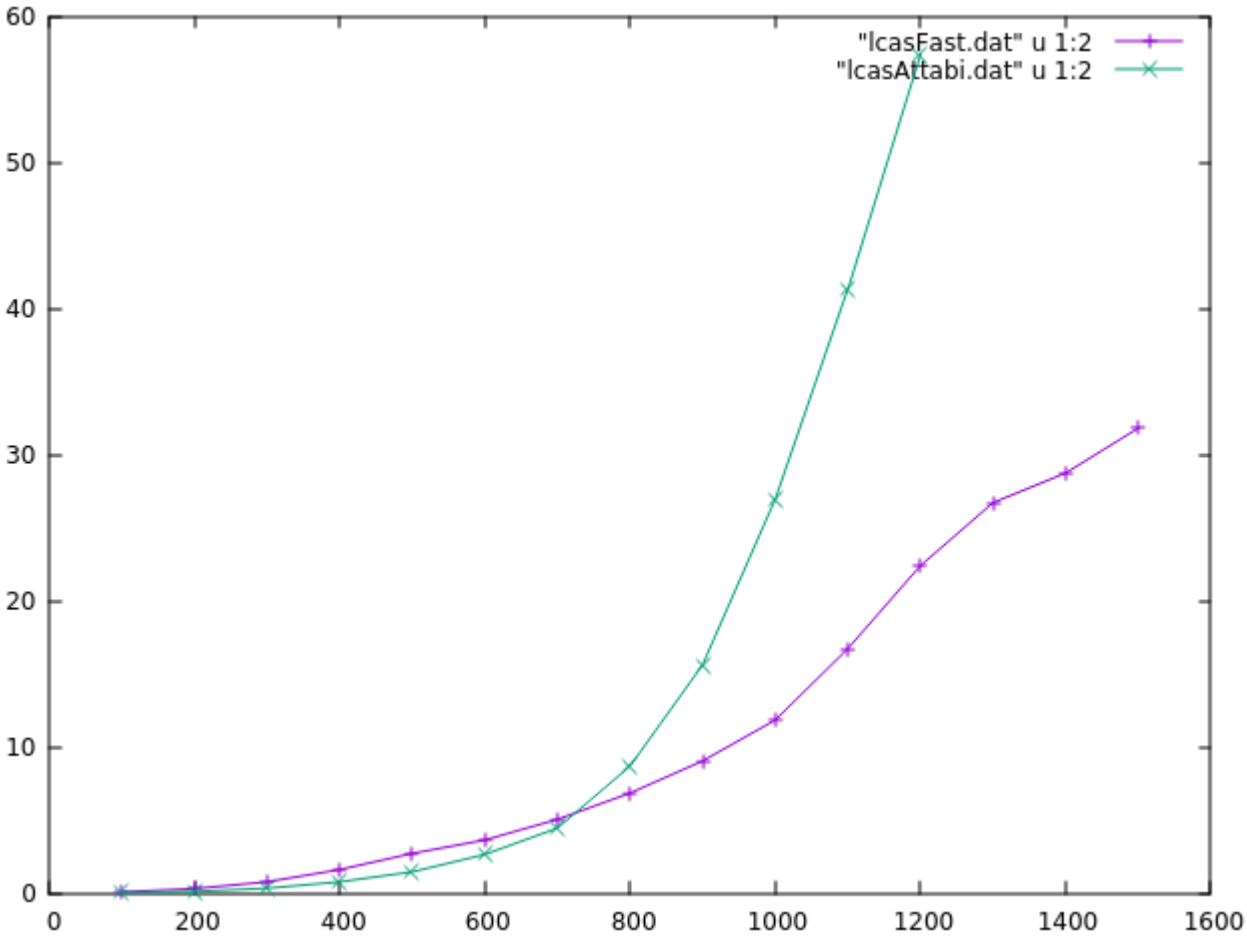

Рисунок 3 – Зависимость времени работы алгоритмов от n на случайной строке над большим алфавитом

леннее, чем алгоритм из [1]. Это можно объяснить наличием скрытой в асимптотике алгоритма константой, но она достаточно мала, чтобы все равно обеспечить выигрыш на реальных данных.

3.3. Количество Абелевых подквадратов

Мы реализовали алгоритм нахождения числа Абелевых подквадратов с помощью сведения к алгоритму решения монотонного ограниченного случая $3SUM^{+}$. Он был протестирован на строках из одинаковых символов, и на наборе пар случайных бинарных строк. Как оказалось, константа алгоритма довольно велика, и на практике он показал себя достаточно плохо, в несколько раз проигрывая наивному решению за $\mathcal{O}(n^2)$.

На рисунке 4 можно увидеть зависимость времени работы решения в секундах от n на тесте из строки, состоящей из n одинаковых символов. График действительно довольно похож на $\mathcal{O}(n^{1.86})$, но из-за нескольких логарифмов в асимптотике растет несколько быстрее.

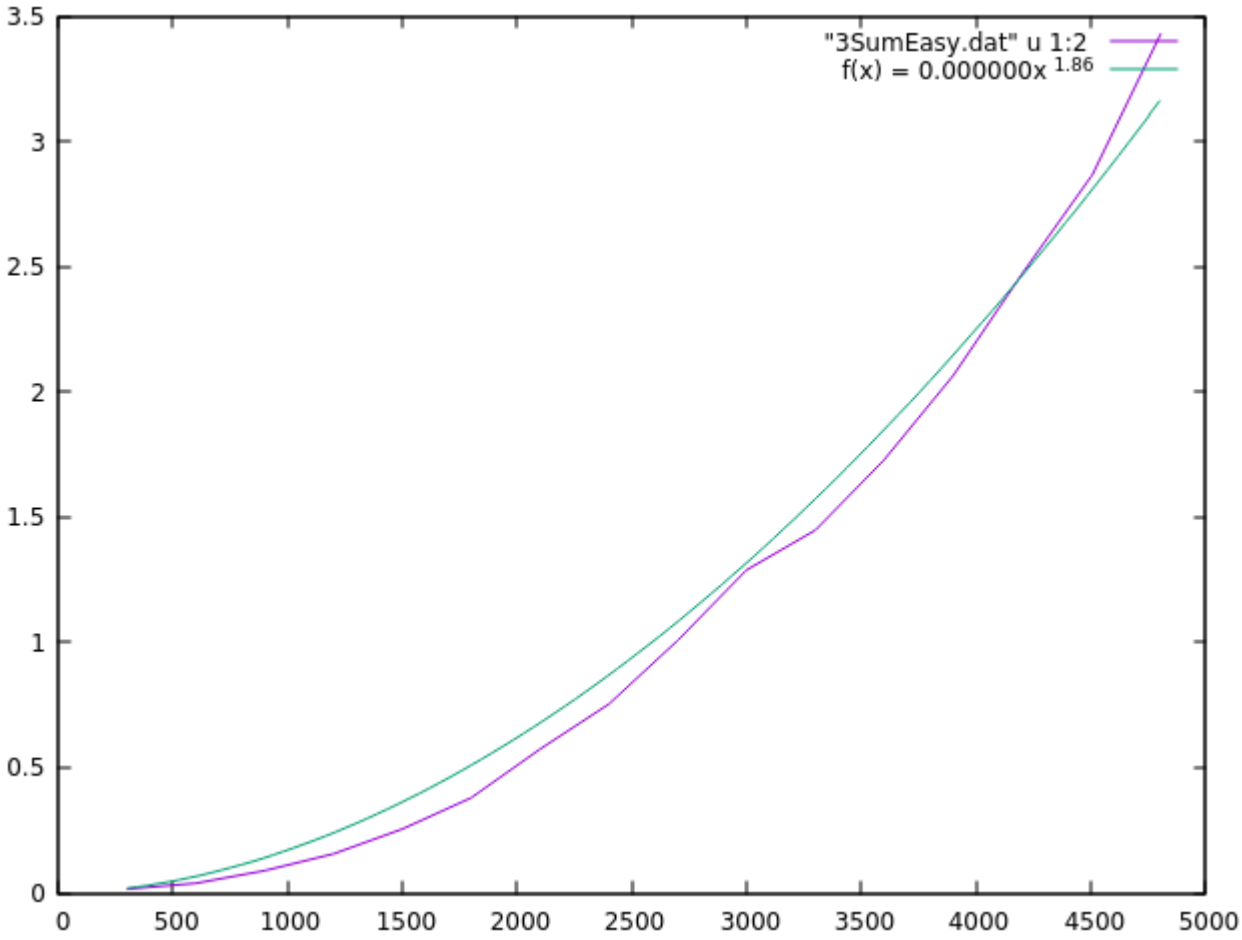

Рисунок 4 – зависимость средней времени работы на строке из одинаковых символов от ее длины

На рисунке 5 можно увидеть зависимость времени работы решения в секундах от n на тесте из строки, состоящей из n случайно сгенерированных символов.

Время работы алгоритма довольно сильно меняется как от запуска к запуску из-за разных тестов, так и от различных значений n, так как различные ветки программы выполняются с разными вероятностями и работают разное время — не приходится удивляться некоторому увеличению производительности при увеличении n.

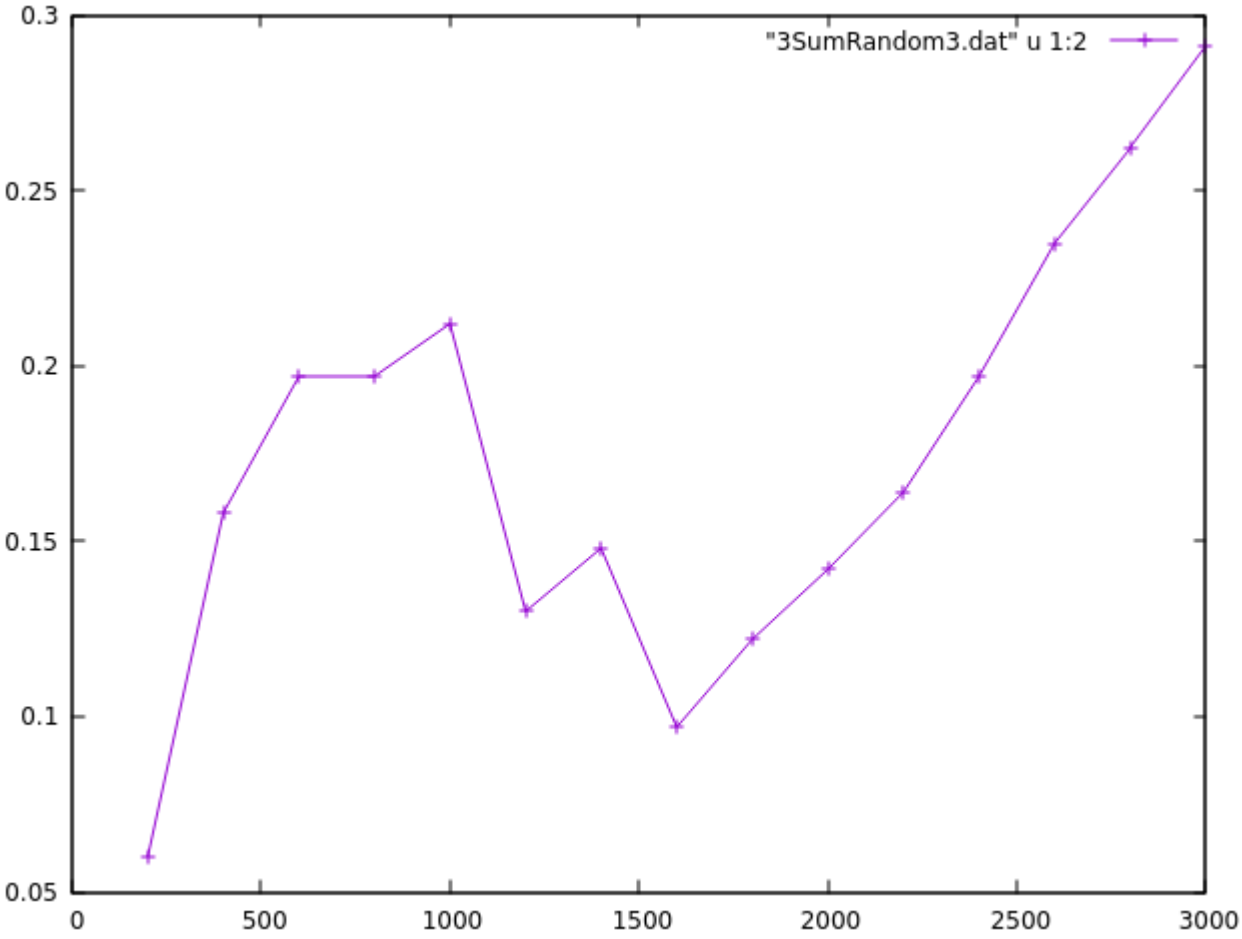

Рисунок 5 – зависимость средней времени работы на случайной строке от ее длины

ЗАКЛЮЧЕНИЕ

В рамках данной работы была сведена задача о числе Абелевых подквадратов в бинарной строке к монотонному и линейно ограниченному случаю задачи $3SUM^{+}$. У этой задачи недавно было опубликовано неожиданное асимптотически очень хорошее решение. Этот алгоритм был реализован, и мы пришли к выводу, что его реализация не очень эффективна, проигрывая в несколько раз простому алгоритму с более плохой асимптотикой, но хорошей константой. Несмотря на это, заметен потенциал предложенного метода, и он вероятно может быть улучшен до производительности, действительно применимой на практике.

Кроме того, в этой работе было проведено исследование задачи НОАП. Был разработан новый алгоритм, существенно улучшающий известные теоретические результаты для задачи в общей формулировке без ограничений на размер алфавита. По сравнению с предыдущим лучшим результатом для больших алфавитов, алгоритм не только асимптотически выигрывает по затрачиваемому времени и памяти, но и более прост для понимания. Несмотря на хороший результат, остается открытым вопрос поиска более быстрых детерменированных алгоритмов. Так же показано на практике, что предложенный алгоритм значительно превосходит известные ранее алгоритмы.

Был проведен анализ поведения НОАП для случайных бинарных строк — вопрос, исследовавшийся ранее. Экспериментально показана линейная зависимость матожидания НОАП от длины строк, и теоретически обоснованы линейные оценки сверху и снизу. Поиск точных оценок остался в качестве нерешенной задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Algorithms for Longest Common Abelian Factors / A. Alatabbi [и др.] // International Journal of Foundations of Computer Science. — 2016. — Т. 27, № 05. — С. 529–543. — DOI: 10.1142/S0129054116500143. — eprint: http://www.worldscientific.com/doi/pdf/10.1142/S0129054116500143. — URL: http://www.worldscientific.com/doi/abs/10.1142/S0129054116500143.

2 Longest Common Abelian Factors and Large Alphabets / G. Badkobeh [и др.] // String Processing and Information Retrieval: 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings / под ред. S. Inenaga, K. Sadakane, T. Sakai. — Cham : Springer International Publishing, 2016. — С. 254–259. — ISBN 978-3-319-46049-9. — DOI: 10.1007/978-3-319-46049-9_24. — URL: http://dx.doi.org/10.1007/978-3-319-46049-9_24.

3 *Mehlhorn K.*, *Sundar R.*, *Uhrig C.* Maintaining Dynamic Sequences Under Equality-tests in Polylogarithmic Time // Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. — Arlington, Virginia, USA : Society for Industrial, Applied Mathematics, 1994. — С. 213–222. — (SODA '94). — ISBN 0-89871-329-3. — URL: http://dl.acm.org/citation.cfm?id=314464.314496.

4 *ИТМО И.-о.* Третья командная олимпиада, задача A. — 2015. — URL: http://neerc.ifmo.ru/school/io/archive/20151107/problems-20151107-advanced.pdf.

5 *Kociumaka T.*, *Radoszewski J.*, *Wiśniewski B.* Subquadratic-Time Algorithms for Abelian Stringology Problems // Mathematical Aspects of Computer and Information Sciences: 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers / под ред. I. S. Kotsireas, S. M. Rump, C. K. Yap. — Cham : Springer International Publishing, 2016. — С. 320–334. — ISBN 978-3-319-32859-1. — DOI: 10.1007/978-3-319-32859-1_27. — URL: http://dx.doi.org/10.1007/978-3-319-32859-1_27.

6 *Chan T. M.*, *Lewenstein M.* Clustered Integer 3SUM via Additive Combinatorics // CoRR. — 2015. — T. abs/1502.05204. — URL: http://arxiv.org/abs/1502.05204.

7 *Википедия* Среднеквадратическое отклонение — Википедия, свободная энциклопедия. — 2017. — URL: http://ru.wikipedia.org/?oldid=85800806.

8 *Г. П. Иванова В. Я. К.* Одна оценка вероятности разорения // Теория вероятн. и ее примен. — Cham : Springer International Publishing, 2007. — С. 359–363. — URL: http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=tvp&paperid=178&option_lang=rus.

9 *Uspensky J. V.* Introduction to mathematical probability. — 1st ed. — New York ; London : McGraw-Hill, 1937. — "Problems for solution"with answers at end of each chapter.