Федеральное государственное автономное образовательное учреждение высшего профессионального образования

«Уральский Федеральный Университет

имени первого Президента России Б.Н.Ельцина»

На правах рукописи

Рубинчик Михаил Валентинович

Вычислительная сложность некоторых задач обработки строк

Специальность 01.01.09 - дискретная математика и математическая кибернетика

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель
доктор физико-математических наук
профессор Шур Арсений Михайлович

Екатеринбург 2016

Оглавление

Глава 1

Введение

Строка (последовательность символов) — один из центральных объектов компьютерных наук. Любые данные, имеющие линейную структуру (например, текст на естественном или искусственном языке, лог сервера или колебания курса валюты) можно представить в виде строки и изучать особенности в этой строке. Начало активного изучения алгоритмов, связанных со строками, приходится на 1970е годы. Раздел компьютерных наук, посвящённый изучению алгоритмов обработки строк, называется *стрингологией* (stringology). Впервые это название было предложено в 1985 в работе [21], когда область уже имела какой-то набор результатов, но сложно было назвать её отдельной наукой. Однако уже в 90х годах наблюдается стремительный рост числа публикаций по данной тематике. Одной из самых сильных предпосылок развития является появление биоинформатики, науки, изучающей «биологические строки» (ДНК, РНК, белки и др.). Именно биоинформатика породила множество задач, изучаемых стрингологами. Не менее важным источником задач являются современные информационные технологии, требующие эффективных алгоритмов анализа и обработки данных. Среди задач этой группы выделяются задача поиска в массиве данных и задача сжатия данных. Кроме того, стрингология, относимая к фундаментальной информатике, решает большое число чисто математических задач, не нашедших еще практического применения. Сюда относится, в частности, классификация задач о строках по их вычислительной сложности. Многие теоретические задачи приходят в стрингологию из комбинаторики слов.

Развитость стрингологии как самостоятельной науки может показать, к примеру, наличие ряда монографий [2, 12, 13, 48] и специализированных международных конференций и семинаров: String Processing and Information Retrieval (SPIRE), Combinatorial Pattern Matching (CPM),

Prague Stringology Conference (PSC), StringMasters.

Данная работа сконцентрирована на двух классах задач из стрингологии. Первая — алгоритмы, связанные с поиском и анализом палиндромов[1] в строках. Вторая — алгоритмы, связанные с повреждёнными строками.

1.1 Структура диссертации и организация текста

Данная работа разбита на 3 главы, первая из которых вводная, две остальные содержат непосредственно новые результаты.

Во введении после данного раздела следуют предварительные сведения, содержащие формальное описание модели вычисления, в которой мы будем работать, многие базовые определения, а также используемые алгоритмы и структуры данных. После чего сформулированы основные результаты данной работы и приводится содержание глав, в которых будут описаны эти результаты.

Глава 2 посвящена изучению палиндромов в строках. Большая её часть содержит описание новой структуры данных под названием «овердрево»[2] и множеству её применений. Среди них: поиск различных подпалиндромов строки, задачи о разбиении строк на палиндромы, а также перечисление палиндромно-насыщенных (содержащих наибольшее число палиндромов) строк.

Известно, что число различных подпалиндромов в строке длины n не превосходит $n + 1$ (см. [16]) и известен линейный от n алгоритм нахождения их всех (см. [25]). Нахождение оптимального алгоритма, позволяющего за линейное время онлайново посчитать число различных подпалиндромов в каждом префиксе строки, — открытая проблема, поставленная в [25]. Мы опишем онлайновый алгоритм, работающий за время $O(n \log |\Sigma|)$, основанный на классических алгоритмах Укконена [49] и Манакера [35]. После чего приведём существенно более простой и быстрый алгоритм с такой же асимптотикой, обобщив который, получим новую структуру данных «овердрево».

Задачи о разбиении на палиндромы представляют давний интерес в теории формальных языков. Известно, что асимптотически наилучший универсальный алгоритм распознавания контекстно-свободных языков — алгоритм Валианта [50] — работает за более чем квадратичное время. При этом ни для одного контекстно-свободного языка до сих пор не удавалось доказать

[1] Палиндром — строка, равная своему развороту

[2] Название отражает то, что это древовидная структура для хранения и обработки палиндромов

несуществование алгоритма, распознающего этот язык за линейное время. Одно время предполагалось, что языки конкатенации чётных палиндромов и язык конкатенации палиндромов с длиной больше единицы невозможно распознать за $O(n)$ (см. [31, раздел 6]). Оказалось, что это не так: в [31] описан линейный алгоритм для распознавания первого языка, а в [22] — для второго. Распознавание языка конкатенации k палиндромов для произвольного фиксированного k оказалось гораздо более сложной задачей. В [22] приведён линейный алгоритм, распознающий язык для $k = 1, 2, 3, 4$. Вариант такого алгоритма можно найти в [13, раздел 8]. В [22] и [13] был поставлен вопрос о существовании линейного алгоритма для $k > 4$. В статье [34] был приведён алгоритм за $O(kn)$, который, к сожалению, важен только для теории ввиду своей реализационной сложности и большой константы. В данной работе будет приведён алгоритм, основанный на «овердреве», работающий за $O(n \log n)$, но имеющий маленькую константу, не зависящую от k, и решающий одновременно задачу нахождения палиндромной длины[3].

Структура палиндромно-насыщенных строк изучается в комбинаторных работах начиная с [16]. В [23] впервые был поставлен вопрос об оценке числа таких строк. Используя одну из модификаций овердрева, мы строим алгоритм, перечисляющий все такие строки до заданной длины над заданным алфавитом константного размера за время $O(1)$ на одну строку. Полученные результаты вошли в онлайн-энциклопедию целочисленных последовательностей, см. [46, A216264] и были использованы в работе [26] для оценки функции роста множества бинарных палиндромно-насыщенных строк.

В **главе 3** рассматривается задача об оптимальном восстановлении повреждённых строк. Решаются два варианта этой задачи, различающиеся критерием оптимальности, который в обоих случаях связан с поиском подстроки в строке. Для обоих вариантов будут приведены полиномиальные решения в частных случаях и доказательства NP-трудности для общего случая.

Задача поиска подстроки в строке — это одна из самых известных и хорошо изученных задач стрингологии. Ее стандартная постановка состоит в следующем. Дано две строки — более длинный «текст» S и более короткий «шаблон» P, требуется найти все вхождения шаблона в текст, т. е. все пары (i, j) такие, что подстрока в S, начинающаяся в i-й и заканчивающаяся в j-й позиции, равна P. Эта задача имеет большое прикладное значение для информационного поиска и биоинформатики, и это значение непрерывно возрастает с развитием информационных

[3]Палиндромная длина строки — минимальное число палиндромов, конкатенация которых даёт исходную строку.

технологий. Существует множество вариаций данной задачи (например, с ограничениями на используемую память или на доступ к строкам, с приблизительным поиском, с «неточным» шаблоном, и т.п.).

Одно из наиболее интересных с практической точки зрения обобщений задачи поиска подстроки в строке состоит в предположении, что текст и/или шаблон могут быть повреждены при передаче данных (например, из-за шума в канале, или при распознавании печатного текста, или при точечных мутациях, если речь идет о биологических «строках», таких как цепочки ДНК). повреждённые символы нельзя идентифицировать однозначно, но для каждого из них можно указать множество символов исходного алфавита, из которых в результате повреждения мог получиться данный символ. Таким образом, каждому символу алфавита повреждённых строк поставлено в соответствие некоторое множество символов исходного алфавита, т.е. между этими алфавитами определено бинарное отношение — *отношение совместимости*. Две строки совместимы, если их длины равны и буквы, стоящие в одинаковых позициях, совместимы.

Рассмотрим пример. Пусть есть текст на русском языке. Текст был распечатан на струйном принтере, но при передаче попал под дождь и был повреждён. Предположим, что в повреждённом тексте встретился символ «I». Мы можем сделать вывод, что в данной позиции в исходном тексте могла быть, например, буква «Б», «П», «В», но не могла быть буква «О».

Важным частным случаем повреждённых строк являются *частичные* строки, содержащие повреждённые символы только одного вида — «джокеры», совместимые со всеми символами алфавита. Задачи поиска для повреждённых строк хорошо известны и более сложны, чем для обычных строк; см., например, [11, 19, 38]. Кроме того, модель повреждённых символьных последовательностей изучается в комбинаторике слов, см., например, [29], а работы по частичным словам составляют достаточно большой массив, см., например, [6, 9, 10, 27, 28, 36].

В данной работе рассматриваются две задачи, в которых поиск играет вспомогательную роль, а цель — восстановить повреждённый текст и повреждённый шаблон оптимальным образом. Постановки задач различаются выбором критерия оптимальности.

Задача 1. *Заданы повреждённый текст и повреждённый шаблон, необходимо среди всех возможных вариантов восстановления исходных неповреждённых текста и шаблона выбрать пару, для которой количество вхождений шаблона в текст максимально.*

Задача 2. *Заданы повреждённый текст и повреждённый шаблон, необходимо среди всех возможных вариантов восстановления исходных неповреждённых текста и шаблона выбрать*

пару, для которой минимальна «непохожесть», вычисляемая как сумма расстояний Хэмминга[4] *для всех пар (шаблон, подстрока той же длины в тексте).*

1.2 Предварительные сведения

1.2.1 Понятия из стрингологии

Строкой длины n над алфавитом Σ называется отображение $\{1, 2, \ldots, n\} \mapsto \Sigma$. Длина строки S обозначается $|S|$. Пустую строку обозначим через ε. Запись $S[i]$ означает i-й символ строки S, где $1 \leqslant i \leqslant |S|$. Строка u называется *подстрокой* строки S, если для некоторых x и y выполняется $S = xuy$. Число $|x|+1$ при этом называется *вхождением* строки u в S. Строка может иметь несколько вхождений в другую строку. Если $x = \varepsilon$ (соответственно, $y = \varepsilon$), то u называется *префиксом* (соответственно, *суффиксом*) S. Префикс (суффикс) строки, не равный самой строке, называется *собственным*. Подстроку строки S, начинающуюся с i-го символа и заканчивающуюся j-м символом, будем обозначать $S[i..j]$. Положим $S[i..i-1] = \varepsilon$ для любого i.

Собственный суффикс строки, равный её равный префиксу, называется *гранью* строки. *Префикс-функцией* строки S называется целочисленный массив, i-й элемент которого равен длине максимальной грани префикса $S[1..i]$ строки S, $i = 1, \ldots, |S|$. *Периодом* строки S называется натуральное число t такое, что $S[i] = S[i+t]$ для любого i, $1 \leq i \leq |S| - t$. Легко проверить, что длина строки минус длина ее максимальной грани равна минимальному периоду этой строки.

Напомним, что *палиндром* — строка $S[1..n]$, равная своему *развороту* $S[n]S[n-1]\ldots S[1]$. Подпалиндромом (суффикс-палиндромом) называется подстрока (суффикс), являющаяся палиндромом.

Следующие две несложные леммы очень важны для работы с палиндромами.

Лемма 1.2.1. *[16] Строка S длины n содержит не более одного палиндрома, не содержащегося в ее префиксе $S[1..n-1]$, причем таким палиндромом может быть только максимальный суффикс-палиндром S.*

Лемма 1.2.2. *[25] Любой подпалиндром строки является максимальным суффикс-палиндромом некоторого префикса этой строки.*

[4] *Расстоянием Хэмминга* между строками S и T одинаковой длины называется число $d(S,T) = |\{i \mid S[i] \neq T[i]\}|$.

1.2.2 Модель вычисления Word-RAM

В данной работе во всех задачах мы будем использовать модель RAM (Random Access Memory), т.е. модель с произвольным доступом к памяти. Сама память рассматривается как массив целых чисел и измеряется в «ячейках», т.е. элементах массива. Мы полагаем, что обращение (запись, либо чтение) к ячейке такого массива выполняется всегда за одинаковое время, независимо от позиции ячейки в массиве. Также за константное время выполняются арифметические и логические операции($*, /, +, -$, побитовые ИЛИ, И, $\oplus$, НЕ).

Замечание 1.2.1. *Массив символов можно эмулировать с помощью массива чисел. Поэтому в дальнейшем массивы символов (т.е. строки) мы будем использовать, не уточняя этот момент.*

Мы будем всегда считать, что на вход алгоритму подаётся строка над некоторым алфавитом. Алгоритм считывает входные символы по порядку слева направо. Считывание символа занимает фиксированное время независимо от его позиции и значения. Считывание состоит в записи данного символа в ячейку массиве и переходе к следующему символу. Алгоритм также имеет «выходную ленту» неограниченной длины, на которую за одну операцию можно записать значение, находящееся в любой из ячеек.

Будем считать, что символы алфавита являются целыми числами от 0 до M, где M не более чем в константное число раз превосходит размер входа.

Замечание 1.2.2. *Существуют другие варианты соглашений об алфавите. Например, считать алфавит константным по размеру, либо считать символы абстрактными объектами, для которых определено только сравнение на равенство, либо на равенство и неравенство (если на алфавите задан линейный порядок). Однако в этой работе мы не будем их использовать.*

Помимо вышеперечисленных операций (арифметических, логических, ввода и вывода) алгоритм может использовать оператор if и оператор $while$. Оператор if выполняет некоторое действие при условии истинности некоторого условия. Оператор $while$ выполняет действие несколько раз (возможно 0 или 1), пока условие истинно.

Решением большинства задач будет являться некоторый алгоритм. Для каждого алгоритма будет доказана корректность (т.е. теорема о том, что он выполняет поставленную ему задачу). Кроме того, для всех алгоритмов нас будет интересовать время работы и используемая память. Во всех случаях мы будем стремиться минимизировать одну из этих двух величин.

Введем понятие функции времени работы алгоритма. Пусть f — функция из $\mathbb{N}_0$ в $\mathbb{N}_0$. Введём параметр n. В каждой задаче он будет определяться отдельно. В большинстве случаев это либо длина строки, либо число запросов к некоторой структуре данных. Каждому входу алгоритма соответствует некоторое значение этого параметра. Тогда равенство $f(n_0) = F$ означает, что максимум по числу действий, которые выполняет алгоритм для всех входов со значением параметра $n = n_0$, равен F. Функцию f будем называть временем работы алгоритма от параметра n. Аналогично определяется функция памяти, используемой алгоритмом.

Для времени работы алгоритма и используемой памяти мы будем использовать асимптотические оценки[5]. Для оценок мы используем стандартные асимптотические классы функций, см., напр., [4].

Для неотрицательных функций f, g неотрицательного целого аргумента говорят, что функция f принадлежит классу $O(g)$, если существуют константы c_f и N_f такие, что для любого $n > N_f$ верно неравенство $f(n) \leq cg(n)$. Говорят, что если f принадлежит $O(g)$, то g принадлежит $\Omega(f)$, а если f принадлежит $O(g)$ и f принадлежит $\Omega(g)$, то f принадлежит $\Theta(g)$. Принадлежность f к классу $O(g)$ принято записывать в виде $f = O(g)$ (аналогично для Ω и Θ).

В некоторых задачах нам потребуется оценка в зависимости от двух или трёх величин. Обычно это длины заданных строк и размер алфавита.

1.2.3 Типы алгоритмических задач

Оффлайн и онлайн

Если задача решается *оффлайн*, решающему алгоритму разрешено вначале выполнить чтение всей входной строки, а затем приступить к обработке и выдаче результатов. В случае решения *онлайн*, условия значительно более жесткие: после прочтения очередного символа алгоритм должен выдать ответ для прочитанной к настоящему моменту строки до того, как читать следующий символ. Таким образом, задача решается для всех префиксов входной строки. Такая модель особенно важна в алгоритмах, использующихся на практике, когда, например, к серверу поступает множество запросов и данные нужно обновлять «на лету», а не выполнять сразу много запросов «оптом».

[5]При сравнении алгоритмов из одного класса сложности мы иногда будем сравнивать константы при старшем члене функций.

Далее мы говорим об оффлайновых и онлайновых алгоритмах в зависимости от вида решения, которое они дают.

Общая оценка и оценка на запрос

Для оффлайновых алгоритмов единственный способ измерять скорость работы алгоритма — это измерить число действий, необходимых для выполнения алгоритма.

Для онлайновых алгоритмов помимо замерения общего времени работы появляется смысл измерять время работы на один запрос (максимальное по всем запросам). Мы столкнёмся с несколькими алгоритмами, требующими для выполнения линейного времени, что означает, что большинство запросов будет выполнено за константное время. Однако некоторые запросы могут потребовать большего (вплоть до линейного) времени. Поэтому для онлайновых алгоритмов часто ставится задача ограничить время работы для каждого запроса.

1.2.4 Структуры данных

Структура данных — это набор данных, на котором поддерживается определённый набор возможных запросов к нему. Например, структура данных «массив целых чисел» позволяет хранить набор чисел фиксированного размера, имеет запросы на изменение одного любого элемента по номеру и новому значению, либо получение значения одного любого элемента по его номеру.

Структуры с откатами

Для структур данных часто имеют интерес их модификации, позволяющие производить «откат» (т.е. отмену последнего действия или группы действий). Очевидно, что любую структуру можно сделать структурой с «откатами», достаточно при каждом запросе строить структуру заново. Однако, во многих задачах это будет недостаточно быстро. Под структурой с «откатами» мы будем подразумевать структуру, позволяющую выполнить запрос «отменить последнее действие» за приемлемое время (отдельно определяемое для каждой задачи).

Персистентные структуры

Более универсальными структурами данных являются персистентные структуры. Такая структура создается для сохранения последовательных изменений в «обычной» структуре данных,

например, массиве или дереве, и представляет собой набор пронумерованных версий этой структуры. Персистентная структура должна быстро отвечать на запросы, адресованные к любой из существующих версий и быстро создавать новую версию всякий раз, когда приходит запрос на изменение имеющейся версии. Персистентные структуры имеют многочисленные применения, например, в базах данных.

1.2.5 Исходные коды программ

В данной работе довольно часто будет использоваться запись алгоритма в виде псевдокода. За основу псевдокода взят язык *c* с некоторыми упрощениями. К примеру, для присваивания мы будем использовать знак «=», для сравнения на равенство знак «==», также повсеместно будет использоваться тернарный оператор for.

1.2.6 Используемые структуры данных

Бор

Бором называется корневое дерево, каждое ребро которого помечено символом, причём для любой вершины все исходящие ребра помечены разными символами. Дополнительно, некоторые вершины бора помечены как терминальные.

Сжатый бор — это корневое дерево, каждое ребро которого помечено непустой строкой таким образом, чтобы ни у одной вершины не было двух детей, к которым ведут рёбра, помеченные строками с одинаковой первой буквой. Сжатый бор набора строк — это минимальный по количеству вершин сжатый бор, в котором каждую строку из набора можно прочитать на пути от корня к какой-либо вершине.

Суффиксный массив

Пусть на алфавите Σ зафиксирован линейный порядок; тогда строки над Σ линейно упорядочены отношением лексикографического (словарного) порядка.

Суффиксный массив строки S — это массив, состоящий из всех суффиксов строки S, отсортированных лексикографически. На практике суффиксный массив задается как массив ссылок на суффиксы: $SA[i] = j$ означает, что суффикс $S[j..|S|]$ имеет номер i в отсортированном лексикографически списке суффиксов. Параллельно с суффиксным массивом строки обычно

вычисляют массив наибольших общих префиксов LCP: $LCP[i]$ есть длина наибольшего общего префикса суффиксов $SA[i]$ и $SA[i+1]$, соседних в суффиксном массиве. Вычислить оба массива для строки S можно за время $O(|S|)$ [30, 39].

Суффиксное дерево

Суффиксное дерево — это сжатый бор всех суффиксов строки. Его можно построить за время $O(n \log |\Sigma|)$ онлайн [49].

И суффиксное дерево, и суффиксный массив широко используются как «индексаторы» текстов, т.е. как структуры, позволяющие осуществлять быстрые запросы о подстроках данного текста. Помимо них есть и другие подобные структуры, но в данной работе мы пользовались только этими двумя.

Массив радиусов

Палиндром называется чётным (нечётным), если имеет чётную (нечётную) длину. Радиусом палиндрома называется его длина, делённая пополам и округлённая вверх до целого. Центром подпалиндрома называется среднее арифметическое номеров первой и последней его позиций. Радиусом в данной точке c (целой или полуцелой) строки S называется наибольший из радиусов подпалиндромов в S, имеющих центр c.

Массив радиусов — структура данных, основанная на алгоритме, сформулированном в [35]. Он позволяет находить радиусы палиндромов по их центру, а также поддерживать максимальный суффикс-палиндром. Массив радиусов внутри себя поддерживает радиусы для чётных и нечётных палиндромов, а также хранит информацию для ответов на несколько типов запросов:

1. За $O(1)$ выдать максимальный суффикс-палиндром.
2. За $O(1)$ выдать радиус по данному на вход центру палиндрома.
3. Дописать новый символ справа к строке. В худшем случае данный запрос будет работать за линейное время, однако суммарно на n дописываний потребуется $O(n)$ времени.

Сбалансированные деревья поиска

Рассмотрим подвешенное бинарное дерево (каждая вершина имеет от нуля до двух детей), каждой вершине которого поставлено в соответствие некоторое целое число, при этом все числа в вершинах попарно различны. Будем называть такое дерево *деревом поиска*, если число в каждой вершине больше всех чисел в вершинах левого поддерева и меньше всех чисел в вершинах правого поддерева. Название обусловлено тем, что для ответа на запрос «есть ли число x в вершинах данного дерева?» достаточно простого спуска от корня к листу. Таким образом, поиск элемента выполняется за $O(h)$, где h — высота дерева. Назовём дерево поиска *сбалансированным*, если оно на каждом шаге имеет высоту $O(\log n)$, где n — число вершин в дереве. Добавление и удаление вершины с сохранением сбалансированности можно осуществить (см. [4]) за время $O(\log n)$.

Сбалансированным деревом с дополнительной информацией будем называть дерево, каждой вершине которого сопоставлено два параметра — ключ и значение. Ключ — это число; по ключам дерево является деревом поиска. Значение является любой дополнительной информацией без налагаемых ограничений. В таком дереве запрос поиска возвращает значение в данной вершине, либо *null*, если вершины с запрошенным значением ключа нет в дереве.

В данной работе нам понадобятся помимо, обычных деревьев поиска, персистентные деревья поиска, хранящие историю своих версий. Известно [15], что все запросы на изменение в персистентных деревьях выполняются за $O(\log n)$, требуя дополнительного выделения $O(\log n)$ памяти[6], остальные запросы выполняются за $O(\log n)$ времени и не требуют дополнительной памяти.

1.3 Обзор результатов диссертации

Все результаты в данной работе разбиты на две большие главы $(2, 3)$. Глава 2 посвящена алгоритмам, связанным с палиндромами, глава 3 — алгоритмам, связанным с повреждёнными строками.

[6]Пусть, например, требуется добавить вершину в существующую версию. Новая версия создается так. Вначале делается копия исходной версии, для чего создается новый корень и соединяется с сыновьями старого корня. В полученном дереве осуществляется добавление вершины, при этом для вершин, затронутых изменениями (их $O(\log n)$), создаются, как и для корня, копии, а оставшаяся часть дерева не меняется.

1.3.1 Цели, задачи и основные результаты

Целями диссертации являются классификация ряда комбинаторных задач о строках по их вычислительной сложности, а также построение теоретически и практически эффективных алгоритмов решения этих задач.

Задачами диссертации являются:
— построение алгоритмов низкой вычислительной сложности для ряда задач о палиндромах в строках, включая подсчет различных палиндромов строки, подсчет числа планидромно-насыщенных строк, разбиение строки на заданное число палиндромов;
— классификация задач об оптимальном восстановлении поврежденных строк по их вычислительной сложности и построение эффективных алгоритмов для полиномиально разрешимых вариантов этих задач.

На защиту выносятся следующие основные результаты:

— новая структура данных «овердрево» («eertree») для быстрого решения различных задач о палиндромах в строках, алгоритмы построения этой структуры и ее специализированных версий [44];
— два алгоритма решения задачи о числе различных палиндромов в строке [33, 44];
— два алгоритма решения задачи о разбиении строки на заданное число палиндромов [34, 44];
— алгоритм решения задачи о разбиении строки на минимально возможное число палиндромов [44];
— алгоритм перечисления палиндромно-насыщенных строк [44];
— алгоритмы решения частных случаев и доказательство NP-трудности общего случая задачи о восстановлении повреждённых строки и шаблона с максимизацией числа вхождений [5];
— алгоритмы решения частных случаев и доказательство NP-трудности общего случая задачи о восстановлении повреждённых строки и шаблона с минимизацией суммарного расстояния Хэмминга [5].

1.3.2 Научная новизна, значимость и корректность результатов

Все результаты, приведенные в диссертации, являются новыми. Диссертация носит теоретический характер. Результаты могут быть использованы для дальнейших научных исследований в области алгоритмов на строках, в частности, по сформулированным в диссертации открытым вопросам, связанным с палиндромами и повреждёнными строками. Также можно применять полученные результаты при обучении алгоритмам. Например, структура данных «овердрево» может предварять тему «суффиксное дерево». Овердрево уже сейчас входит в программы курсов и спецкурсов по алгоритмам ряда российских вузов, в том числе СПб АУ, МФТИ, ВШЭ. Кроме того, все алгоритмы и структуры данных, предложенные в диссертации, просто и эффективно реализуются в виде программного кода, что делает их практически применимыми, например, для постановки вычислительных экспериментов. Все результаты диссертации, в том числе корректность всех алгоритмов и оценки их вычислительной сложности, снабжены строгими математическими доказательствами.

1.3.3 Основные методы исследования

В данной работе большинство результатов — это алгоритм, либо структура данных. Для каждого алгоритма или структуры приведено описание, его скорость и используемая память, а также доказательство корректности, скорости работы и используемой памяти. Остальные результаты — доказательство принадлежности задачи к тому или иному классу сложности. В диссертации использованы различные методы теории алгоритмов, теории вычислительной сложности, стрингологии и комбинаторики слов. Разработан новый метод решения задач, связанных с палиндромами, основанный на предложенной автором эффективной структуре данных для хранения информации о палиндромах.

1.3.4 Апробация и публикации

Все представленные результаты опубликованы в 4 статьях автора (в соавторстве с другими авторами):

1. [5] (в соавторстве с Гамзовой Ю. В.). Опубликована в Сибирских Электронных Математических Известиях; журнал входит в список ВАК

2. [44] (в соавторстве с Шуром А. М.). Опубликована в трудах 26-й международной конферен-

ции по комбинаторным алгоритмам (IWOCA 2015, Верона, Италия) в серии Lecture Notes in Computer Science (индексируется Scopus).
3. [33] (в соавторстве с Шуром А.М. и Косолобовым Д.А.). Опубликована в трудах 41-ой международной конференции по современным тенденциям в теории и практике компьютерных наук (SOFSEM 2015, Пец-под-Снежкой, Чехия) в серии Lecture Notes in Computer Science (индексируется Scopus)
4. [34] (в соавторстве с Шуром А.М. и Косолобовым Д.А.). Опубликована в трудах Пражской конференции по стрингологии (PSC 2013, Прага, Чехия), проиндексирована Scopus

Дополнительно, совместная с А.М. Шуром статья [45], принятая к печати в журнале «Fundamenta Informaticae», содержит результаты экспериментов автора, проведенных с помощью программной реализации результатов диссертации.

Часть результатов опубликована в тезисах:
1. [42] (в соавторстве с Гамзовой Ю.В.)
2. [43] (в соавторстве с Шуром А.М.)
3. [32] (в соавторстве с Косолобовым Д.А.)

Результаты, приведенные в диссертации, докладывались на международной конференции по комбинаторным алгоритмам (IWOCA 2015), на российско-финском симпозиуме по дискретной математике (RuFiDim 2012), на днях стрингологии в Лондоне/Лондонском алгоритмическом семинаре (LSD & LAW 2016), на днях компьютерных наук в Екатеринбурге (CSEdays2013), а также на семинаре «Алгебраические системы» (УрФУ, рук. Шеврин Л. Н., 2015–2016).

Шуру А.М. принадлежит постановка задачи и оптимизация некоторых доказательств в работах [33,34,44], Гамзовой Ю.В. принадлежит постановка задачи и общая методика исследования в работе [5], Косолобову Д.А. принадлежит нижняя оценка в работе [34], а также основной алгоритм и доказательство его корректности и эффективности в работе [33]. Автору принадлежат алгоритмы и доказательства основных результатов в работах [5, 44], основной алгоритм в [34] и первоначальная конструкция алгоритма из [33].

1.4 Благодарности

Автор благодарит Юлию Васильевну Гамзову (УрФУ), своего первого научного руководителя, привившего вкус к научным исследованиям, Арсения Михайловича Шура (УрФУ), руководителя данной работы, внёсшего большой вклад в постановку многих задач, получение новых результатов и в написание статей. А также Дмитрия Косолобова (УрФУ) за совместную работу над статьями, подтолкнувшую к получению других результатов, Григория Назарова (УрФУ), Олега Меркурьева (УрФУ) и Александра Кулькова (МФТИ) за ценные обсуждения.

Глава 2

Палиндромы

2.1 Введение

Данная глава посвящена изучению палиндромов в строках. В ней мы введём новую структуру данных для эффективной и удобной работы с палиндромами, а также решим с помощью неё задачи о поиске различных подпалиндромов, о разбиении на палиндромы, а также много других.

В разделе 2.2 данной работы мы начнём с решения задачи поиска подпалиндромов онлайн. Сначала приведём решение, основанное на известных тяжёлых (использующих много памяти) структурах данных, после этого покажем принципиально новое эффективное решение, имеющее такую же асимптотику, но значительно меньшие константы времени и памяти. Далее выделим основные идеи этого решения и сформулируем в виде новой эффективной структуры данных по работе с подпалиндромами строки — овердрева. В конце раздела обсуждаются некоторые свойства овердрева, а также иллюстрируются возможности его применения на примерах задач с международных соревнований по программированию.

Раздел 2.3 посвящён обсуждению применимости овердрева в различных известных задачах. Будет рассмотрена ситуация, когда нужно решать задачи о подпалиндромах сразу для нескольких строк и описана подходящая версия овердрева. Мы покажем, как получить овердрево с хорошей оценкой времени работы на один запрос. Это даст нам возможность решать задачи о палиндромах в более сложных структурах данных, чем строка, а именно, в строках с откатами (допускающих операцию удаления, а не только добавления символа) и персистентных строках (деревьях, хранящих всю историю изменений строки). При помощи модернизированного таким образом овердрева удалось решить задачу эффективного перечисления палиндромно-

насыщенных строк.

Раздел 2.4 содержит новые решения задач о разбиении строки на k палиндромов и о разбиении на наименьшее число палиндромов. Для второй задачи решение с той же самой асимптотикой существовало раньше, но решение с помощью овердрева проще и имеет меньшие константы по времени и по памяти. В конце раздела выдвигается и обосновывается гипотеза о возможности решения данных задач за линейное время.

2.1.1 Определения и обозначения

Напомним, что мы называем *подпалиндромом* подстроку-палиндром заданной строки, а чётными (нечётными) палиндромами — палиндромы с чётной (нечётной) длиной. Для подпалиндрома $S[l..r]$ его *центром* называется число $(l+r)/2$, а радиусом — число $\lceil (r-l+1)/2 \rceil$. Также нам понадобятся термины *суффикс-палиндром* и *префикс-палиндром* — это, соответственно, суффикс и префикс, являющиеся палиндромами.

2.2 Овердрево

2.2.1 Побуждающая задача: поиск подпалиндромов онлайн

Массив радиусов, который строится алгоритмом Манакера [35] и описан во введении, позволяет найти все подпалиндромы заданной строки (а их количество равно длине массива плюс сумма всех значений в нем). Другая интересная задача — нахождение всех *различных* подпалиндромов в строке. Известно, что число различных непустых подпалиндромов не превосходит n, где n — длина строки (см. лемму 1.2.1) и известен линейный от n алгоритм нахождения их всех для случая целочисленного алфавита (см. [25]).

В [25] была поставлена проблема нахождения оптимального онлайнового алгоритма, подсчитывающего число различных подпалиндромов в строке.[1]

[1]Отметим, что описанные ниже алгоритмы для решения этой задачи позволяют найти не только количество палиндромов, но и сами палиндромы (их можно возвращать в компактном виде, например, в виде указателей на подстроки входной строки).

Решение с помощью стандартных структур

Палиндромным замыканием строки S называется такая строка P минимальной длины, что S является префиксом P и P является палиндромом.

Решим вначале более простую задачу: *привести онлайновый алгоритм нахождения длины палиндромных замыканий всех префиксов строки.* Эта задача также поставлена в [25] и, как мы увидим ниже, близка к задаче онлайнового нахождения числа различных подпалиндромов строки.

Пусть $S = uv$ и $v = \overleftarrow{v}$ — максимальный суффикс-палиндром строки S. Тогда легко видеть, что $uv\overleftarrow{u}$ — это палиндромное замыкание строки S. Таким образом, для решения задачи о палиндромных замыканиях префиксов достаточно найти максимальные суффикс-палиндромы всех префиксов. Оффлайновый алгоритм, решающий эту задачу, описан, например, в [13, гл. 8]. А алгоритм Манакера [35] является онлайновым алгоритмом, ищущим максимальные суффикс-палиндромы префиксов.

Опишем версию реализации массива радиусов (см. 1.2.6). Пусть Δ — константа, равная нулю в случае поиска нечётных палиндромов и единице в случае чётных. Будем поддерживать на строке S операцию добавления символа в конец (add(c)) и значение длины максимального чётного/нечётного суффикс-палиндрома (maxSufPal — длина чётного/нечётного в зависимости от значения константы $\Delta = 1/0$).

Для работы структуры данных понадобятся следующие внутренние переменные:
n — текущая длина строки;
r — массив целых чисел, в j-й ячейке которого содержится максимальный радиус нечётного палиндрома с центром в j в случае $\Delta = 0$ и чётного палиндрома с таким центром в случае $\Delta = 1$;
mid — центр максимального нечётного/чётного суффикс-палиндрома строки $S[1..n]$ в соответствии с $\Delta = 0/1$.

Вообще говоря, центр чётного палиндрома — это нецелое число, но мы для возможности работы с массивом, индексируемым целыми числами, будем округлять его в меньшую сторону.

Изначально полагаем массив r заполненным нулями, $n = 1$, $mid = 2$, $S =$ "\$", где "\$" — символ, нигде более не встречающийся.

В каждый момент времени будет соблюдаться следующий инвариант: все элементы массива r, которые находятся левей mid, заполнены и уже не будут меняться. В самом деле, палиндромы

с центрами левей *mid* не являются суффикс-палиндромами, а значит могут быть вычислены к этому времени. В позиции *mid* между запросами находится радиус максимального суффикс-палиндрома, а позиции правей *mid* заполнены нулями.

При дописывании нового символа алгоритм сначала пытается расширить палиндром с центром в *mid* так, чтоб он стал суффикс-палиндромом. В случае неудачи алгоритм движется по массиву радиусов от *mid* вправо, инициализирует очередной элемент *mid*, используя значение из позиции, симметричной данной относительно *mid*, и пытается увеличить радиус палиндрома с данным центром. Процесс продолжается, пока не найдется центр, для которого такое увеличение удастся; это и будет максимальный суффикс-палиндром (заданной четности) новой строки.

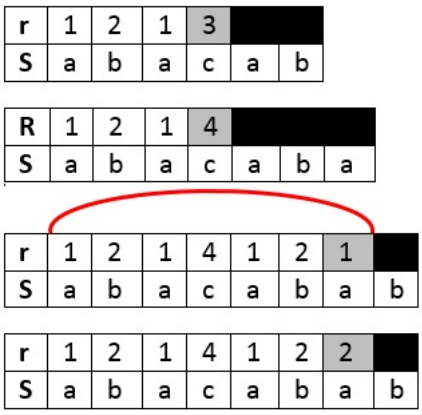

Рис. 2.1: Пример вычисления массива радиусов нечетных палиндромов для строки *abacab*, затем *abacaba* и *abacabab*

```
void add(c)
        S[n] = c
        S[n + 1] = '$'
        while(S[i - r[i]] ≠ S[i + r[i] + Δ])
                i++;
                r[i] = min(r[mid - (i - mid)], n - i - 1);
        r[i]++;
        mid = i;
        n++;
void maxSufPal()
        return ( n - mid - 1 ) * 2 + 1 -Δ
```

Лемма 2.2.1. *[13, лемма 8.1]*

Если k таково, что $1 \leqslant k \leqslant r[i]$ и $r[i-k] \neq r[i]-k$, то $r[i+k] = \min(r[i-k], r[i]-k)$.

Следствие 2.2.1. *Существует линейный онлайновый алгоритм для нахождения длин максимальных суффикс-палиндромов всех префиксов строки.*

Доказательство. Корректность чётного и нечётного случаев вытекает из алгоритма Манакера (см. [35]) и леммы 2.2.1. Для доказательства временной сложности достаточно заметить, что каждая итерация цикла увеличивает i на единицу. □

Пример. Рассмотрим строку $P = abadaadcaa$. После выполнения последовательных вызовов add($P[i]$) при $i = 1, 2, \ldots, |P|$ выполняется равенство $S = P$ и массив r равен $\{1, 2, 1, 2, 1, 1, 1, 1, 1, 1\}$ (для чётного случая $\{0, 0, 0, 0, 2, 0, 0, 0, 1, 0\}$). Если после каждого добавления символа мы будем запрашивать максимальный суффикс-палиндром (maxSufPal), то получим последовательность $1, 1, 3, 1, 3, 1, 1, 1, 1, 1$ (для чётного случая $0, 0, 0, 0, 0, 2, 4, 0, 0, 2$).

Предложение 2.2.1. *Существует линейный онлайновый алгоритм для нахождения длин палиндромных замыканий всех префиксов строки.*

Доказательство. Вытекает из следствия 2.2.1 и того факта, что палиндромное замыкание строки uv с максимальным суффикс-палиндромом v равно $uv\overleftarrow{u}$. □

Вернемся к задаче онлайнового подсчета различных подпалиндромов строки. Дальнейшее рассуждение будет опираться на лемму 1.2.2.

Из леммы очевидно, что для решения задачи об онлайновом подсчёте достаточно поддерживать структуру данных, позволяющую на каждом шаге алгоритма из предложения 2.2.2 определять, встречался ли раньше в строке найденный для данного префикса максимальный суффикс-палиндром. Такой структурой данных нам будет служить *суффиксное дерево Укконена* (см. [49]).

Суффиксным деревом (или просто деревом) Укконена будем называть структуру данных Ukkonen, поддерживающую строку и соответствующее ей суффиксное дерево, а также запрос Ukkonen.add(c) для добавления символа в конец строки. Помимо этого, нам понадобится реализовать запрос Ukkonen.minUniqueSuff, возвращающий длину минимального суффикса текущей строки, не имеющего других вхождений в эту строку. Для того, чтобы понять, как реализовать Ukkonen.minUniqueSuff эффективно, необходимо рассмотреть внутреннюю реализацию дерева Укконена.

Дерево Укконена поддерживает тройку (v, e, i), где v — вершина суффиксного дерева, e — ребро, помеченное строкой T и ведущее от v к некоторому дочернему узлу, i — число от 0 до $|T|$, причём строка T' на пути от корня к v такова, что $P = T'T[1..i]$ образует самый длинный суффикс S, имеющий более одного вхождения в S (см. [49]). Несложно модифицировать дерево Укконена так, чтобы оно поддерживало для каждой вершины u суффиксного дерева поле u.depth, равное длине строки, по которой можно дойти из корня до u. Отсюда очевидно, что Ukkonen.minUniqueSuff $= v$.depth $+ \mathrm{i} + 1$.

Пусть Manacher — структура данных, описанная в предложении 2.2.2 и поддерживающая запросы Manacher.add(c) для добавления символа в конец строки и Manacher.maxSufPal для получения максимального суффикс-палиндрома текущей строки.

Предложение 2.2.2. *Пусть Σ — это алфавит. Существует онлайновый алгоритм для подсчёта числа всех различных подпалиндромов строки, работающий за время $O(n \log |\Sigma|)$.*

Доказательство. Каждый вызов add(c) в псевдокоде ниже добавляет символ c к текущей строке, хранимой в структурах Manacher и Ukkonen (обе структуры содержат одну и ту же строку) и выводит первую и последнюю позиции нового появившегося подпалиндрома. В начале работы $n = 0$.

```
void add(c)
        n++
        Manacher.add(c)
        Ukkonen.add(c)
        if(Ukkonen.minUniqueSuff <= Manacher.maxSufPal)
                printPair(n - Manacher.maxSufPal + 1, n)
```

Из предложения 2.2.2 и того, что n запросов Ukkonen.add(c) работают за время $O(n \log |\Sigma|)$ (см. [49]), следует оценка времени работы алгоритма. □

Пример. Для той же строки $S = abadaadcaa$ запрос Manacher.maxSufPal последовательно возвратит следующие значения: $1, 1, 3, 1, 3, 2, 4, 1, 1, 2$. Запрос Ukkonen.minUniqueSuff будет возвращать следующие значения: $1, 1, 2, 1, 2, 2, 3, 1, 2, 3$. Соответственно результат работы алгоритма будет следующий: 1 1; 2 2; 1 3; 4 4; 3 5; 5 6; 4 7; 8 8;.

Более экономное по памяти решение

Алгоритм, приведённый в предыдущем пункте, описан в [33]. В той же работе была доказана эквивалентность этой задачи одному из видов словарей, что доказывает неулучшаемость временной оценки в модели, основанной на сравнениях (comparison model). Несмотря на хорошую асимптотику, алгоритм имеет недостаток — он опирается на две довольно громоздкие структуры данных. Это отражается и в большом расходе памяти (для строки длины n структура Manacher использует $2n$ ячеек памяти, а структура Ukkonen — $8n$ ячеек[2] + словарь с дополнительной памятью в каждой вершине[3]), и в большом и сложном коде. Естественно пытаться построить «легкую» структуру данных, которая позволила бы решить ту же задачу с той же асимптотикой, но существенно меньшими константами в O-выражениях, а также с более простым и коротким кодом. Структура «овердрево», описанная в следующем разделе, решает эту задачу. При дальнейшем исследовании этой структуры оказалось, что она универсальна и дает возможность эффективно решать широкий спектр задач о палиндромах.

2.2.2 Интерфейс структуры данных

В базовой версии мы поддерживаем всего один вид запроса:
$add(c)$ — функция, которая приписывает к концу строки символ c, соответствующим образом обновляет содержимое структуры данных и сообщает нам число новых палиндромов, которые появились в строке. Согласно лемме 1.2.1 add всегда возвращает 0 или 1.

2.2.3 Внутреннее устройство структуры данных

Структура реализована в виде ориентированного графа. Вершины графа занумерованы натуральными числами и взаимно однозначно соответствуют различным подпалиндромам обработанной строки, так что мы далее отождествляем вершины с палиндромами, используя для них

[2]Всего в дереве $2n$ вершин, каждая вершина хранит два индекса концов подстроки, которой помечено ребро, ведущего в данную вершину, а также ссылку на родителя и суффиксную ссылку, т.е. 4 целых числа в каждой вершине

[3] Общее количество элементов в словарях равно $2n$; если реализовывать их как бинарные сбалансированные деревья, для каждой вершины потребуется одна ячейка для значения (ссылки на вершину дерева) и $4 \log \|\Sigma|$ битов для хранения ключа и ссылок на родителя и детей в дереве. Например, при $|\Sigma| = 2^8$ и $n = 2^{32}$ все словари занимают $4n$ ячеек памяти.

одну и ту же переменную.

Замечание 2.2.1. *Наибольший номер вершины в графе равен числу различных подпалиндромов обработанной строки(см.п. 2.2.1).*

В каждой вершине хранится длина соответствующего палиндрома. Дополнительно вводятся две специальные вершины — с номером 0 и длиной 0 для пустой строки, и с номером -1 и длиной -1 для «мнимой строки», роль которой объяснена ниже.

Ребра графа определены следующим образом: если x — символ, A и xAx — две вершины графа, то из A в xAx ведет ребро с меткой x. При этом из вершины 0 ребро с меткой x ведет в вершину xx, а из вершины -1 — в вершину x (при наличии соответствующих вершин в графе). Для хранения исходящих ребер для каждой вершины v используется словарь $to[v]$ с дополнительной информацией, который по символу алфавита c возвращает номер вершины $to[v][c]$, в которую ведёт ребро, помеченное данным символом. Такой словарь можно реализовать с помощью бинарного сбалансированного дерева поиска (см. раздел 1.2.6) с временем доступа $O(\log|\Sigma|)$.

Не имеющая метки *суффиксная ссылка* ведет из A в B (обозначается $link[A] = B$), если B является длиннейшим собственным суффикс-палиндромом A. Суффиксная ссылка из любого палиндрома длины 1 ведет в вершину 0 ($link[c] = 0$), а из вершин 0 и -1 — в вершину -1 ($link[0] = link[-1] = -1$). Построенный таким образом граф будем называть *овердревом* строки S (в виду своей связи с деревьями и палиндромами) и обозначать $eertree(S)$ (англ. eerie tree — сверхъестественное дерево). Пример овердрева приведён на рис. 2.2.

2.2.4 Простой онлайн-алгоритм

Лемма 2.2.2. *В вершину овердрева входит не более одного ребра.*

Доказательство. Несложно видеть, что любое ребро помечено символом, которая совпадает с первым символом палиндрома, в которую оно ведёт. Значит все входящие рёбра помечены одним символом. Пусть в некоторую вершину входит хотя бы два ребра (пусть они ведут из вершин A и B), значит оба ребра помечены одинаковыми символами (пусть x). Если $C = x$, то по построению в C входит только ребро из вершины «-1». Иначе $C = xAx = xBx$, что влечёт $A = B$, что противоречит построению графа. □

Предложение 2.2.3. *Овердрево строки S длины n занимает $O(n)$ памяти.*

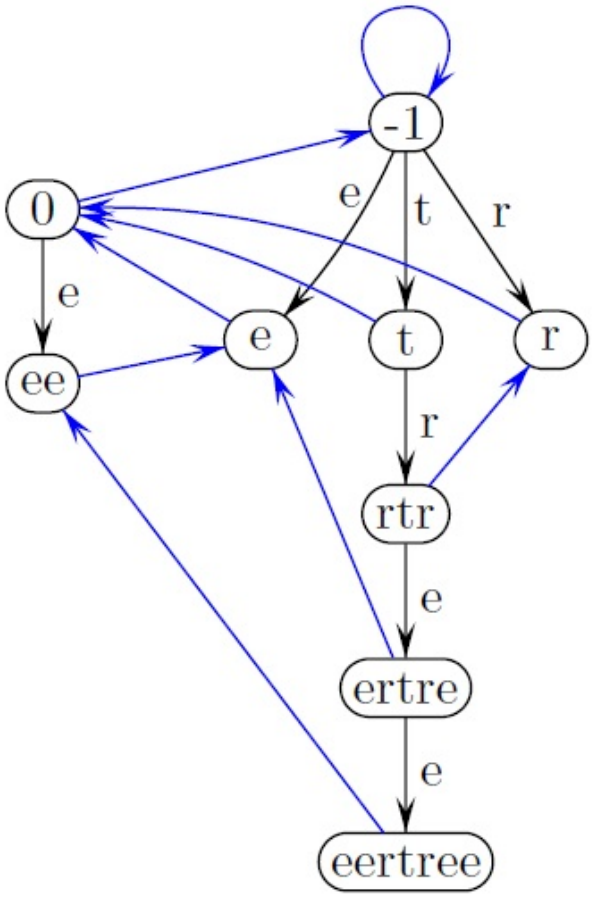

Рис. 2.2: Овердрево для «eertree». Чёрные стрелки — рёбра, синие — суффиксные ссылки.

Доказательство. По лемме 1.2.1, в дереве не более n вершин, не считая двух специальных. Из вершины выходит одна суффиксная ссылка и входит (по лемме 2.2.2) не более чем одно ребро. □

Предложение 2.2.4. *Овердрево строки S длины n над алфавитом Σ можно построить онлайн за время $O(n \log |\Sigma|)$.*

Доказательство. Опишем алгоритм построения овердрева. Он состоит из инициализации (овердрево из двух специальных вершин 0 и -1 и двух суффиксных ссылок в -1) и n вызовов функции *add*. Поддерживаем следующий инвариант: между вызовами функции *add* все ребра и суффиксные ссылки между существующими вершинами проведены. В таком случае при добавлении вершины появляются только ребра/ссылки, инцидентные этой вершине.

С учётом леммы 2.2.2 при добавлении вершины функция *add* должна построить одно ребро и одну суффиксную ссылку. Опишем процедуру создания вершины и инцидентных ей ребра и

суффиксной ссылки.

Пусть у нас есть строка S. К ней приписали символ a. Рассмотрим максимальный суффикс-палиндром P строки Sa. Тогда $P = a$ или $P = aQa$, где aQ — суффикс S и Q — палиндром. Значит поиск максимального суффикс-палиндрома строки Sa равносилен поиску самого длинного суффикс-палиндрома в S, которому предшествует a. Переходя по суффиксным ссылкам, перебираем все суффикс-палиндромы строки S в порядке убывания длины, пока не встретим такой, которому предшествует a. Если такой палиндром Q найден, проверяем для него наличие исходящего ребра с меткой a. Если оно есть, то оно ведет в $aQa = P$ и P не является новым, а если нет, то P — новый, его нужно добавить, присвоить длину $|Q| + 2$ и провести в него ребро из Q. Если же Q не найден (т.е. мы дошли по суффиксным ссылкам до вершины -1), то $P = a$, и мы проверяем наличие в графе вершины a, добавляя ее в случае отсутствия.

Осталось провести суффиксную ссылку из P. Она ведет во второй по величине суффикс-палиндром строки Sa. Его можно получить аналогично P, продолжив сканировать суффикс-палиндромы S, начиная с палиндрома, на который ссылается Q. Таким образом, мы теперь умеем поддерживать и создавать максимальный и второй по величине суффикс-палиндромы (создавать нужно только максимальный, второй создан когда-то ранее).

Оценим асимптотику работы алгоритма. На каждом шаге мы проверяем наличие ребра из Q с меткой a по словарю за $O(\log |\Sigma|)$. Оценим число переходов по графу. На каждом шаге происходит ровно один переход по ребру (всего n) и ноль или более переходов по суффиксным ссылкам. Переход по ссылке сдвигает левую границу максимального суффикс-палиндрома вправо, а переход по ребру — на единицу влево.

Поскольку за все время работы эта граница сдвинулась не более чем на n позиций вправо, число переходов по ссылкам можно оценить сверху числом $2n$. Эти же свойства верны и для второго по величине суффикс-палиндрома.

Таким образом, общее время работы алгоритма есть $O(n \log |\Sigma|)$. □

2.2.5 Некоторые свойства овердрева

Назовём вершину, которой соответствует чётный (нечётный) палиндром, чётной (соответственно, нечётной). Если из вершины A в вершину B ведёт ребро, то длина B на два больше длины A. Что влечёт:

Замечание 2.2.2.

1. Из чётных вершин eertree(S) не достижимы нечётные и наоборот.

2. Рёбра eertree(S) не образуют циклов.

Из замечания 2.2.2 следует, что построенный нами граф, без учёта суффиксных ссылок, состоит из двух не связанных друг с другом направленных деревьев, с корнями в вершинах 0 и -1 соответственно. При этом можно заметить, что дерево с корнем в вершине 0 — это бор всех правых половин чётных палиндромов, а дерево с корнем в -1 — это в точности бор всех правых половин (включая центр) нечётных палиндромов.

Получили овердрево — это два независимых бора, которые могут быть связаны суффиксными ссылками. Например, строка abaaba является чётным палиндромом, наибольший её суффикс-палиндром — это строка aba, которая является нечётным палиндромом. А значит, суффиксная ссылка ведёт из одного бора в другой. Такие ссылки назовём *перекрёстными.*

Замечание 2.2.3. *Суффиксная ссылка всегда уменьшает длину вершины (за исключением вершины -1). А значит, единственным циклом из суффиксных ссылок является петля на -1.*

Назовем суффиксным путём последовательность вершин, в которой из каждой вершины в следующую (если такая есть) ведёт суффиксная ссылка.

Из любой вершины существует суффиксный путь в -1. Следовательно, верна

Лемма 2.2.3. *Граф, образованный вершинами овердрева и обращёнными суффиксными ссылками, является направленным деревом с петлёй на корне.*

Для дальнейшей работы нам понадобятся дополнительные обозначения. Вся информация о графе, включая дополнительную информацию для конкретных задач, хранится в его вершинах; величину x, хранимую в вершине v, мы обозначаем через $x[v]$. Например, $len[v]$ — длина v, $link[v]$ — суффиксная ссылка из v, $to[v]$ — словарь, в котором $to[v][c]$ равно номеру вершины, в которую ведёт ребро из v по символу c.

Замечание 2.2.4. *Мы можем запретить перекрёстные суффиксные ссылки. Т.е. из каждой вершины ссылаться на максимальный суффикс-палиндром той же чётности. Тогда получим две абсолютно независимые структуры. Назовём их нечётное овердрево и чётное овердрево. Изложенный в доказательстве предложения 2.2.4 алгоритм можно использовать и для построения четного и нечетного овердрева.*

Замечание 2.2.5. *В дальнейшем у нас будут появляться и другие (вспомогательные) рёбра, а также различная дополнительная информация в вершинах. Полученную структуру мы называем модифицированным овердревом.*

Суффиксные структуры данных позволяют хранить информацию о всех подстроках данной строки. Если мы возьмём все суффиксы строки и сложим их в бор, то получим структуру, занимающую $O(n^2)$ памяти. В строковых алгоритмах эту проблему решают разными способами. Два популярных способа заключаются в сжатии такого бора: удалить вершины степени два (сконкатенировав метки рёбер) и получить суффиксное дерево (suffix tree), либо объединить эквивалентные вершины и получить суффиксный автомат (DAWG).

В овердреве хранятся все подпалиндромы строки (т.е. часть подстрок), а их, как известно, всего $O(n)$. Поэтому структура уже изначально требует всего $O(n)$ памяти и не нуждается в сжатии. А значит, овердрево можно считать в некотором смысле и аналогом суффиксного дерева и суффиксного автомата для всех подпалиндромов строки. А в большинстве случаев размер овердрева намного меньше n, поскольку матожидание числа различных палиндромов в строке длины n есть $O(\sqrt{|\Sigma|n})$ [45].

2.2.6 Задачи, иллюстрирующие возможности овердрева

Для иллюстрации возможностей приложения овердрева к решению разнообразных задач мы выбрали три существенно различных задачи о палиндромах, предлагавшиеся недавно на международных соревнованиях по программированию. Все эти задачи позиционировались жюри соревнований как сложные, а одна из них не была решена никем из участников. Мы покажем, как решить эти задачи при помощи овердрева.

Задача «Palindromic refrain»

При обучении суффиксным структурам данных часто используют задачу Refrain (см. напр. [53, Problem G]): для данной строки S найти строку P, максимизирующую величину $|P| \cdot f(S, P)$, где $f(S, P)$ — число вхождений P в S. Также известна модификация этой задачи, когда P должна быть палиндромом (Palindromic Refrain). Эта модификация предлагалась на всеазиатской олимпиаде школьников по информатике [51, Problem A] и подразумевалась авторами, как сложная задача на комбинацию какой-нибудь суффиксной структуры и алгоритма Манакера.

Предложение 2.2.5. *Задача Palindromic refrain решается овердревом с использованием $O(n)$ дополнительного времени и дополнительной памяти.*

Пусть $occ[v]$ — число вхождений палиндрома v в строку S. Тогда ответом задачи является вершина v овердрева, для которой величина $len[v] \cdot occ[v]$ максимальна. Для вычисления occ сопоставим каждой вершине v величину $occAsMax[v]$, равную количеству префиксов S, для которых v является максимальным суффикс-палиндромом. На каждой итерации построения дерева мы находим максимальный суффикс-палиндром, а значит, можем увеличить для него $occAsMax$ на единицу. После построения дерева мы можем вычислить значения occ, основываясь на следующей лемме.

Лемма 2.2.4. $occ[v] = occAsMax[v] + \sum_{u:link[u]=v} occ[u]$.

Доказательство. Если для некоторого i строка $S[1..i]$ заканчивается на v, то либо v — максимальный суффикс-палиндром $S[1..i]$, и это вхождение учтено в $occAsMax[v]$, либо v — максимальный суффикс-палиндром некоторой строки u, являющейся суффикс-палиндромом $S[1..i]$, и тогда $link[u] = v$, а данное вхождение учтено в $occ[u]$. □

Вычислить значения occ в порядке, указанном леммой 2.2.4, можно обходом дерева ссылок снизу вверх:

```
for (v = size; v ≥ 1; v--)
        occ[v] = occAsMax[v]
for (v = size; v ≥ 1; v--)
        occ[ link[v] ] += occ[v]
```

$size$ — размер овердрева, а значит и номер последней вершины. Суффиксную ссылку мы всегда проводим при создании вершины, значит в этот момент вершина, в которую проводится ссылка, всегда уже создана, т.е. имеет меньший номер.

После вычисления occ для всех вершин, $ans = max(occ[v] \cdot len[v])$. □

Задача «Palindromic pairs»

Теперь рассмотрим задачу «Palindromic pairs» [52, Problem B]: для строки S найти количество подстрок, являющихся произведением двух палиндромов, иными словами, количество троек i, j, k таких, что $1 \le i \le j < k \le |S|$, $S[i..j]$ —палиндром, $S[j+1..k]$ — палиндром.

Предложение 2.2.6. *Задача «Palindromic pairs» решается овердревом с использованием $O(n \log |\Sigma|)$ дополнительного времени и $O(n)$ дополнительной памяти.*

За $O(n)$ переберём j и решим задачу за $O(1)$ для фиксированного j, т.е. найдём число подходящих пар (i, k). Несложно видеть, что ответ на задачу равен произведению числа палиндромов, кончающихся в j, на число палиндромов, начинающихся в $j+1$. Значит нам заранее нужно предпросчитать для каждой позиции строки число начинающихся палиндромов и число заканчивающихся. Достаточно найти только число заканчивающихся, а после развернуть строку и решить такую же задачу.

Замечание 2.2.6. *Овердрево развёрнутой строки совпадает с овердревом исходной. Отличается только порядок посещения вершин. Для экономии памяти, после разворота не нужно строить овердрево снова, достаточно посчитать интересующую нас величину в вершинах.*

Итак, нам для каждого символа нужно найти число палиндромов, заканчивающихся в нём. Переформулируем задачу: для каждого префикса нужно найти число его суффикс-палиндромов.

Чтобы найти число суффикс-палиндромов строки, достаточно взять максимальный суффикс-палиндром и выдать количество его суффикс-палиндромов. А так как найти максимальный суффикс-палиндром нам позволяет овердрево, то нам осталось посчитать количество суффикс-палиндромов для каждого подпалиндрома строки, т.е. вершины дерева. Назовём это поле $suffCount$.

Несложно понять, что $suffCount[v] = 1 + suffCount[link[v]]$, а значит мы можем вычислять его прямо при создании вершины и далее он остаётся константой. Далее мы для каждого префикса $S[1..j]$ вычисляем вершину в дереве, являющуюся его максимальным суффикс-палиндромом. Эту вершину обозначим через $suff[j]$, а вершину, являющуюся максимальным префикс-палиндромом строки $S[j..|S|]$ — через $pref[j]$ (её мы вычислим, построив овердрево развёрнутой строки «поверх» уже существующего).

Как было сказано выше, ответ можно получить, просуммировав $suffCount[suff[j]] \cdot suffCount[pref[j+1]]$ по всем подходящим j.

Таким образом, мы решили задачу, использовав овердрево с двумя дополнительными полями. □

Задача «String synthesis»

Задача *Изначально мы имеем пустую строку, далее на каждом шаге мы можем либо приписать символ (слева или справа) к строке, либо развернуть строку и приписать к самой себе справа (реплицировать). Требуется для каждого префикса данной строки выдать минимальное число операций, которое нужно для его получения.*

Эта задача предлагалась на студенческом соревновании по алгоритмам, которое является отборочным на студенческий Чемпионат мира по спортивному программированию (ACM ICPC CERC и не была решена никем из участников, [54]). Жюри предложило довольно трудоемкий метод решения задачи. Мы в этом разделе разберём, как с помощью овердрева решить её существенно проще.

В данном соревновании задачу предлагали в оффлайн, т.е. вычислить ответ для данной строки. Мы же опишем онлайновое решение.

Заметим, что результатом репликации всегда является чётный палиндром. Поэтому для задачи нам может понадобиться только дерево четных палиндромов (см.2.2.4). Далее в данной задаче под палиндромом мы будем подразумевать чётный палиндром.

Предложение 2.2.7. *Задача «String synthesis» разрешима с помощью чётного овердрева за время $O(n \log |\Sigma|)$ онлайн.*

Доказательство. Для решения задачи нам потребуется в каждой вершине овердрева хранить дополнительный параметр $halfv$ — ссылку на длиннейший суффикс-палиндром (а значит и префикс-палиндром) данного палиндрома, длина которого не превышает половины длины v.

Замечание 2.2.7. *При построении овердрева мы поддерживаем максимальный и второй по величине суффикс-палиндромы, причем их пересчет требует, в сумме за все итерации построения, линейного времени. Аналогично, суммарно за линейное время можно найти все палиндромы вида $halfv$.*

Будем использовать метод динамического программирования. Пусть $Ans[S]$ обозначает минимальное количество операций, необходимое для получения строки S. Чтобы решить задачу, надо заполнить массив $prefAns$, где $prefAns[i] = Ans[S[1..i]]$. Дополнительно, в каждой вершине v четного овердрева будем хранить целые числа $palAns[v]$ и $halfAns[v]$, равные, соответственно, $Ans[v]$ и $Ans[u]$, где u — левая половина v, т.е. $v = u\overleftarrow{u}$.

Опишем как в общем будет выглядеть итерация алгоритма после дописывания нового символа:

1. Актуализировали овердрево, т.е. добавили не более одной новой вершины.

2. Если на первом шаге была добавлена новая вершина v, вычислили в ней $halfAns[v]$.

3. Если на первом шаге была добавлена новая вершина v, вычислили в ней $palAns[v]$.

4. Вычислили $prefAns[n]$ (n — текущая длина строки).

Замечание 2.2.8. *Важно, что во время выполнения шага* 4 *все* $palAns$ *и* $halfAns$ *вычислены. При выполнении шагов* 2 *и* 3 *вычислены все* $palAns$ *и* $halfAns$*, кроме как для новой вершины* v*. Значит, их можно использовать при пересчёте новых значений.*

Сформулируем общий метод пересчёта на шагах $2-4$ алгоритма:

1. Для любой строки S в каждой оптимальной стратегии ее получения последним действием была либо репликация, либо дописывание символа (слева или справа). Если мы найдём кратчайшую стратегию, заканчивающуюся репликацией, и кратчайшую стратегию, заканчивающуюся дописыванием, то $Ans[S]$ будет равен минимуму из длин этих стратегий.

2. В случае, если последним действием была репликация, то мы точно знаем, что текущая строка — палиндром. Если мы находимся на шаге 2 или 4, то ответ для этого палиндрома уже вычислен. Если мы находимся на шаге 3, то ответом будет ответ для половинки палиндрома, увеличенный на единицу.

3. В случае, если последним действием было дописывание, пусть x — число дописываний после последней репликации (либо длина стратегии, если репликаций не было). Рассмотрим три случая: за последние x шагов были только дописывания слева, только дописывания справа, либо были и те, и другие. Для решения задачи достаточно найти ответ для всех трёх случаев и взять из них минимум.

4. Для каждого из трёх шагов алгоритма (2, 3, 4) некоторые из трёх случаев предыдущего пункта будут схлапываться в один. Ниже приведена таблица, основанная на лемме 2.2.5 для нахождения ответа для каждого из трёх шагов алгоритма и каждого из трёх случаев предыдущего пункта.

Следующими замечанием и леммой мы будем пользоваться при заполнении таблицы.

Замечание 2.2.9. *Если из строки A была получена строка B только дописываниями, то порядок этих дописываний неважен и можно рассмотреть любой.*

Лемма 2.2.5. *Пусть некоторая оптимальная стратегия получения строки S длины n заканчивается $a + b$ дописываниями, перед которыми была либо репликация, либо пустая строка, причем из этих дописываний a было сделано слева, а b — справа. Тогда если для некоторых i, j таких, что $i-1 \leq a$ и $n-j \leq b$, известен $Ans[S[i..j]]$, то $Ans[S] = Ans[S[i..j]]+(i-1)+(n-j)$.*

Доказательство. Перед последними $a + b$ шагами у нас была оптимальная стратегия построения строки $S[a + 1..n - b]$. По определению i, j и замечанию 2.2.9, последующие дописывания можно упорядочить так, чтобы на одном из шагов получить $S[i..j]$, причем количество шагов к этому моменту равно $Ans[S[i..j]]$ ввиду оптимальности стратегии. Дописав оставшиеся символы в произвольном порядке, получим требуемую формулу. □

Таблица детализирует работу алгоритма на 2-4 шагах. Таким образом, приведённый алгоритм вычисляет массив $prefAns$, для решения задачи остаётся для каждого запроса выдавать $prefAns[n]$. □

2.3 Вариации овердрева и некоторые их приложения

2.3.1 Совместное овердрево для нескольких строк

До этого момента мы везде использовали овердрево как некоторую структуру, позволяющую хранить все подпалиндромы данной строки. Отметим, что ни рёбра, ни суффиксные ссылки не зависят от конкретных строк. Строка задаёт только множество вершин. Вообще говоря, мы можем для данного алфавита взять вообще все палиндромы (которых счётное число) и

	Слева	Справа	оба
$halfAns[v]$	$halfAns[parent[v]] + 1$	$palAns[half[v]] + len[v]/2 - len[half[v]]$	как слева
$palAns[v]$	$palAns[link[v]] + len[v] - len[link[v]]$		$palAns[parent[v]] + 2$
$prefAns[i]$	$palAns[maxSufPal] + n - len[maxSufPal]$	$prefAns[i-1] + 1$	

Таблица 2.1: Детализация алгоритма решения задачи «String synthesis». Более подробно данная таблица разобрана в авторском решении [55]. $parent[v]$ — вершина, из которой ведёт ребро в v

построить овердрево для них (назовём его *общее овердрево*). Овердрево для любой конечной строки можно получить путём удаления лишних вершин (и инцидентных им рёбер/суффиксных ссылок) из общего овердрева.

Когда задача требует сравнения двух или более строк, удобно иметь «совместную» структуру данных, хранящую информацию сразу обо всех строках. Примером такой структуры может служить «совместное суффиксное дерево», приведённое в [2].

Пусть нам нужно построить овердрево для нескольких строк. Можно поступить следующим образом: для первой строки строим овердрево как обычно и проставляем во всех вершинах битовый флаг принадлежности первой строке. Для второй строки овердрево строим «поверх» первого: если очередной максимальный суффикс-палиндром уже существует, отмечаем его флагом принадлежности ко второй строке (а если не существует, создаем его как обычно и помечаем флагом). Для третьей и последующих строк строим аналогично. В результате получаем большую экономию места по сравнению с набором отдельных овердрев, а также возможность быстро решать некоторые специальные задачи (отметим, что «совместные» суффиксные деревья также строят в подобных целях). В таблице 2.2 приведены несколько таких задач и их решения.

Формулировка	Решение
Найти количество строк-палиндромов, которые входят во все k данных строк как подстроки.	Построим совместное овердрево для всех строк. Одним линейным пробегом посчитаем количество вершин, которые помечены флагом всех строк.
Найти самый длинный палиндром, который входит как подстрока во все k данных строк.	Построим совместное овердрево для всех строк. Одним линейным пробегом возьмём максимальную (по длине) вершину среди тех вершин дерева, которые помечены флагом всех строк.
Для данных строк S и P определить количество строк T таких, что T -палиндром, число вхождений T в S больше числа вхождений T в P.	Построим совместное овердрево для строк S и P, и вычислим в его вершинах значения $occS$ и $occP$ (см. задачу $Palindromic\ refrain$ из раздела 2.2.6). В качестве ответа выдадим количество вершин, для которых $occS > occP$.
Для данных строк S и P определить количество равных подпалиндромов, т.е. четвёрок (i, j, k, t) таких, что $1 \le i \le j \le \|S\|$ и $1 \le k \le t \le \|P\|$, $S[i..j] = P[k..t]$, $S[i..j]$ — палиндром.	Аналогично предыдущей задаче построим совместное овердрево для двух строк и посчитаем $occS$, $occP$. Ответом будет $\sum_v occS[v] \cdot occP[v]$.

Таблица 2.2: Несколько задач, решаемых совместным овердревом нескольких строк

2.3.2 Поиск с откатами

В доказательстве предложения 2.2.4 приведён алгоритм построения овердрева с общей оценкой $O(n \log |\Sigma|)$. Но добавление одного символа на некоторых строках может работать за $O(n)$, а в некоторых задачах бывает важна оценка не общая оценка времени работы, а время работы одного конкретного шага. В этом параграфе речь пойдёт о модификациях овердрева, которые работают быстро на каждом добавлении нового символа.

Типичным примером, когда нам нужна оценка на шаг, являются задачи, в которых помимо добавления символа, есть его удаление. Применительно к овердреву, у нас помимо $add(c)$ появляется функция $pop()$, которая удаляет последний символ.

Пример. Рассмотрим последовательность вызовов:

$$\underbrace{add(a), ..., add(a)}_{\text{n / 3 раз}} \underbrace{add(b), pop(), add(b), pop(), ..., add(b), pop()}_{\text{n / 3 раз}}$$

Поскольку каждое добавление символа b потребует $n/3$ переходов по суффиксным ссылкам, алгоритм из предложения 2.2.4 обработает эту последовательность за $\Omega(n^2)$ операций независимо от того, как будет реализована функция $pop()$.

В данном разделе мы разберём две модификации дерева, которые позволяют решать задачу с откатами эффективно.

Поиск суффикс-палиндромов с помощью быстрых ссылок

Рассмотрим пару вершин $(v, link[v])$ в некотором овердреве. Поскольку $link[v]$ является суффиксом v, рассмотрим символ b, который идёт перед $link[v]$ внутри v. Теперь проведём дополнительное ребро. Назовём его быстрой ссылкой ($quickLink$). По определению, $quickLink[v]$ — максимальный собственный суффикс-палиндром палиндрома v такой, что символ перед ним не равен символу b.

Лемма 2.3.1. *При создании вершины v можно вычислить $quickLink[v]$ за время $O(1)$.*

Доказательство. Максимальный суффикс, который может нам подойти, равен $link[link[v]]$. Рассмотрим символы внутри v: b — находящийся перед $link[v]$ и c — перед $link[link[v]]$. Если $b \neq c$, то, по определению, быструю ссылку нужно провести в $link[link[v]]$. Рассмотрим случай, когда $b = c$. $quickLink[link[v]]$ ведёт в максимальный суффикс, символ перед которым отличен от c. А поскольку, $c = b$, то $quickLink[v] = quickLink[link[v]]$. □

```
if ( S[n - len[link[v]]] == S[n - len[link[link[v]]]] )
        quickLink[v] = quickLink[link[v]]
else
        quickLink[v] = link[link[v]]
```

Теперь разберёмся, как использовать эти ссылки. При вычислении максимального суффикс-палиндрома строки после добавления к ней символа c мы выполняем спуск по суффиксным ссылкам в поиске символа c перед суффикс-палиндромом.

Пусть мы должны спуститься из вершины v и обнаружили, что символ перед $link[v]$ отличен от c, тогда если мы перейдём по $quickLink$, то точно не пропустим c (по определению быстрой ссылки). Значит такой алгоритм корректен.

В нашем алгоритме теперь мы спускаемся по суффиксному пути, пропуская некоторые вершины. Назовём такой путь быстрым суффиксным путём. Таким образом, новый алгоритм асимптотически не хуже старого. Значит общая оценка времени работы сохранилась. Рассмотрим число шагов при добавлении одной конкретной вершины. Утверждения о «рядах» палиндромов, аналогичные следующему предложению, доказаны в ряде работ, см., например, [34, леммы 5,6] и [17, лемма 5].

Предложение 2.3.1. *В овердреве строки длины n любой быстрый суффиксный путь имеет длину $O(\log n)$.*

Следствие 2.3.1. *Алгоритм построения овердрева с использованием быстрых суффиксных ссылок тратит $O(\log n)$ времени на один вызов add.*

Пример. Пусть надо реализовать вызов $add(b)$ к строке $S = cabaabaaba$. Максимальным суффикс-палиндромом S является $v = abaabaaba$. После неудачных сравнений с b символов, предшествующих v и $link[v] = abaaba$, выполняется переход к $quickLink[v] = a$. После успешного сравнения будет найден новый максимальный суффикс-палиндром bab. Суффикс-палиндрому aba строки S предшествует та же буква, что и $link[v]$, так что пропуск заведомо неудачного сравнения не влияет на результат работы алгоритма.

Замечание 2.3.1. *Более естественно выглядела бы ссылка $quickLink$ из v в максимальный суффикс v, символ перед которым отличен от символа перед v. Однако провести такую ссылку из v невозможно, т.к. v — вершина дерева и в ней нельзя сохранить символ, который идёт перед v, ведь v мог несколько раз входить в строку и иметь разный предшествующий символ. С другой стороны, мы знаем, какой символ стоит перед и внутри v, если $u = link[v]$. Соответственно, быструю ссылку можно определять для пары $(v, link[v])$, а эта пара однозначно задаётся через v. Поэтому $quickLink$ можно провести только из v.*

Поиск суффикс-палиндромов с помощью прямых ссылок

В предыдущем разделе мы использовали переход к суффиксу, которому предшествует другой символ. Теперь пойдём дальше в этой идее и укажем еще более быстрый алгоритм, который, правда, требует $O(\min \log |\Sigma|, \log \log n)$ памяти дополнительно при создании каждой вершины. Из каждой вершины сделаем $|\Sigma|$ ссылок, назовём их прямыми ссылками ($directLink[v][c]$). $directLink[v][c]$ — это максимальный из всех таких суффикс-палиндромов палиндрома v, которым внутри v предшествует символ c. Следующая лемма очевидна.

Лемма 2.3.2. *Массивы $directLink[v]$ и $directLink[link[v]]$ совпадают во всех позициях, кроме позиции c, соответствующей символу, предшествующему $link[v]$ в v.*

Предложение 2.3.2. *Алгоритм построения овердрева с использованием прямых ссылок тратит $O(\log |\Sigma|)$ времени на один вызов add и $O(\min \log |\Sigma|, \log \log n)$ памяти на добавление одной вершины к дереву.*

Доказательство. При создании вершины v мы сначала находим $link[v]$, после чего нам нужно заполнить массив $directLink[v]$. Для этого нам достаточно его скопировать из $directLink[link[v]]$, после чего изменить один символ.

В основной версии овердрева мы вычисляем, куда провести $link$, с помощью спуска по существующим суффиксным ссылкам. При имеющемся $directLink$ мы можем заменить спуск по суффиксным ссылкам на одно обращение к $directLink$.

Понятно, что в данном случае самый затратный по времени шаг — это копирование массива. Для ускорения этого шага, мы будем хранить массивы $directLink$ всех вершин овердрева, как персистентное сбалансированное дерево поиска [15] (см. описание в п. 1.2.6). Каждый массив $directLink[v]$ будет храниться как «обычное» сбалансированное дерево поиска, т.е., версия персистентного дерева. Ключами в дереве будут символы алфавита, а значениями — ссылки на вершины овердрева. От персистентного дерева нам потребуется всего две операции: копирование версии за $O(1)$ и вставка/изменение элемента за логарифмическое время и логарифмическую память от размера версии. Формируя массив $directLink[v]$ для новой вершины v овердрева, мы создаем новую версию, копируя ее из версии $link[v]$ и изменяем ссылку по символу, предшествующему $link[v]$ в v (либо создаем эту ссылку, если она была не определена). Поскольку версия содержит не более $|\Sigma|$ элементов, расходы времени и памяти укладываются в $O(\log |\Sigma|)$.

Поскольку в версии для v вершины определены только для символов, которые встречались слева от какого-нибудь суффикс-палиндрома v, из предложения 2.3.1 следует, что размер версии есть $O(\log n)$. А значит, на создание версии выделяется $O(\min\{\log\sigma, \log\log n\})$ памяти. □

В итоге, получили алгоритм, который по времени лучше предыдущего, а по памяти — хуже.

Возможные реализации и открытый вопрос оптимального решения

Для сравнения изложенных методов построения овердрева сведем их в таблицу:

Версия	Время на n запросов	Время на один запрос	Память на одну вершину
basic	$\Theta(n \log \lvert\Sigma\rvert)$	$O(n)$	$\Theta(1)$
quickLink	$\Theta(n \log \lvert\Sigma\rvert)$	$O(\log n)$	$\Theta(1)$
directLink	$\Theta(n \log \lvert\Sigma\rvert)$	$O(\log \lvert\Sigma\rvert)$	$O(\min \log \lvert\Sigma\rvert, \log\log n)$

Преимущество версии *basic* в том, что она наиболее просто реализуется и имеет минимальную константу по памяти. Версия *quickLink* лучше всего работает, когда размер алфавита и строки одного порядка. Версия *directLink* хорошо работает для маленьких алфавитов и длинных строк. Преимущество второй и третьей версий в том, что мы можем безболезненно отменить одно действие (оно не очень дорогое), а значит, можем реализовать структуру данных с откатами. А именно, функция *pop*() может быть реализована за константное время: функция *add* добавляет в стек номер вершины максимального суффикс-палиндрома и флаг, был ли этот палиндром новым, а *pop*() считывает эту информацию со стека и восстанавливает предшествующее состояние дерева.

Одна из двух оптимизированных версий позволяет на каждом шаге выделять $O(1)$ памяти, а другая — укладываться в $O(\log |\Sigma|)$ по времени. Отсюда возникает

Открытая проблема 2.3.1. *Существует ли онлайновый алгоритм построения овердрева, который на каждом шаге тратит $O(1)$ памяти и $O(\log |\Sigma|)$ времени?*

2.3.3 Подсчёт палиндромно-насыщенных строк

Согласно лемме 1.2.1, число различных подпалиндромов строки не превышает длины строки. Строки длины n, которые имеют в точности n различных подпалиндромов, называются *палиндромно-насыщенными*. Их изучению посвящен ряд работ, см., напр., [23].

В Онлайн-энциклопедии целочисленных последовательностей [46] приведена последовательность $A216264$, представляющая собой комбинаторную сложность языка бинарных

палиндромно-насыщенных строк, т.е. функцию, сопоставляющую числу L количество таких строк длины L. Дж. Шаллит вычислил указанную функцию для всех $L \leq 25$, последнее значение — 3 089 518. С помощью овердрева мы продвинулись в вычислениях гораздо дальше — до $L = 60$ (последнее значение — 2 132 734 033 216), причем вычисления заняли всего 10 часов работы пользовательского ноутбука. Новые полученные данные позволили понять, что данная последовательность возрастает не так быстро, как считалось ранее. В частности, полученные данные использовались в работе [26].

Аналогичные эксперименты были проведены для трёх- и четырёхбуквенного алфавита; там тоже удается дойти до значений того же порядка за сравнимое время. Результаты для двухбуквенного алфавита теперь приведены в упомянутой онлайн-энциклопедии ($oeis.org/A216264/b216264.txt$), результаты для больших алфавитов доступны по ссылкам `http://pastebin.com/u7qBqLpU`, `http://pastebin.com/gWsgcXjy`.

Теоретической основой для такого продвижения является приведенное ниже предложение.

Следующая лемма элементарно следует из леммы 1.2.1.

Лемма 2.3.3. *Любой префикс палиндромно-насыщенной строки является палиндромно-насыщенным.*

Предложение 2.3.3. *Пусть над k-буквенным алфавитом существует ANS палиндромно-насыщенных строк длины $\leq n$ для некоторых фиксированных k и n. Тогда бор, составленный из этих строк, можно обойти за время $O(ANS)$.*

Доказательство. Рассмотрим простейшую функцию, реализующую основанный на лемме 2.3.3 перебор всех бинарных палиндромно-насыщенных строк для всех длин, не превосходящих n, с одновременным подсчетом числа этих строк. Очевидно, эта функция масштабируется на произвольные алфавиты (с добавлением множителя $|\Sigma|$ во времени работы).

```
void calcRichString(S)
        ans[|S|]++;
        if (isRich(S + '0') && |S| < n)
                calcRichString(S + '0')
        if (isRich(S + '1') && |S| < n)
                calcRichString(S + '1')
```

Заметим, что внутри каждого конкретного вызова функции выполняется $O(1)$ действий помимо двух вызовов *isRich*. Кроме того, при каждом вызове *calcRichString* мы увеличиваем один из элементов массива *ans*, а значит, наше решение работает за $O(ANS \cdot T)$, где ANS — число различных бинарных палиндромно-насыщенных строк длины не больше n, а T — время работы одного вызова функции.

Если для проверки каждой строки на палиндромную насыщенность мы построим ее овердрево, то общее время работы будет $O(ANS \cdot n)$ (поскольку мощность алфавита — константа).

Ниже мы реализуем решение, которое работает за $O(ANS)$.

Для начала немного модифицируем предыдущий код.

```
void calcRichString(i)
        ans[i]++
        if (i < n)
                if (add('0') )
                        calcRichString(i + 1)
                pop()
                if (add('1') )
                        calcRichString(i + 1)
                pop()
```

Он отличается тем, что мы используем версию овердрева с откатами. В данном случае *add* приписывает к палиндромно-насыщенной строке новый символ и сообщает, появился ли новый палиндром (что по лемме 1.2.1 эквивалентно тому, что полученная строка также палиндромно-насыщенная). Если новая строка — палиндромно-насыщенная, то мы делаем рекурсивный вызов функции. Далее независимо от того, палиндромно-насыщена строка или нет, мы откатываем (pop) последнее действие, возвращая строку в исходное состояние. Таким образом, при вызове *calcRichString*(0) мы моделируем обход префиксного дерева всех палиндромно-насыщенных строк в глубину. Кроме того, мы теперь не передаём строку явно в функцию, чтобы сократить время на передачу параметров до $O(1)$.

Как мы знаем из п. 2.3.2, функция *pop*() реализуется за $O(1)$. Теперь разберёмся с *add*. Нам потребуется реализация *directLink*. В нашем случае алфавит бинарный, что даёт возможность просто копировать массив *directLink* из двух элементов за $O(1)$. Итого функции *add*, *pop* рабо-

тают за $O(1)$, давая общую асимптотику $O(ANS)$. □

Замечание 2.3.2. *Чтоб вычислить первые* 58 *членов последовательности* $A216264$ *за 10 часов на пользовательском ноутбуке, достаточно реализовать описанный выше алгоритм (пример реализации см. по ссылке* `http://pastebin.com/4YJxVzep`*).*

Чтобы дойти до $L = 60$*, мы использовали различные неасимптотические оптимизации, сокращающие константу времени выполнения. Они существенно затрудняют чтение исходного кода, поэтому мы не описываем их здесь. Сам код можно найти по ссылке* `http://pastebin.com/hGdKwZkr`

2.3.4 Поиск подпалиндромов на дереве

В предыдущем разделе мы построили овердрево с откатами. Логичным обобщением структур данных с откатами являются персистентные структуры. Это структуры, в которых есть возможность адресовать каждый очередной запрос не только к самой структуре, но и к предыдущим версиям этой структуры, при этом каждый изменяющий запрос возвращает новую (последнюю) версию структуры, и любая из версий сохраняется в дальнейшем.

Пусть есть дерево T (будем называть его *деревом версий*), вершины которого, кроме корня, помечены символами; дерево интерпретируется как множество версий некоторой строки (вершине v соответствует строка, читаемая на пути из корня до v). Задача состоит в сопоставлении каждой вершине версии овердрева для соответствующей строки. Более конкретно, есть функция $addVersion(v, c)$, которая подвешивает к вершине v новую вершину u, помеченную символом c, и при этом вычисляет версию овердрева для вершины u. Структуру данных, реализующую запросы $addVersion(v, c)$, назовем персистентным овердревом. Как показывает следующее предложение, эту сложную структуру данных можно реализовать весьма эффективно.

Предложение 2.3.4. *Существует реализация персистентного овердрева, которая выполняет каждый запрос* $addVersion(v, c)$ *за время* $O(\log |v|)$ *с использованием* $O(\log |v|)$ *памяти на одну вершину.*

Используем $directLink$-версию построения овердрева. Как и ранее, строится совместное овердрево для всех версий; в каждой вершине дерева версий хранятся ссылки на палиндромы соответствующей этой вершине строки. В совокупности, вершина v хранит следующую информацию: персистентное дерево поиска $searchTree[v]$, содержащее ссылки на все подпалиндромы v; ссыл-

ка $maxSufPal$ на максимальный суффикс-палиндром v; массив $pred[v]$, i-й элемент которого содержит ссылку на предка v в дереве версий, расстояние от которого до v равно $2^i (i \geq 0)$; символ $symb[v]$, который нужно добавить к родителю v, чтобы получить v. Все вышеперечисленное, кроме дерева поиска, занимает $O(\log |v|)$ памяти. Персистентное дерево поиска (см. раздел 1.2.6) требует также $O(\log |v|)$ времени и столько же дополнительной памяти на создание его копии и вставку одного элемента.

Отождествим вершину в дереве версий (или версию) со строкой, которой она соответствует. Будем через v обозначать и вершину, и строку. Покажем, как реализовать $addVersion(v, c)$ за время $O(\log |v|)$. Вначале заметим, что для любого i символ $v[i]$ можно найти за время $O(\log |v|)$ следующим образом. Этот символ хранится в предке вершины v, который находится от нее на расстоянии $h = |v| - i$. Значит нужно по ссылкам на предков подняться на h. Разложив число h на сумму степеней двойки, мы за не более $\log d$ переходов по ссылкам на предков можем добраться до необходимой вершины, а в ней взять соответствующий символ из $symb$.

Обозначим через V текущее количество версий. Для создания новой версии u нам будет достаточно увеличить V на единицу и заполнить все необходимые поля. Первое, что нам нужно сделать, —вычислить массив $pred$ для данной вершины. Это делается за время $O(\log |v|)$, поскольку $pred[u][0] = v$ и $pred[u][i] = pred[pred[u][i-1]][i-1]$ для $i > 0$.

Для вычисления $maxSufPal[u]$ выполним вызов функции $add(c)$ (версия $directLink$) для текущей строки, равной v. Палиндром $maxSufPal[u]$ в овердреве получаем по ребру либо из палиндрома $maxSufPal[v]$ (если этому суффиксу в строке v предшествует символ c), либо из палиндрома $directLink[maxSufPal[v]][c]$ (в противном случае). Таким образом, для вычисления $maxSufPal[u]$ нужно один раз обратиться к символу строки v, а именно $v[|v| - maxSufPal[v]]$. Если палиндром $maxSufPal[u]$ окажется новым, для него надо создать вершину овердрева; при этом в массиве $directlink[maxSufPal[u]]$ вычисления требует ровно один элемент (остальные копируются), что требует одного обращения к строке v.

```
array getpred(v, par)
        ans[0] = par
        for (i = 1; ans[i] > 0; i++)
                ans[i + 1] = pred[ ans[i] ][i]
        return ans
int addVersion(v, c)
```

```
V++ // число вершин, начальное значение - ноль
u = V
symb[u] = c
pred[u] = getpred(u, v)
if (c == v[len[v] - len[maxSufPal[v]]])
        x = maxSufPal[v]
else
        x = directLink[maxSufPal[v]][c]
if(to[x][c] == null)
        to[x][c] = createNode(x, c)
maxSufPal[u] = to[x][c]
searchTree[u] = insert(searchTree[v], maxSufPal[u])
return u
```

Функция $createNode(x, c)$ создаёт новую вершину и возвращает её, предварительно проведя в неё ребро из x, помеченное символом c, также она вычисляет в ней суффиксную ссылку.

Таким образом, вычисление $maxSufPal[u]$ требует $O(\log|\Sigma|)$ времени (время работы $add(c)$ за вычетом обращений к строке) плюс два обращения к строке v (каждое за $O(\log|v|)$, как показано выше). Кроме того, надо скопировать персистентное дерево поиска $searchTree$ из v в u (со вставкой нового элемента или без нее), что также требует $O(\log|v|)$ времени. Таким образом, получаем общую оценку $O(\log|v|)$. □

Замечание 2.3.3. *В данном алгоритме для каждой вершины v вычисляется множество всех её различных подпалиндромов ($searchTree[v]$). При этом в самом алгоритме $searchTree$ не используется. Эти данные нужны для ответов на запросы, которые могут быть заданы к структуре. Например, запрос может иметь вид «посчитать число различных подпалиндромов в строке версии v».*

2.4 Задачи о разбиении на подпалиндромы

В [22] было доказано, что разбиваемость строки на 1, 2, 3 или 4 палиндрома может быть проверена за линейное время и была поставлена задача проверки существования разбиения строки

на k палиндромов для произвольного заданного значения параметра k. В [34] приведено решение этой задачи за время $O(kn \log n)$, которое затем при помощи метода битового сжатия [1], называемого в литературе методом четырёх русских, доведено до асимптотики $O(kn)$. В данном разделе мы приведем простой и практичный алгоритм, работающий за время $O(n(k + \log n))$, а затем сфокусируемся на построении решения, не зависящего от k.

2.4.1 Простое решение за $O(kn + n \log n)$

В данном разделе мы приведём без доказательства идею алгоритма разбиения строки на k палиндромов за время $O(nk + n \log n)$. Эта идея, с некоторыми изменениями, лежит в основе приводимого ниже алгоритма разбиения за время $O(n \log n)$, и все используемые комбинаторные свойства будут строго доказаны в п. 2.4.3. Приводимый ниже алгоритм лег в основу работы [34]; однако поскольку использование битового сжатия для него не позволяет получить оценку сложности $O(nk)$, в самой работе [34] приведена его более сложная модификация, предложенная Д. Косолобовым.

Ключевым для нашего алгоритма (как и для его модификации, а также алгоритма из [17]) является понятие *серии палиндромов*. Серия палиндрома v состоит из v и, возможно, некоторых суффикс-палиндромов v. Пусть $v_1, \ldots, v_k$ — все суффикс-палиндромы v в порядке убывания длины. Тогда $p_1 = |v| - |v_1|$ — минимальный период v, $p_2 = |v_1| - |v_2|$ — минимальный период v_1, и т.д. Тогда $p_1 \geq p_2 \geq \cdots \geq p_k$, и каждая серия состоит из всех палиндромов с одним и тем же минимальным периодом. Общее число серий, на которые разбиваются суффикс-палиндромы v, есть $O(\log |v|)$. Таким образом, любая строка имеет логарифмическое число серий суффикс-палиндромов.

Каждую серию суффикс-палиндромов строки будем задавать парой (L, p), где $L = |v|$ для самого длинного («ведущего») палиндрома v этой серии, а p — минимальный период палиндромов серии. В каждый момент времени будем явно хранить массив серий (т.е. пар чисел), он имеет логарифмическую длину. На каждом шаге после дописывания символа будем перебирать все серии и обновлять информацию о них.

Обновление списка серий занимает время, линейное от длины списка. Ввиду периодичности, перед всеми палиндромами серии, кроме, быть может, ведущего, в строке стоит один и тот же символ. Это значит, что при дописывании символа к строке либо все эти палиндромы «выживут». т.е. расширятся до суффикс-палиндромов, либо все «отомрут». Это позволяет

перестроить каждую серию за константное время. После этого к концу списка серий добавляется серия для суффикс-палиндрома длины 2 (если он есть) и суффикс-палиндрома длины 1. Наконец, поскольку период серии увеличивается в случае, когда от нее остался только один палиндром (а на предыдущем шаге было больше), за линейный пробег по всем сериям проверяется, сливаются ли некоторые соседние серии в одну. Итого, поддержание актуального списка серий суффикс-палиндромов на каждом шаге требует суммарно времени $O(n \log n)$.

Будем решать задачу разбиения на k палиндромов методом динамического программирования. Будем хранить булеву матрицу res размера n на k. Элемент $res[j][i]$ будет равен $true$ в случае, если строку $S[1..i]$ можно разбить на j палиндромов.

Чтобы найти значение $res[j][i]$, нужно взять булево ИЛИ всех значений $res[j-1][t]$ по всем позициям t, предшествующим суффикс-палиндромам $S[1..i]$. Это можно сделать за время, линейное от числа серий: для одноэлементных серий требуется одна операция ИЛИ, а для многоэлементной серии (L, p) при обработке $(i-p)$-го символа строки существовала серия $(L-p, p)$ и для нее вычислялось ИЛИ по всем требуемым в данный момент позициям, кроме самой правой. Запоминая соответствующие результаты вычисления ИЛИ, мы и для многоэлементных серий обойдемся ровно одним ИЛИ.

Если для каждого префикса строки вначале обновлять список серий, а затем обновлять ИЛИ для всего столбца размера k, то побитовые ИЛИ для столбцов из k битов можно проводить группами по $\log n$ битов. Для каждой серии можно сделать это за время $O(k/\log n)$, что в сумме дает $O(k)$ по всем сериям. Таким образом, получим время $O(nk)$ на обновление таблицы. Вместе с приведенной выше оценкой на время обновления серий, получаем требуемый результат.

2.4.2 Разбиение на k палиндромов и поиск палиндромной длины

Потребность в решении задачи о разбиении на k палиндромов за время, не зависящее от k, обусловлена тем, что палиндромная длина слова, т.е. минимальное k, для которого существует разбиение на k палиндромов, обычно очень велика. Так, в [41] доказано, что для случайной строки длины n палиндромная длина есть $\Omega(n)$. Понятно, что в таком случае решение за $O(kn)$ — слишком медленное.

Чтобы разбить строку на k палиндромов за время $O(n \log n)$ независимо от k, мы вначале установим связь между существованием такого разбиения и палиндромной длиной слова.

Лемма 2.4.1. *Если дано разбиение строки длины n на k палиндромов, то за время $O(n)$ можно*

получить разбиение строки на $k + 2t$ палиндромов для любого натурального t такого, что $k + 2t \leq n$.

Доказательство. Считаем, что в разбиении есть хотя бы два палиндрома не единичной длины, т.к. другой случай тривиален.

Возьмём самый большой палиндром в разложении. Если его длина больше двух, то мы отрежем с его концов по одному символу и получим три палиндрома вместо одного. Т.е. увеличим число палиндромов в разбиении на два. Если же все палиндромы не длинней двух, то возьмём два палиндрома длины два и разрежем, получив на два больше палиндрома в разложении. □

Таким образом, задача разбиения на k палиндромов сводится за время $O(n)$ к решению двух похожих задач: разбить строку на минимальное четное число палиндромов и на минимальное нечетное число палиндромов. Ниже мы покажем, как решить эти задачи при помощи овердрева. Для этого мы сначала приведём решение задачи о поиске палиндромной длины с помощью овердрева.

2.4.3 Решение задачи о палиндромной длине

В данном разделе мы приведём онлайновый алгоритм, решающий задачу разбиения строки на минимальное число палиндромов за время $O(n \log n)$.

Заметим, что существует другое решение этой задачи, работающее также за $O(n \log n)$, оно приведено в [17]. Наше решение асимптотически такое же по времени, но имеет существенно меньшую константу (см. Замечание 2.4.2). На наш взгляд, наше решение также проще с точки зрения реализации и уменьшает количество потенциально возможных ошибок.

Предложение 2.4.1. *По строке длины n за время $O(n \log n)$ с помощью овердрева можно найти онлайн минимальное количество палиндромов, на которое эту строку можно разбить.*

Доказательство. Для решения задачи вычислим массив ans, где $ans[i]$ — палиндромная длина строки $S[1..i]$. Заметим, что $(k+1)$-разбиение строки $S[1..i]$ получается добавлением палиндрома $S[j..i]$ к k-разбиению строки $S[1..j-1]$. Тогда для вычисления ответа $ans[i]$, нам достаточно перебрать все суффикс-палиндромы, взять минимум из чисел, находящихся в ans на местах, непосредственно предшествующих началам этих суффикс-палиндромов, и добавить к нему единицу. Таким образом верна формула

$$ans[i] = 1 + \min_{S[j+1..i] \in Pal} ans[j] \tag{2.1}$$

Для данной задачи нам понадобится в каждой вершине овердрева хранить две дополнительные величины: разность ($diff$) и серийную ссылку ($seriesLink$). Здесь $diff[v] = len[v] - len[link[v]]$, т.е. разница длин данного палиндрома и его максимального суффикс-палиндрома, а $seriesLink[v]$ — вершина в суффиксном пути v, имеющая максимальную длину среди тех, разность которых отлична от $diff[v]$. Таким образом, $seriesLink[v]$ — это разновидность быстрой суффиксной ссылки, пропускающая все вершины с разностью $diff[v]$.

В дальнейшем *серией* палиндрома v будем называть последовательность вершин в суффиксном пути v от v (включительно) до $seriesLink[v]$ (исключительно). Обе эти величины очевидно могут быть вычислены при создании вершины с использованием $O(1)$ времени и $O(1)$ дополнительной памяти. Следующий код демонстрирует это.

```
if (diff[v] == diff[link[v]])
        seriesLink[v] = seriesLink[link[v]]
else
        seriesLink[v] = link[v]
```

Замечание 2.4.1. *Можно заметить, что seriesLink очень похожа на quickLink. Для целей данного раздела нам не подойдёт quickLink, но несложно понять, что вместо quickLink мы всегда можем использовать seriesLink без ухудшения асимптотики, т.к. спуск по seriesLink также имеет логарифмическую длину (см. лемму 2.4.5). Однако, множество вершин в пути по quickLink является подмножеством вершин в пути по seriesLink.*

Рассмотрим наивную реализацию, вычисляющую за линейное время значение $ans[n]$.

```
ans = ∞
for (v = maxSufPal; len[v] > 0; v = link[v])
        ans[n] = min(ans[n], ans[n - len[v]] + 1)
```

Теперь используем серийные ссылки и реализуем алгоритм за то же время.

```
ans = ∞
for (v = maxSufPal; len[v] > 0; v = seriesLink[v])
        ans[n] = min(ans[n], getMin(v))
int getMin(u)
```

```
res = ∞
for (v = u; v ≠ seriesLink[u]; v = link[v])
        res = min(res, ans[n - len[v]] + 1)
return res
```

Функция $getMin$ в худшем случае работает за линейное время. Ниже мы ускорим его до константного времени. Заметим, что сравнением $diff[u]$ и $diff[link[u]]$ можно проверить, состоит ли серия палиндрома u из одного элемента, а для таких серий $res = ans[n - len[u]] + 1$ можно вычислить за $O(1)$. Поэтому в дальнейшем мы работаем только с сериями из более, чем одного элемента. Нам потребуется четыре вспомогательные леммы.

Лемма 2.4.2. *Пусть для данной строки $S[1..n]$ палиндром v длины $L \geq n/2$ является одновременно префиксом и суффиксом. Тогда S — палиндром.*

Доказательство. Пусть $i \leq n/2$. Тогда $S[i] = v[i] = v[L-i+1] = S[n-i+1]$, т.е. S – палиндром по определению. □

Лемма 2.4.3. *Пусть для данной строки $S[1..n]$ суффикс-палиндром v — максимальный в некоторой серии, т.е. следующий по возрастанию длины палиндром имеет другой $diff$ и $diff[v] = diff[link[v]]$, т.е. в серии более одного элемента. Тогда $link[v]$ входит в строку v ровно два раза: как префикс и как суффикс.*

Доказательство. Положим $i = n - len[v] + 1$, а $u = link[v]$. Тогда мы знаем, что $v = S[i..n], u = S[i+diff[v]..n] = S[i..n-diff[v]]$. Серия v имеет более одного палиндрома, значит $diff[v]$ меньше половины длины v, а, следовательно, два указанных вхождения u перекрываются. Далее будем доказывать от противного. Предположим, существует k, $i < k < i + diff[v]$ такое, что $S[k..k + len[u] - 1] = u$. Тогда по лемме 2.4.2 строка $S[k..n]$ является палиндромом, что противоречит тому, что u — максимальный суффикс-палиндром v. □

Лемма 2.4.4. *Пусть для данной строки $S[1..n]$ суффикс-палиндром v — максимальный в некоторой серии, $u = link[v]$, $diff[v] = diff[u]$, т.е. серия состоит не из одного палиндрома. Тогда в строке $S[1..n - diff[v]]$ суффикс-палиндром $link[v]$ является максимальным в серии.*

Доказательство. Если u — не максимальный в серии, то строка $S[1..n-diff[v]]$ имеет суффикс-палиндром $z = S[j..n-diff[v]]$ такой, что $link[z] = u$ и $diff[z] = diff[u] = diff[v]$. Поскольку

u — суффикс и префикс z и $|z| = |v| \leq 2|u|$, очевидно, $z = v$. Тогда $w = S[j..n]$ является палиндромом по лемме 2.4.2(требуемым в лемме префиксом и суффиксом является v). Предположим, что w имеет суффикс-палиндром v', который длинней, чем v. То, что u является суффикс-палиндромом v' влечёт, что u — префикс-палиндром v' и данное вхождение u заканчивается между позициями $n - diff[v]$ и n, что противоречит лемме 2.4.3. Значит, такого v' не существует, т.е. $v = link[w]$. Но $diff[w] = |w| - |v| = |w| - |z| = diff[v]$, что невозможно, т.к. v максимальный в серии. Полученное противоречие доказывает, что палиндром u — максимальный в серии в строке $S[1..n - diff[v]]$. □

Лемма 2.4.5. *Длина пути по серийным ссылкам имеет порядок* $O(\log n)$.

Доказательство. Вначале заметим, что наименьший период палиндрома v равен $diff[v]$. Действительно, пусть $u = link[v]$. Тогда u — суффикс и префикс v, а значит, величина $|v| - |u| = diff[v]$ — период v по определению. Обратно, если p — период v, то префикс v длины $|v| - p$ равен суффиксу той же длины, а значит, является палиндромом. Поскольку u — максимальный суффикс-палиндром v, имеем $p \geq diff[v]$.

Значит, мы можем записать $v = (xy)^k x$, где $k \in \mathbb{N}$, y — непустая строка (а строка x может быть пустой) и $|xy| = diff[v]$. Пусть v — максимальный в серии и $seriesLink[v] = w$. Тогда $w = (xy)^l x$, где $0 \leq l < k$. Если $l = 0$, то $|w| < |v|/2$. Пусть $l > 0$. Заметим, что w имеет не только период $diff[v]$, но и строго меньший период $diff[w]$ (это минимальный период w, и он не совпадает с $diff[v]$ по определению $seriesLink$). Одно из основных свойств периодов (см. напр. [18]) гласит, что все периоды слова, не превосходящие половину его длины, кратны минимальному периоду. Таким образом, при $l \geq 2$ $diff[v]$ кратно $diff[w]$, т.е. можно записать $xy = z^m$, где $m = diff[v]/diff[w]$. Но тогда $|z| = diff[w]$ является периодом всего слова v, противоречие. Следовательно, $l = 1$, и из условия $k > l$ следует, что $|w| < \frac{2}{3}|v|$. Итак, мы доказали, что длина палиндрома $seriesLink[v]$ меньше длины v по крайней мере в полтора раза; значит, длины максимальных элементов в сериях суффикс-палиндромов убывают по экспоненте, что и требовалось доказать. □

По лемме 2.4.5, алгоритм вызывает $getMin$ $O(\log n)$ раз, а значит, реализовав $getMin$ за $O(1)$, мы решим задачу за $O(\log n)$ на шаг.

На Рис. 2 сверху изображена серия палиндрома v, а снизу — серия предыдущего вхождения $link[v]$, в соответствии с леммами 2.4.3 и 2.4.4. Напомним, что мы работаем со случаем $diff[v] = diff[link[v]]$.

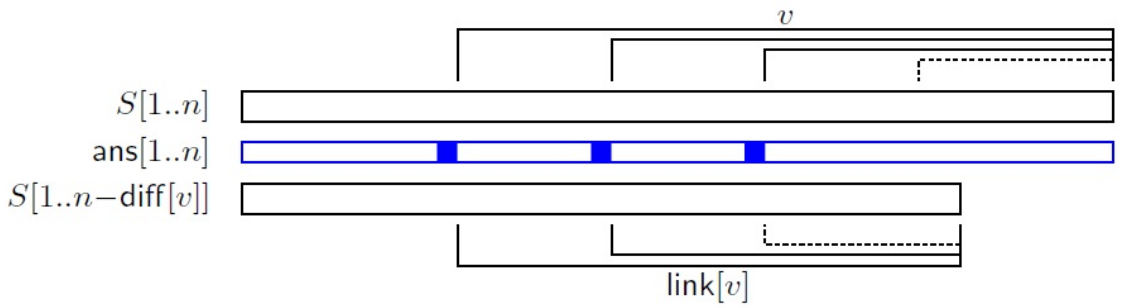

Рис. 2.3: Серии палиндрома v в $S[1..n]$ и палиндрома $link[v]$ в $S[1..n - diff[v]]$. Максимальные палиндромы в следующих сериях показаны пунктирными линиями. Функция $getMin(v)$ возвращает минимум значений, помеченных в массиве ans, увеличенный на единицу.

На рисунке видно, что левые границы палиндромов в сериях совпадают везде, кроме одной позиции. Поскольку ссылка из v ведёт в $link[v]$, количество элементов в серии v ровно на единицу больше. Левые границы первых палиндромов в сериях совпадают, т.к. $link[v]$ является префиксом v, а последующих — ввиду равенства $diff$ в этих сериях.

Отсюда видно, что $diff[v]$ шагов назад мы уже считали минимум из всех нужных чисел, кроме одного. Значит этот минимум нужно было тогда запомнить, чтоб сейчас использовать. Мы можем это сделать прямо внутри функции $getMin$, т.к. $link[v]$ встретился в прошлый раз $diff[v]$ шагов назад по лемме 2.4.3, причём являлся на том шаге максимальным в своей серии (по лемме 2.4.4). Наш код сохранит $diff[v]$ шагов назад старое значение в вершине овердрева в переменной $dp[link[v]]$. Приведённый ниже код запоминает значение:

```
int getMin(v)
        dp[v] = ans[n - (len[seriesLink[v]] + diff[v])] //поместим в dp значение,
           находящееся слева от минимального суффикс-палиндрома в серии
        if (diff[v] == diff[link[v]])// обрабатываемая серия нетривиальна
              dp[v] = min(dp[v], dp[link[v]])
        return dp[v] + 1
```

В первой строке функции мы инициализируем значение dp при помощи последнего палиндрома в обрабатываемой серии. Длина первого палиндрома в следующей серии равна $len[seriesLink[v]]$, а чтоб взять предыдущий, достаточно прибавить $diff[v]$. А значит нас интересует позиция $n - (len[seriesLink[v]] + diff[v])$.

Для случая, когда в серии всего один палиндром, ответ уже посчитан. Если в серии не один палиндром, то мы используем факт, показанный выше: все кроме одного (самого правого) значения посчитаны в $dp[link[v]]$. Значит с помощью $dp[link[v]]$ мы можем обновить результат.

Предложение доказано. □

Замечание 2.4.2. *Пусть T_i — количество серий у строки $S[1..i]$.*

В нашем алгоритме на каждом шаге мы тратим $O(\log|\Sigma|)$ на обновление овердрева. Затем T_i раз вызываем функцию $getMin$, которая заполняет одну новую ячейку в массиве dp и одну новую ячейку в массиве ans. Т.е. всего $2 \cdot T_i$ записей в массивы.

В алгоритме из [17, Figure 8] поочерёдно строятся массивы троек чисел G, G', G'', каждый из которых размера T_i. Т.е. на каждом шаге заполняется $9 \cdot T_i$ ячеек. Таким образом, константа времени работы нашего алгоритма намного меньше, чем у алгоритма [17, Figure 8].

Наша реализация приведена по ссылке `http://ideone.com/xE2k6Y`.

Для задачи разбиения на k палиндромов нам требовался алгоритм, который разбивает строку на минимальное чётное/нечётное число палиндромов. Чтоб получить такой алгоритм, достаточно вместо массивов ans и dp в вышеприведенном алгоритме создать массивы $ansOdd$, $ansEven$, $dpOdd$, $dpEven$, после чего при вычислении значений в массивах с суффиксом Odd в названии использовать массивы с суффиксом $Even$ и наоборот.

Тем самым, получим алгоритм с той же оценкой сложности, который вычисляет минимальное четное и минимальное нечетное число палиндромов в разбиении строки. Следовательно, справедлива

Теорема 2.4.1. *Задача разбиваемости строки на ровно k палиндромов разрешима за время $O(n \log n)$.*

2.4.4 Гипотеза о линейном решении

Важный вопрос: можно ли вычислить палиндромную длину быстрее, чем за время $O(n \log n)$? Во-первых, можно заметить, что асимптотика $O(n \log n)$ выглядит избыточной. В самом деле, для построения овердрева мы обрабатываем только $O(n)$ суффикс-палиндромов даже когда используем только обычные суффиксные ссылки. Для палиндромной длины, на каждом шаге мы пробегаем по всем суффикс-палиндромам, но возможно пропуская многие из них с помощью

серийных ссылок. Сокращается ли при этом число обрабатываемых палиндромов до $O(n)$? Как было отмечено в [17], ответ: «да» в среднем, но «нет» в худшем случае: анализ всех префиксов известных *слов Зимина* требует обработки $\Theta(n \log n)$ серий палиндромов (каждая из них одноэлементная, но это не помогает сократить время).

Итак, мы имеем алгоритм, который разбивает строку на минимальное число палиндромов за $O(n \log n)$, из них затрачивает на построение овердрева $O(n \log |\Sigma|)$. В этом разделе мы обсудим идеи, которые могут помочь добиться линейного поиска палиндромной длины. Для начала укажем алгоритм, строящий овердрево оффлайн за линейное время для строки длины n над алфавитом $\{1, \ldots, n\}$.

Для такого алфавита мы можем за линейное время построить для строки суффиксный массив [40] и массив радиусов палиндромов (с помощью алгоритма Манакера [35]).

1. Применив алгоритм Манакера, получим массивы радиусов $oddR$ и $evenR$, где $oddR[i]$ содержит радиус максимального палиндрома с центром в i, $evenR[i]$ — радиус максимального палиндрома с центром в точке $i + 1/2$.

2. Посчитаем максимальный и второй по величине суффикс-палиндромы.
Заведём для этого три переменные: r, ℓ, ℓ'. Между итерациями алгоритма поддерживается следующий инвариант: $S[\ell..r]$ — максимальный суффикс-палиндром строки $S[1..r]$, а $S[\ell'..r]$ — второй по величине среди суффикс-палиндромов $S[1..r]$.

```
ℓ = 2
for (r = 1; r ≤ n; r++)
        ℓ--
        while ( !isPal(S[ℓ..r] )
                ℓ++
        ℓ' = max(ℓ' - 1 , ℓ + 1)
        while ( !isPal(S[ℓ'..r] ) && (ℓ' ≤ r) )
                ℓ'++
        C[(ℓ + r) / 2].push(1, r)
        C[(ℓ' + r) / 2].push(2, r)
```

Функция $isPal$ проверяет верно ли, что строка — палиндром (за константное время по мас-

сиву радиусов).

Видно, что в приведённом коде внутренние циклы while делают не более $2n$ увеличений каждой из переменных ℓ и ℓ'. Таким образом, алгоритм линеен.

Каждый элемент массива C содержит связный список.

Заметим, что $\ell + r$ может оказаться нечётным числом (для случая чётного палиндрома), в данном случае мы можем считать, что индексы в массиве C могут быть как целыми, так и полуцелыми числами.

3. Построим суффиксный массив SA и массив LCP. Это можно сделать за линейное время [40]. Напомним, что ячейка i массива LCP равна наибольшему общему префиксу суффиксов $S[SA[i]..n]$ и $S[SA[i-1]..n]$.

4. Теперь, основываясь на данных первых трёх пунктов, мы можем построить овердрево за линейное время.

Из раздела 2.2.5 известно, что овердрево состоит из бора, хранящего правые половины чётных подпалиндромов, и бора, хранящего правые половины нечётных подпалиндромов.

Известно, что за линейное время можно построить суффиксное дерево по суффиксному массиву [30]. Данный алгоритм строит сжатый бор всех суффиксов, используя суффиксный массив и массив LCP. В данном случае нам нужно построить бор из правых половин всех подпалиндромов (один бор для чётных и один для нечётных). Для этого из каждого суффикса нужно взять только некоторый его префикс, по длине равный радиусу в начальной позиции. Из п. 2.2.4 знаем, что итоговый бор имеет размер $O(n)$, а значит нам не нужно производить сжатие, в отличие от суффиксного дерева.

Итак, рассмотрим модификацию алгоритма [30], который решает задачу для овердрева. Приведём алгоритм для нечётных палиндромов.

```
path = (-1) // стек текущей ветви бора
for (i = 1; i ≤ n; i++)
        k = SA[i] // начала обработки палиндромов с центром в k
        while (path.size() > LCP[i] + 1)
                path.pop()
        for (j = path.size(); j ≤ oddR[k]; j++) //цикл может оказаться пустым
                path.push( newNode(path.top(), S[k + j - 1]) )
        for (j = 1; j ≤ C[k].size(); j++)
```

```
(rank, right) = C[k][j]
node[rank][right] = path[right - k + 1] // сохраняем ссылку на
    длиннейший, либо второй по длине суффикс-палиндром строки
    S[1..right]
```

Функция $newNode(v, a)$ в данном коде возвращает новую вершину, подвешенную к вершине v ребром с меткой a.

Оценим время работы. Внешний цикл, очевидно, работает за линейное время. Нужно оценить внутренние циклы. Функция $path.pop()$ вызовется не больше раз, чем $push$, значит число её вызовов можно оценить через $push$. Эта функция вызовется ровно столько же раз, сколько и $newNode$, но мы знаем, что эта функция добавляет вершину в овердрево, которых не больше n. Таким образом первые два цикла суммарно работают не более $O(n)$ времени. Третий работает за суммарную длину массивов $C[k]$ по всем k, а эти массивы строились за линейное время, т.е. занимают линейную память. Таким образом алгоритм работает за линейное время.

Приведённый выше код нужно запустить дважды: в явном виде (для нечётных палиндромов) и с небольшими модификациями (для чётных палиндромов). После его работы у нас создано овердрево и для каждого префикса $S[1..i]$ хранятся $node[1][i]$ и $node[2][i]$ — вершины в дереве, которые являются его максимальным и вторым по величине суффикс-палиндромами.

Очевидно, что после этого суффиксные ссылки мы легко можем посчитать:

```
for (i = 1; i ≤ n; i++)
      link[ node[1][i] ] = node[2][i];
```

Таким образом мы доказали

Предложение 2.4.2. *Овердрево оффлайн можно построить за время $O(n)$ для случая, когда алфавит — это множество целых чисел от 1 до n.*

Итак, для оффлайновой версии задачи о палиндромной длине мы можем построить овердрево заранее за $O(n)$, после чего всё равно за $O(n \log n)$ найти разбиение.

В [34] алгоритм, работающий за $O(kn \log n)$, был оптимизирован до $O(kn)$ с помощью метода четырёх русских.

В том алгоритме вычислялась битовая матрица $k \times n$ из флагов, указывающих на разбива-

емость префиксов строки на заданное число палиндромов, поэтому такой метод ускорения был естественным. В нашей задаче мы работаем с массивом целых чисел, поэтому здесь непосредственное применение метода четырёх русских невозможно.

Лемма 2.4.6. *Пусть S — строка, c — символ. Если минимальное число палиндромов, на которое можно разбить S, равно k, то минимальное число палиндромов, на которое можно разбить Sc, равно $k-1, k$ или $k+1$.*

Доказательство. 1. Очевидно, что мы всегда можем получить $k+1$, для этого достаточно взять разбиение предыдущей строки и добавить в разбиение один новый символ. Значит ответ не больше $k+1$.

2. От противного. Допустим минимальный ответ для новой строки равен t и $t < k-1$. Тогда возьмём последний палиндром в разбиении. Если его длина больше двух, то мы можем разделить его на три палиндрома (первый символ, последний символ, центральная часть). Тогда видно, что для предыдущей строки существует разбиение из $t+1$, $t+1 < k$, получили противоречие. Если его длина равна двум или единице, то для меньшей строки существует разбиение t и $t-1$ соответственно, что аналогично приводит к противоречию. □

Теперь рассмотрим битовую матрицу n на n. В ячейке (i, j) располагается единица в случае, если префикс длины i можно разбить на j палиндромов, ноль иначе. Теперь вспомним, что проверять в каждой строке нам нужно только три числа ($k-1, k, k+1$, где k — минимальное разбиение с предыдущего шага). Для каждого из них нам нужно взять побитовый ИЛИ от $\log n$ битовых значений. Т.е. всего $3n \log n$ битовых операций. Если бы такие операции у нас были бы сгруппированы одним большим блоком, то мы методом четырёх русских выполнили бы их за $3n$ действий. Возможно, существует метод, который позволяет с группировать эти битовые операции. Отсюда

Гипотеза 2.4.1. *Используя лемму 2.4.6, овердрево и метод четырёх русских, можно решить задачу о разбиении на минимальное число палиндромов и о разбиении на k палиндромов за $O(n \log |\Sigma|)$ в онлайне и $O(n)$ в оффлайне.*

Глава 3

Повреждённые строки

3.1 Введение

В данной главе речь пойдёт о двух задачах о восстановлении повреждённых текстов, основанном на дополнительной информации. Напомним постановки задач, а также кратко опишем полученные результаты.

Задача 1. *Заданы повреждённый текст и повреждённый шаблон, необходимо среди всех возможных вариантов восстановления исходных неповреждённых текста и шаблона выбрать пару, для которой количество вхождений шаблона в текст максимально.*

Для частных случаев задачи 1 (когда повреждён только текст или только шаблон) мы приведем эффективные (полиномиальные по времени) алгоритмы, а в общем случае докажем, что данная задача NP-трудна даже если алфавит текста и шаблона — бинарный.

Задача 2. *Заданы повреждённый текст и повреждённый шаблон, необходимо среди всех возможных вариантов восстановления исходных неповреждённых текста и шаблона выбрать пару, для которой минимальна «непохожесть», вычисляемая как сумма расстояний Хэмминга для всех пар (шаблон, подстрока той же длины в тексте).*

Для частных случаев задачи 2 (для бинарного алфавита; для случая, когда повреждён только текст или только шаблон; для частичных циклических строк) мы приводим полиномиальные по времени алгоритмы, в общем случае докажем, что задача NP-трудна и даже APX-трудна. Для случая частичных строк вопрос о сложности остался открытым; мы предполагаем, что задача NP-трудна и приводим аргументы в пользу этой гипотезы.

Отметим, что в варианте распознавания (существует ли вариант восстановления с заданным

значением критерия оптимальности) и задача 1, и задача 2 принадлежат классу NP: полиномиальным сертификатом является пара (восстановленный текст, восстановленный шаблон), и его корректность очевидно проверяется за полиномиальное время.

3.2 Предварительные сведения

3.2.1 Основные определения

Пусть заданы конечные множества Σ, Γ ($\Sigma \cap \Gamma = \varnothing$) и бинарное отношение ρ на $\Sigma \cup \Gamma$, состоящее из всех пар (a, a), где $a \in \Sigma$, и некоторых пар (a, b), где $a \in \Sigma$, $b \in \Gamma$, с условием, что каждый элемент из Γ принадлежит хотя бы одной паре из отношения. Будем называть Σ *основным алфавитом* (или просто алфавитом), Γ — *множеством повреждённых символов*, $\Sigma \cup \Gamma$ — *повреждённым алфавитом*, а ρ — *отношением совместимости* на нем.

Строку над повреждённым алфавитом будем называть *повреждённой*. Строку над исходным алфавитом будем называть неповреждённой или полной. Неповреждённую строку S и повреждённую строку w будем называть *совместимыми*, если их длины равны и $(S[i], w[i]) \in \rho$ для любого i от 1 до $|S|$. Две повреждённых строки S и P будем называть *совместимыми*, если существует неповреждённая строка, совместимая с каждой из них.

Рассмотрим пример. Пусть множество повреждённых символов состоит из одного символа и этот символ совместим со всеми символами исходного алфавита. Строки с таким отношением совместимости представляют собой так называемые *частичные строки*. повреждённый символ в этом случае будем называть *джокером* и обозначать $\diamond$. Таким образом, строки с отношением совместимости являются обобщением частичных строк, которые, в свою очередь, являются обобщением обычных строк.

В дальнейшем текст S и шаблон P считаются повреждёнными строками. Кроме того, положим $n = |S|$, $m = |P|$.

3.2.2 Исторический обзор

Для задачи поиска подстроки в строке существуют различные алгоритмы, решающие ее за линейное время. Например, алгоритм Бойера-Мура, алгоритм Кнута-Морриса-Пратта, и множество других (см., например, [2]).

Несмотря на обилие алгоритмов решения задачи, ни один из них не обобщается для решения данной задачи для строк с произвольным отношением совместимости. Более того, не известен линейный алгоритм решения задачи поиска даже для частичных строк, а в естественном предположении из теории сложности такого алгоритма не существует, см. [38].

В статье [19] был приведен алгоритм, решающий задачу поиска подстроки для частичных строк за время $O(n \log n \log |\Sigma|)$. Позднее в статье [38] эта идея была использована для построения алгоритма, решающего задачу для произвольного отношения совместимости за время $O(|\Sigma| n \log n)$. Позже в [11] был приведён алгоритм, решающий задачу для частичных строк над целочисленным алфавитом за время $O(n \log n)$. Опишем кратко решение для повреждённых строк.

За $O(|\Sigma \cup \Gamma|)$ действий можем перебрать все символы алфавита. При фиксированном символе a алфавита выполним следующие действия. Мы хотим найти для каждой подстроки из S длины m количество совпадений символа a в этой подстроке с совместимыми с ним символами из P. Для этого заменим в строке a на 1, а остальные символы на 0. В шаблоне заменим все символы, совместимые с a, на 1, а остальные на — 0. Получим две битовых строки. Строку, полученную из шаблона, перепишем в обратном порядке (справа налево). Возьмем два многочлена, в которых коэффициентами будут элементы полученных битовых строк, и перемножим их. Это действие называется *булевой конволюцией*. Алгоритм Шенхаге-Штрассена, использующий быстрое преобразование Фурье, решает эту задачу за время $O(n \log n)$ [3]. Откинув в полученном произведении первые $m-1$ и последние $m-1$ коэффициентов, мы получим массив количеств совпадений a с совместимыми ему символами для каждой подстроки длины m строки S. Наконец, просуммируем полученные для всех символов a массивы и проверим каждую ячейку итогового массива на равенство длине шаблона. Таким образом, общая сложность алгоритма есть $O(|\Sigma| n \log n)$.

Рассмотрим улучшение приведенного выше алгоритма для случая частичных строк. Закодируем каждый символ основного алфавита битовой строкой, а для джокера зарезервируем два кода: из одних нулей и из одних единиц. Длину кода выберем равной $k = \lceil \log(|\Sigma| + 2) \rceil$. Таким образом, при кодировании текст и шаблон заменяются на битовые строки длины kn и km, соответственно. Заменим в шаблоне все джокеры на строку из всех единиц, а в тексте на строку из всех нулей. Выполним булеву конволюцию. Аналогично, заменим все джокеры в шаблоне на строку из всех нулей, а в тексте — на строку из всех единиц. Еще раз посчитаем булеву кон-

волюцию. Шаблон совместим с подстрокой в том и только том случае, когда соответствующий коэффициент первой конволюции равен числу единиц в коде подстроки, а коэффициент второй конволюции — числу единиц в коде шаблона. Примерно на таких идеях основан алгоритм в [19].

В статье [37] данный алгоритм был доработан и было доказано, что для произвольного отношения совместимости можно найти все вхождения шаблона в текст за $O(cn \log n)$, где c — минимальное количество двудольных полных подграфов, объединение которых представляет собой граф конфликта (дополнение к двудольному графу, задающему отношение совместимости).

В [11] приведён алгоритм, решающий данную задачу для частичных строк независимо от алфавита за время $O(n \log n)$. В нём все символы строки воспринимаются как целые числа и используется конволюция многочленов с целыми коэффициентами.

3.3 Задача максимизации числа вхождений

Итак, у нас есть две повреждённых строки — текст S длины n и шаблон P длины m. Требуется найти такие неповреждённые строки v и u, где v совместима с S, а u совместима с P, чтобы количество вхождений u в v было максимально. Фактически, требуется в каждой из строк S, P заменить каждый повреждённый символ на какой-то из совместимых с ним неповреждённых. Таким образом мы и получим строки v и u соответственно.

Ниже будут представлены полиномиальные по времени алгоритмы для двух частных случаев. В общем виде задача остается NP-трудной даже для случая частичных строк над бинарным алфавитом.

3.3.1 Решение для неповреждённого текста

Пусть текст S не повреждён. Сформулируем простейший алгоритм решения. Переберем i от 1 до $n-m+1$. Для фиксированного i проверим, может ли шаблон входить начиная с i. Если может, то считаем, что одно из вхождений будет начиная с i. В таком случае автоматически определяются замены всех повреждённых символов шаблона на символы алфавита. Теперь текст и шаблон уже неповреждённые строки. Эта задача решается за время $O(n + m)$, например, с помощью префикс-функции. Затем из всех решений (для каждого i) нужно выбрать наилучшее. Итого общая временная сложность решения $O((n - m)(n + m)) = O(n^2)$.

Покажем теперь, что можно добиться более оптимального решения.

Предложение 3.3.1. *Существует алгоритм, решающий задачу 1 для случая неповреждённого текста за время $O(n \log n)$ для частичных строк и за время $O(|\Sigma| n \log n)$ для произвольного отношения совместимости.*

Доказательство. Применим упомянутый в разделе 3.2.2 алгоритм из [38] (для случая частичных строк алгоритм из [11]) и найдем все вхождения шаблона в строку за время, указанное в условии предложения. Теперь вычислим для текста суффиксный массив и массив LCP(см. раздел 1.2.6). Это можно сделать за время $O(n)$ [30, 39]. Все подстроки длины m в тексте — это в точности все префиксы длины m суффиксов текста. Значения в массиве LCP разделяют суффиксный массив на блоки суффиксов, имеющих один и тот же префикс длины m. При любом восстановлении шаблона он совпадет только с одним таким префиксом. Среди блоков, пересекающихся со множеством позиций, найденных поиском по шаблону, найдем длиннейший. Общий префикс этого блока и будет результатом оптимального восстановления шаблона. □

3.3.2 Решение для неповреждённого шаблона

Предложение 3.3.2. *Существует алгоритм, решающий задачу 1 для случая неповреждённого шаблона за время $O(|\Sigma| n \log n + tm)$ (для частичных строк $O(n \log n + tm)$) , где t — число вхождений шаблона в текст.*

Доказательство. Будем решать задачу методом динамического программирования.

Напомним, что собственный суффикс строки, равный её равный префиксу, называется *гранью* строки. Вычислив для шаблона P префикс-функцию (см. определения в п. 1.2.1), мы можем получить длины всех граней строки P: это, в порядке убывания, числа $\pi(|P|), \pi(\pi(|P|)), \ldots$. Известно [31], что префикс-функцию можно вычислить за линейное время, т.е. длины всех граней P вычисляются за $O(m)$.

Пусть $a_1 < a_2 < \ldots < a_t$ — последовательность всех индексов, с которых начинаются вхождения шаблона P в текст S; эти индексы можно найти за $O(|\Sigma| n \log n)$ (а для частичных строк за $O(n \log n)$) с помощью алгоритма из [38] (для частичных алгоритмом из [11]).

Рассмотрим некоторый индекс a_i и все возможные неповреждённые строки, совместимые с исходным текстом S и тоже содержащие вхождение шаблона P, начиная с a_i. Среди всех таких

неповреждённых строк найдем такую строку v, для которой количество вхождений шаблона P в префикс длины $a_i + m - 1$ максимально. Обозначим это максимальное количество через f_i.

Имеем $f_1 = 1$ по определению a_1. Скажем, что вхождения шаблона P в позициях a_i и a_j *согласованы*, если неповреждённая строка, совместимая с S, может содержать оба указанных вхождения P. Тогда при $i > 1$ получаем

$$f_i = 1 + \max\{f_j \mid j < i, \text{такие, что вхождения в } a_i \text{ и в } a_j \text{ совместимы}\}. \tag{3.1}$$

При $j < i$ согласованность эквивалентна выполнению одного из двух условий: $a_i - a_j \geq m$ или $m - a_i + a_j$ — грань P. Таким образом, согласованность проверяется за константное время. Теперь заметим, что если $i > 2m$, то вхождение шаблона в позиции a_{i-m} согласовано как с вхождением в a_i, так и со вхождением в a_{i-2m}. Тогда $f_{i-m} \geq 1 + f_{i-2m}$, а значит, при вычислении f_i максимум не может быть достигнут на $j = i-2m$ (и на меньших значениях j). Следовательно, при вычислении f_i можно добавить условие $j > i - 2m$, что гарантирует вычисление f_i за $O(m)$ операций.

Осталось заметить, что требуемое в Задаче 1 количество вхождений есть $\max\{f_i \mid 1 \leq i \leq t\}$ и для каждого из вхождений мы сделали не более $O(m)$ действий. Таким образом, на вычисление f и последующее получение ответа потребовалось время $O(tm)$. □

3.3.3 Доказательство NP-трудности в общем случае

Для доказательства NP-трудности задачи 1 сведем задачу о поиске максимальной клики в графе к задаче 1 над бинарным алфавитом (при этом, очевидно, повреждённый символ может быть только один — джокер). Пусть у нас есть граф из n вершин и m ребер. Возьмем граф дополнения и занумеруем его ребра от 1 до $k = \frac{n(n-1)}{2} - m$. Поставим в соответствие каждой вершине j частичную строку w_j длины k над алфавитом $0, 1, \diamond$. Пусть ребро с номером i соединяет вершины a и b. Тогда $w_a[i] = 1$, $w_b[i] = 0$ (или наоборот) и $w_j[i] = \diamond$ для $j \notin a, b$.

Лемма 3.3.1. *В исходном графе ребро между двумя вершинами a и b существует тогда и только тогда, когда w_a совместима с w_b.*

Доказательство. По построению очевидно, что w_a совместима с w_b тогда и только тогда, когда ни одно из ребер графа дополнения не равно (a, b). □

Пусть вершины графа занумерованы от 1 до n. Положим $v_j = 1w_j10^{k-1}, S = v_1...v_n, P = 1\diamond^k1$.

Лемма 3.3.2. *Все вхождения шаблона P в текст S начинаются с индексов вида $1+i(2k+3)$, где i от 0 до $n-1$.*

Доказательство.

$$S = 1\underbrace{w_1}_{k}1\underbrace{00\ldots0}_{k+1}\ldots1\underbrace{w_n}_{k}1\underbrace{00\ldots0}_{k+1}, \tag{3.2}$$

$$P = 1\underbrace{\diamond\diamond\ldots\diamond}_{k}1. \tag{3.3}$$

Очевидно, что с каждой позиции указанного вида начинается вхождение P в S, а других вхождений нет из-за длинных блоков нулей. □

Лемма 3.3.3. *Пусть дано множество частичных строк и полная строка S, совместимая с каждой частичной строкой множества. Тогда все строки множества попарно совместимы.*

Доказательство. Очевидно, что длины всех частичных строк в множестве равны. Рассмотрим некоторое i от 1 до $|S|$. Тогда $S[i]$ — неповреждённый символ, совместимый с i-ми символами всех строк нашего множества. Значит, эти символы либо совпадают с $S[i]$, либо являются джокерами. □

Теорема 3.3.1. *Задача восстановления текста и шаблона с максимизацией количества вхождений шаблона в текст NP-трудна даже в случае бинарного алфавита.*

Доказательство. Напомним, что вхождения шаблона в текст мы называем согласованными, если существует способ восстановить текст и шаблон так, чтобы все эти вхождения сохранились. По лемме 3.3.2 все вхождения шаблона P в текст S начинаются только с индексов $1+i(2k+3)$, где $i = 0,\ldots,n-1$. Поскольку шаблон имеет длину $k+2$, вхождения согласованы. Наша задача состоит в том, чтобы найти наибольшую клику в заданном графе, что по лемме 3.3.1 эквивалентно поиску наибольшего подмножества попарно совместимых строк из множества $\{w_1, w_2, \ldots, w_n\}$. Заметим, что w_i совместимо с w_j тогда и только тогда, когда v_i совместимо с v_j. Теперь пусть мы решили задачу максимизации вхождения шаблона P в текст S. Другими словами, мы нашли такую полную строку u, совместимую с P, что количество совместимых с ней строк из множества $\{v_1, v_2, \ldots, v_n\}$ максимально. Заметим, что из того, что v_i совместимо с u и v_j совместимо с u, по лемме 3.3.3 следует совместимость v_i с v_j. Значит, мы решили исходную задачу. Таким образом, за полиномиальное время задача о поиске максимальной клики графа (которая, как

известно, NP-трудна, а как задача распознавания NP-полна) сведена к задаче 1 над бинарным алфавитом. Теорема доказана. □

3.4 Задача минимизации расстояния Хэмминга

Определим $t(S, P)$ — *непохожесть* текста S длины n и шаблона P длины m — следующим образом. Возьмем все подстроки длины m в тексте S. Найдем расстояние Хэмминга от каждой такой подстроки до шаблона P. Их сумма и будет непохожестью шаблона и текста.

Напомним, что в задаче 2 даны две повреждённых строки: текст S и шаблон P. Требуется найти неповреждённые строки v и u, совместимые с S и P, соответственно, и такие, что $t(v, u)$ минимальна.

Сопоставим паре (S, P) двудольный граф $G_{S,P}$ следующим образом: в первой доле n вершин, во второй m, будем соединять ребром вершину i первой доли и j второй доли тогда и только тогда, когда соответствующие им позиции в тексте и шаблоне накладываются друг на друга при совмещении шаблона P и некоторой подстроки текста S. Переформулируем задачу 2 в терминах графа $G_{S,P}$. Сопоставим каждому символу a повреждённого алфавита цвет. Покрасим вершину k первой доли в этот цвет, если $S[k] = a$, аналогично для вершин второй доли; полученную раскраску назовем *исходной*. Назовем раскраску *полной*, если все вершины покрашены в цвета, сопоставленные неповреждённым символам. Назовем две раскраски *совместимыми*, если для каждой вершины графа цвета, в которые она покрашена в этих раскрасках, соответствуют совместимым символам. Тогда задача 2 формулируется следующим образом: *из всех полных раскрасок графа $G_{S,P}$, совместимых с исходной раскраской, найти такую, в которой количество ребер, инцидентных вершинам разных цветов, минимально.*

3.4.1 Случай неповреждённого текста

Предложение 3.4.1. *Существует алгоритм, решающий задачу 2 для неповреждённого текста за время $O(n + |\Sigma|m)$.*

Доказательство. Для повреждённого символа в i-й позиции шаблона лучший вариант замены — тот из совместимых символов, который встречается чаще всего в строке $S[i..n - m + i]$. Заведем массив длины $|\Sigma|$ для счетчиков неповреждённых символов. Посчитаем с их помощью все символы S с первого по $(n - m)$-й. После этого для каждого i от 1 до m увеличиваем счётчик для символа $S[n - m + i]$ и уменьшаем для $S[i - 1]$. После этого, если $P[i]$ повреждён, то за линейный пробег по счётчикам выберем символ, совместимый с $P[i]$, значение счётчика для которого максимально, и заменим $P[i]$ на этот символ. Каждый символ текста будет ровно

один раз добавлен к счетчику и не более одного раза отнят. Таким образом, общее время на обновление счётчиков $O(n)$, на получение ответа для всех символов — $O(m|\Sigma|)$. □

Замечание 3.4.1. *Для случая частичных строк значение повреждённого символа каждый раз выбирается среди всех символов алфавита. Поэтому мы можем хранить количества вхождений символа не в массиве, а в сбалансированном дереве (см. раздел 1.2.6). Тогда изменение и извлечение максимума и добавление/удаление элемента будет выполняться за время $O(\log|\Sigma|)$, а первоначально сбалансированное дерево из первых $(n-m)$ символов строится за линейное время сортировкой подсчётом. Общее время работы алгоритма в этом случае $O(n+m\log|\Sigma|)$.*

3.4.2 Случай неповреждённого шаблона

Предложение 3.4.2. *Существует алгоритм, решающий задачу 2 для неповреждённого шаблона за время $O(|\Sigma|n)$.*

Доказательство. Заведем массив длины $|\Sigma|$ для счетчиков неповреждённых символов. После этого для каждого i от 1 до n увеличиваем счётчик для символа $P[i]$ (если $i \le m$), либо уменьшаем $P[m-(n-i)]$ (если $i > n-m$). Если $S[i]$ повреждён, то за линейный пробег по счётчикам выберем символ, совместимый с $S[i]$, значение счётчика для которого максимально, и заменяем $S[i]$ на него. Каждый символ P будет ровно один раз добавлен и ровно один раз удалён. Таким образом, общее время на обновление счётчиков $O(m)$, на получение ответа для всех символов — $O(n|\Sigma|)$. □

Замечание 3.4.2. *Для частичных строк аналогично случаю неповреждённого текста можно решить задачу за общее время $O(m\log|\Sigma|+n)$ с помощью бинарной кучи (heap), которая позволяет добавлять/удалять элементы за логарифмическое время, а извлекать максимум за константное.*

3.4.3 Случай частичных строк с циклическим текстом

Циклической строкой называется последовательность символов, упорядоченная не линейно, а циклически. Тем самым, циклическая строка получается из обычной «склеиванием» концов. Подстрока циклической строки — это обычная строка, отличие только в том, что циклическая строка $u[1..n]$ имеет, для любых $i < j \le n$, не только подстроку $u[i..j]$, но и подстроку $u[j..i] = u[j..n]u[1..i]$. Циклическая строка — естественный комбинаторный объект. Свойства таких строк

исследовались в ряде работ (см., например, [7, 8, 24, 47] и демонстрируют как существенные сходства, так и существенные отличия со свойствами обычных строк. Рассматриваемая нами задача очень сильно упрощается в случае частичных строк, если текст — циклическая строка.

Предложение 3.4.3. *Существует алгоритм, решающий задачу 2 для случая, когда шаблон — частичная строка, а текст — частичная циклическая строка, за время* $O(n)$.

Доказательство. Заметим, что в данном случае граф $G_{S,P}$ будет полный. А значит, все списки смежности вершин первой доли совпадают. Тогда существует оптимальное решение, в котором все вершины первой доли, сопоставленные повреждённым символам, т.е. джокерам, покрашены в один цвет; то же самое верно и для второй доли.

Вычислим для каждого $a \in \Sigma \cup \Gamma$ количества $C_S(a)$ и $C_P(a)$ вхождений a в текст и в шаблон соответственно; требуемое время — $O(n + m + |\Sigma|) = O(n)$.

Далее, за $O(|\Sigma|)$ вычислим символы a_1, a_2 и a_3, на которых достигается максимум $C_S(a)$, $C_P(a)$ и $C_S(\diamond)C_P(a) + C_P(\diamond)C_S(a)$ соответственно. Если присвоить джокерам в тексте и шаблоне одно и то же значение, то максимальное количество полученных за счет этого совпадений символов равно $C_S(\diamond)C_P(a_3) + C_P(\diamond)C_S(a_3) + C_S(\diamond)C_P(\diamond)$. Если же двум указанным группам джокеров присвоены разные значения, то максимальное количество полученных совпадений есть $C_S(\diamond)C_P(a_2) + C_P(\diamond)C_S(a_1)$. Сравнив два полученных количества, выберем оптимальный вариант присвоения. □

3.4.4 Случай бинарного алфавита

Предложение 3.4.4. *Для случая бинарного алфавита Задача 2 полиномиально сводится к задаче о разрезе.*

Доказательство. Как уже упоминалось в разделе 3.3.3, повреждённый символ в случае бинарного алфавита является джокером.

Пусть в графе $G_{S,P}$ существует путь из вершины первого цвета в вершину второго цвета. Тогда легко видеть, что хотя бы одно ребро каждого такого пути будет в итоге (т.е. после раскраски всех джокеров в первый и второй цвет) иметь разноцветные концы. Пусть мы удалили некоторое количество ребер так, чтобы не существовало пути между двумя разнопокрашенными вершинами. Тогда в каждой компоненте связности содержатся вершины только какого-то одного цвета и «прозрачные» (цвет джокера). Очевидно, что после этого нужно покрасить прозрачные

вершины в цвет, который присутствует в ее компоненте связности (а если такого нет, то в любой). Таким образом, все ребра графа будут иметь одинаково покрашенные концы. Значит, наша задача свелась к следующей: *удалить из графа минимальное количество ребер, чтоб не существовало путей между некоторыми двумя вершинами разных цветов.*

Это — в точности задача о минимальном разрезе между двумя множествами вершин. Решим ее следующим образом. Объединим все вершины первого цвета в одну и назовем ее стоком (ребра, которые были смежны с вершинами первого цвета, будут смежны со стоком). Аналогично все вершины второго цвета объединим в исток. Присвоим всем ребрам полученного графа единичный вес. Найдем минимальный разрез между вершинами сток и исток. Для получения ответа к исходной задаче, прозрачные вершины, из которых достижим сток после удаления ребер разреза, раскрашиваются в первый цвет, а остальные — во второй. □

Следствие 3.4.1. *Задача 2 для бинарного алфавита разрешима за полиномиальное время.*

Доказательство. По предложению 3.4.4 задача сводима к задаче о разрезе.

Известно [20], что задача о разрезе полиномиально сводима к задаче нахождения максимального потока в графе.

Таким образом мы показали, что для рассматриваемой задачи существует полиномиальный алгоритм, например, ее можно решить алгоритмом Форда-Фалкерсона за $O(m^2n)$. □

3.4.5 Случай частичных строк и задача о мультиразрезе

Аналогично предыдущему разделу, мы можем свести задачу 2 для случая частичных строк над произвольным алфавитом к нахождению минимального мультиразреза в графе. Пусть в графе $G_{S,P}$ существует простая цепь, содержащая не менее двух вершин, где концевые вершины покрашены в разные цвета, а все промежуточные вершины — прозрачные. В нашей задаче необходимо покрасить все вершины так, чтобы максимизировать количество ребер с одноцветными концами. В каждой такой цепи есть ребро с разноцветными концами. Значит, нужно удалить минимальное количество ребер из графа так, чтоб не существовало пути между двумя покрашенными вершинами разных цветов.

Объединим все вершины одного цвета в одну: вместо всех вершин данного цвета (удалим их) добавим одну новую, соединив ее ребрами с теми вершинами, с которыми была соединена хотя бы одна из исходных. После этого у нас в графе останутся покрашенные и прозрачные вершины,

причем нет двух вершин одного цвета. Теперь в данном графе необходимо удалить минимальное количество ребер, чтобы не существовало путей между двумя покрашенными вершинами. Эта задача и называется задачей о минимальном мультиразрезе. Она является NP-трудной (в варианте распознавания — с вопросом «существует ли мультиразрез из k ребер?» — NP-полной).

Гипотеза 3.4.1. *Задача восстановления текста и шаблона с минимизацией непохожести строки и шаблона для случая частичных строк NP-трудна.*

Замечание 3.4.3. *Для задачи о мультиразрезе существует приближенный полиномиальный алгоритм, находящий мультиразрез, мощность которого менее чем вдвое превышает мощность минимального мультиразреза [14]. Следовательно, приближенный алгоритм с такими же характеристиками есть и для задачи 2 в случае частичных строк.*

3.4.6 Доказательство NP-трудности в общем случае

Предложение 3.4.5. *Задача 2 в общем случае NP-трудна как для обычного, так и для циклического текста.*

Доказательство. Начнем с циклического текста. Напомним, что в данном случае граф $G_{S,P}$ — полный. Значит, порядок следования символов в строках не влияет на решение, что дает возможность рассматривать строки как мультимножества символов. В отличие от случая частичных строк, нам для каждой вершины графа дополнительно дано множество допустимых для нее цветов (т.е. совместимых с данным символом символов из Σ). Пару, состоящую из восстановленного текста S' и восстановленного шаблона P' мы отождествляем с мультимножеством пар $(S'[i], P'[j])$, имеющим мощность mn. Процесс восстановления состоит в выборе допустимого цвета для каждой вершины, а «показателем качества» восстановления является число пар вида (a, a).

Назовем Задачу 2 дизъюнктной, если для любой пары натуральных $1 \leq i < j \leq n$ символы $S[i]$ и $S[j]$ несовместимы. Покажем, что даже дизъюнктная задача 2 для циклического текста NP-трудна.

Замечание 3.4.4. *Если наложить условие дизъюнктности и на шаблон, то задача очевидно решится построением максимального паросочетания за время, кубическое от размера алфавита (который в данном случае асимптотически равен суммарной длине обоих строк).*

Из отсутствия совместимых символов в тексте следует, что каждый символ шаблона (после восстановления) может образовать пару равных элементов максимум с одним символом текста (символ, образующий такую пару, назовем *удовлетворенным*). В частности, показатель качества решения задачи теперь не превосходит длины шаблона.

Рассмотрим задачу в еще более частном случае: каждый символ текста повреждён и совместим ровно с двумя символами основного алфавита. Обозначим эти символы для $S[i]$ за x_i и $\bar{x}_i$. Таким образом, восстановление $S[i]$ состоит в присвоении значения булевой переменной. Каждому символу шаблона сопоставим дизъюнкцию литералов, соответствующих допустимым цветам для этого символа. Эта дизъюнкция истинна тогда и только тогда, когда символ шаблона удовлетворен. Тем самым, задача максимизации числа удовлетворенных символов шаблона — это в точности задача $maxSAT$ с n переменными и m элементарными дизъюнкциями (клозами). Из NP-трудности $maxSAT$ вытекает NP-трудность нашей задачи. Ниже мы даем точную формулировку задачи $maxSAT$ и строгое сведение ее к частному случаю Задачи 2.

Задача maxSAT (или задача об оптимальной выполнимости) формулируется следующим образом. Дано m выражений (клозов), каждое из них — дизъюнкция нескольких литералов (переменная, либо ее отрицание), всего в них участвует не более n булевых переменных. Требуется выбрать значения переменных таким образом, чтоб максимизировать количество истинных клозов.

Пусть дан пример задачи $maxSAT$ с переменными $x_1, \ldots, x_n$ и клозами $C_1, \ldots, C_m$. Построим по ней пример Задачи 2 для циклического текста. Положим $\Sigma = \{x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n\}$, $\Gamma = \{b_1, \ldots, b_n, c_1, \ldots, c_m\}$, причем каждый повреждённый символ b_i совместим в точности с литералами x_i и $\bar{x}_i$, а каждый повреждённый символ c_i — с литералами, входящими в клоз C_i. Далее, $S = (b_1 \ldots b_n)$ (циклическая строка), $P = c_1 \ldots c_m$.

Пусть теперь мы решили построенный пример Задачи 2. Тогда восстановленный текст дает решение для $maxSAT$ (если $b_i = x_i$, присваиваем $x_i = 1$, а если $b_i = \bar{x}_i$, то $x_i = 0$), а показатель качества восстановления равен числу истинных клозов. Задача в случае циклического текста NP-трудна.

Для доказательства теоремы осталось свести задачу о циклической строке к задаче в общем виде. Этого можно добиться, расширив основной алфавит на один символ (несовместимый ни с одним повреждённым) и приписав m таких символов слева и справа к тексту. Очевидно, после этого каждый символ текста совместится с каждым символом шаблона, при этом совмещение с

новым символом никак не повлияет на ответ. Теорема доказана. □

Замечание 3.4.5. *В класс сложности APX входят все задачи оптимизации, для которых существует полиномиальный алгоритм, приближенно решающий данную задачу с ограниченной относительной погрешностью приближения оптимизируемого критерия (В случае задачи максимизации: существует константа c,* $0 < c < 1$ *такая, что алгоритм находит решение со значением критерия не меньше* cM*, если оптимальное значение равно* M*). Задача maxSAT, эквивалентная рассмотренному выше специальному случаю задачи 2, является APX-полной. Следовательно, задача 2 является APX-трудной, а в случае частичных строк, с учётом замечания 3.4.3, принадлежит классу APX.*

Для удобства, полученные по задачам 1 и 2 результаты сведем в таблицу.

Вариант	Задача 1	Задача 2
Неповреждённый текст	Поиск повреждённого шаблона + суффиксный массив, $O(n\|\Sigma\|\log n)$ для повреждённых и $O(n\log n)$ для частичных	Жадный алгоритм, $O(n+m\|\Sigma\|)$ для повреждённых $O(n+m\log\|\Sigma\|)$ для частичных
Неповреждённый шаблон	Динамическое программирование + префикс-функция, $O(n\log n+mt)$ для частичных, $O(n\|\Sigma\|\log n+mt)$ для повреждённых	Жадный алгоритм, $O(n\|\Sigma\|)$ для повреждённых $O(n+m\log\|\Sigma\|)$ для частичных
Бинарный алфавит	NP-трудна	Минимальный разрез
Частичные строки	NP-трудна	NP-трудна (гипотеза)
Частичные строки с циклическим текстом		Жадный алгоритм, $O(n)$
Циклический текст		NP-трудна
Общий случай	NP-трудна	NP-трудна

Таблица 3.1: Сравнение решений двух задач главы. Задача 1 для циклического текста принципиально не отличается от задачи для обычного, поэтому не рассматривалась отдельно.

Заключение

В заключении, представив краткое резюме по каждой из двух основных глав, обсудим пути дальнейшей возможной работы в соответствующих направлениях и сформулируем связанные оставшиеся открытыми проблемы.

Палиндромы

В данной главе мы описали новую компактную и быструю структуру данных — овердрево. Рассмотрели его различные модификации, которые более эффективны в использовании времени, либо памяти. Одна из модификаций позволяет сделать овердрево с откатами, другая делает его персистентным. Разобрали множество возможных применений, получив теоретически быстрые и практически легко реализуемые алгоритмы для ряда комбинаторных задач. Самыми существенными из представленных применений являются простая реализация разбиения строки на палиндромы, перечисление палиндромно-насыщенных строк, а также поиск всех различных подпалиндромов строки.

Дальнейшие исследования в этом направлении связаны с поиском новых приложений овердрева, а также с гипотезой о возможности линейного поиска палиндромной длины и с открытой проблемой об оптимальной реализации овердрева.

Повреждённые строки

В данной работе рассмотрены две задачи о восстановлении повреждений в тексте и шаблоне.

Для обеих задач общий случай оказался NP-трудным, а случаи неповреждённого текста и неповреждённого шаблона — эффективно разрешимыми. При этом во второй задаче, где критерием восстановления является минимизация суммарного расстояния Хэмминга, полиномиально разрешимых частных случаев оказалось больше.

Отдельно отметим, что для решения различных частных случаев использованы совершенно

разные полиномиальные алгоритмы, основанные на жадных стратегиях, преобразовании Фурье, динамическом программировании и потоках в сетях.

Дальнейшее исследование предполагает продолжение изучения задачи о минимизации непохожести шаблона и текста для частичных строк (частный случай, оставшийся открытым), а также выделение дополнительных частных случаев обеих задач с целью выявления более простых подходов к решению.

Список литературы

[1] В. Л. Арлазаров, Е. А. Диниц, М. А. Кронрод, И. А. Фараджев . Об экономном построении транзитивного замыкания ориентированного графа // Доклады АН СССР. — 1970. — Т. 134, № 3. — С. 1209–1210.

[2] Гасфилд Д. Строки, деревья и последовательности в алгоритмах. — Невский диалект СПб., 2003.

[3] Кнут Д. Искусство программирования Т. 2. Получисленные алгоритмы, 3-е изд. — 2000. — Т. 832. — С. 6–1.

[4] Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: Построение и анализ [пер. с англ.] // — Издательский дом Вильямс, 2009.

[5] Рубинчик М. В., Гамзова Ю. В. Две задачи о восстановлении поврежденных строк // Сибирские электронные математические известия. — 2013. — Т. 10. — С. 538–550.

[6] Шур А. М., Гамзова Ю. В. Частичные слова и свойство взаимодействия периодов // Известия Российской академии наук. Серия математическая. — 2004. — Т. 68, № 2. — С. 191–214.

[7] Aberkane A., Currie J. D. Attainable lengths for circular binary words avoiding k-powers // Bull. Belg. Math. Soc. Simon Stevin. — 2005. — Vol. 12, no. 4. — P. 525–534.

[8] Aberkane A., Currie J. D. The Thue-Morse word contains circular $(5/2)^+$-power-free words of every length // Theoret. Comput. Sci. — 2005. — Vol. 332. — P. 573–581.

[9] Berstel J., Boasson L. Partial words and a theorem of Fine and Wilf // Theoret. Comput. Sci. — 1999. — Vol. 218. — P. 135–141.

[10] B. Blakeley, F. Blanchet-Sadri, J. Gunter, N. Rampersad. On the complexity of deciding avoidability of sets of partial words // Theor. Comput. Sci. — 2010. — Vol. 411, no. 49. — P. 4263–4271.

[11] Clifford P., Clifford R. Simple deterministic wildcard matching // Information Processing Letters. — 2007. — Vol. 101, no. 2. — P. 53–54.

[12] Crochemore M., Hancart C., Lecroq T. Algorithms on strings. — Cambridge University Press, 2007.

[13] Crochemore M., Rytter W. Jewels of stringology. — World Scientific Publishing Co. Pte. Ltd., 2002.

[14] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, M. Yannakakis. The complexity of multiterminal cuts // SIAM Journal on Computing. — 1994. — Vol. 23, no. 4. — P. 864–894.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan. Making data structures persistent // Proceedings of the eighteenth annual ACM symposium on Theory of computing / ACM. — 1986. — P. 109–121.

[16] Droubay X., Justin J., Pirillo G. Episturmian words and some constructions of de Luca and Rauzy // Theoretical Computer Science. — 2001. — Vol. 255. — P. 539–553.

[17] G. Fici, T. Gagie, J. Kärkkäinen, D. Kempa. A subquadratic algorithm for minimum palindromic factorization // J. of Discrete Algorithms. — 2014. — Vol. 28. — P. 41–48.

[18] Fine N. J., Wilf H. S. Uniqueness theorems for periodic functions // Proceedings of the American Mathematical Society. — 1965. — Vol. 16, no. 1. — P. 109–114.

[19] Fischer M., Paterson M. String matching and other products // SIAM-AMS Proceedings. — 1974. — Vol. 7. — P. 113–125.

[20] Ford L. R., Jr, Fulkerson D. R. Flows in networks. — Princeton University Press, 1967.

[21] Galil Z. Open problems in stringology // Combinatorial Algorithms on Words. — Springer, 1985. — P. 1–8.

[22] Galil Z., Seiferas J. A Linear-Time On-Line Recognition Algorithm for "Palstar" // J. ACM. — 1978. — Vol. 25, no. 1. — P. 102–111.

[23] A. Glen, J. Justin, S. Widmer, L. Zamboni. Palindromic richness // European J. of Combinatorics. — 2009. — Vol. 30. — P. 510–531.

[24] Gorbunova I. A. Repetition Threshold for Circular Words // Electronic J. Combinatorics. — 2012. — Vol. 19, no. 4. — P. P11.

[25] Groult R., Prieur E., Richomme G. Counting distinct palindromes in a word in linear time // Information Processing Letters. — 2010. — Vol. 110. — P. 908–912.

[26] Guo C., Shallit J., Shur A. M. On the combinatorics of palindromes and antipalindromes. — arXiv:1503.09112 [cs.FL]. — 2015.

[27] V. Halava, T. Harju, T. Kärki, P. Séébold. Overlap-freeness in infinite partial words // Theoret. Comput. Sci. — 2009. — Vol. 410, no. 8-10. — P. 943–948.

[28] Idiatulina L. A., Shur A. M. Periodic partial words and random bipartite graphs // Fundam. Inform. — 2014. — Vol. 132, no. 1. — P. 15–31.

[29] Kärki T., Harju T., Halava V. Interaction properties of relational periods // Discrete Mathematics & Theoretical Computer Science. — 2008. — Vol. 10, no. 1.

[30] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications // Combinatorial pattern matching. — Vol. 2089 of LNCS. — Berlin : Springer, 2001. — P. 181–192.

[31] Knuth D. E., Morris J. H., Pratt V. R. Fast pattern matching in strings // SIAM J. on Computing. — 1977. — Vol. 6. — P. 323–350.

[32] Kosolobov D., Rubinchik M. An optimal online algorithm for finding all distinct subpalindromes of a string // Algorithms & Complexity. Abstracts of Reports and Other Materials of the 6th School “Computer Science Days in Ekaterinburg” A. S. Kulikov, A. M. Shur (eds.). — Ekaterinburg: Ural University Press, 2013. — P. 44–46.

[33] Kosolobov D., Rubinchik M., Shur A. M. Finding distinct subpalindromes online // Proc. Prague stringology conference 2013 / Ed. by Jan Holub, Jan Žďárek. — CTU, 2013. — P. 63–69.

[34] Kosolobov D., Rubinchik M., Shur A. M. Pal^k is linear recognizable online // SOFSEM 2015: Theory and Practice of Computer Science. — Springer-Verlag Berlin Heidelberg, 2015. — Vol. 8939 of LNCS. — P. 289–301.

[35] Manacher G. A new linear-time on-line algorithm finding the smallest initial palindrome of a string // J. ACM. — 1975. — Vol. 22, no. 3. — P. 346–351.

[36] Manea F., Mercaş R. Freeness of partial words // Theoret. Comput. Sci. — 2007. — Vol. 389, no. 1-2. — P. 265–277.

[37] Muthukrishnan S., Palem K. Non-standard stringology: algorithms and complexity // Proc. 26th Annual ACM Symposium on Theory of Computing (STOC'94). — New York : ACM, 1994. — P. 770–779.

[38] Muthukrishnan S., Ramesh H. String Matching Under a General Matching Relation // Proc. 12th Conference on Foundations of Software Technology and Theoretical Computer Science. — Vol. 652 of LNCS. — Berlin : Springer-Verlag, 1992. — P. 356–367.

[39] Nong Ge, Zhang Sen, Chan Wai Hong. Linear time suffix array construction using D-critical substrings // Combinatorial Pattern Matching / Springer. — 2009. — P. 54–67.

[40] Puglisi S. J., Smyth W. F., Turpin A. H. A taxonomy of suffix array construction algorithms // ACM Computing Surveys (CSUR). — 2007. — Vol. 39, no. 2. — P. 4.

[41] Ravsky O. On the palindromic decomposition of binary words // Journal of Automata, Languages and Combinatorics. — 2003. — Vol. 8, no. 1. — P. 75–83.

[42] Rubinchik M., Gamzova Y.V. Two pattern matching problems for strings with compatibility relations // V. Halava, J. Karhumaki, Y. Matiyasevich (Eds.), Proc. 2nd Russian Finnish Symposium on Discrete Mathematics (RuFiDiM II). — Vol. 17 of TUCS Lecture Notes. — Turku Centre for Computer Science, 2012. — P. 151–152.

[43] Rubinchik Mikhail, Shur Arseny M. On the number of distinct subpalindromes in words // Proc. 3rd Russian Finnish Symp. on Discrete Mathematics. Inst. Appl. Math. Research, Petrozavodsk. — 2014. — P. 96–98.

[44] Rubinchik Mikhail, Shur Arseny M. EERTREE: An Efficient data structure for processing palindromes in strings // Combinatorial algorithms: Proc. IWOCA 2015. — Vol. 9538 of LNCS. — Springer International Publishing, 2016. — P. 321–333.

[45] Rubinchik Mikhail, Shur Arseny M. The number of distinct subpalindromes in random words // Fundamenta Informaticae. — 2016. — Available at http://arxiv.org/abs/1505.08043.

[46] Sloane, N.J.A.: The on-line encyclopedia of integer sequences. — 1964–2016. — Available at http://oeis.org.

[47] Shur A. M. On ternary square-free circular words // Electronic J. Combinatorics. — 2010. — Vol. 17, no. # R140.

[48] Smyth W. Computing patterns in strings. — Pearson Education, 2003.

[49] Ukkonen E. On-line construction of suffix trees // Algorithmica. — 1995. — Vol. 14, no. 3. — P. 249–260.

[50] Valiant L. G. General context-free recognition in less than cubic time // J. of Computer and System Sciences. — 1975. — Vol. 10, no. 2. — P. 308–314.

[51] Problems of Asia–Pacific Informatics Olympiad 2014. — Available at http://olympiads.kz/apio2014/apio2014_problemset.pdf.

[52] Problems of the MIPT Fall Programming Training Camp 2014. Contest 12. — Available at https://drive.google.com/file/d/0B_DHLY8icSyNUzRwdkNFa2EtMDQ.

[53] Problems of Northern Grand Prix 2005. — Available at http://codeforces.com/gym/100222/attachments/download/1768/20052006-winter-petrozavodsk-camp-andrew-stankevich-contest-18-en.pdf.

[54] Problems of the Central European Regional Contest 2014. — Available at: https://cerc.tcs.uj.edu.pl/2014/data/cerc2014problems.pdf.

[55] Solutions of the problems of the Central European Regional Contest 2014. — Available at: https://cerc.tcs.uj.edu.pl/2014/data/cerc2014problems.pdf.

Список иллюстраций

Список таблиц