Федеральное государственное автономное образовательное учреждение высшего образования
**«Уральский Федеральный Университет
имени первого Президента России Б.Н.Ельцина»**
Институт естественных наук и математики
Кафедра алгебры и фундаментальной информатики

На правах рукописи

Меркурьев Олег Андреевич

Эффективные строковые алгоритмы в модели потока данных

Специальность 05.13.17 - теоретические основы информатики

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель
доктор физико-математических наук
профессор Шур Арсений Михайлович

Екатеринбург 2020

Оглавление

1 Введение **4**
- 1.1 Предварительные сведения . . . 5
  - 1.1.1 Строки . . . 5
  - 1.1.2 Асимптотические оценки сложности . . . 6
  - 1.1.3 Алгоритмы . . . 6
  - 1.1.4 Хэш Карпа–Рабина . . . 7
  - 1.1.5 Словари . . . 8
  - 1.1.6 Принцип Яо . . . 9
  - 1.1.7 Модель потока данных . . . 10
- 1.2 Обзор диссертации . . . 12
  - 1.2.1 Цели и задачи диссертации . . . 12
  - 1.2.2 Основные методы исследования . . . 12
  - 1.2.3 Структура диссертации и организация текста . . . 12
  - 1.2.4 Апробация и публикации . . . 13
  - 1.2.5 Основные результаты диссертации . . . 13

2 Палиндромы в потоках **17**
- 2.1 Введение . . . 17
- 2.2 Нижние оценки . . . 19
- 2.3 Алгоритмы реального времени . . . 19
  - 2.3.1 Аддитивная погрешность . . . 21
  - 2.3.2 Мультипликативная погрешность $\varepsilon \leq \mathbf{1}$ . . . 22
  - 2.3.3 Мультипликативная погрешность $\varepsilon > \mathbf{1}$ . . . 27

3 Повторы и обратные повторы в потоках **32**
- 3.1 Введение . . . 32
- 3.2 Определения . . . 32
- 3.3 Сведение к сжатым повторам . . . 35
- 3.4 Поиск наибольшего обратного повтора . . . 37
- 3.5 Поиск наибольшего повтора . . . 44
- 3.6 Нижние оценки . . . 46

4 Максимальные периодические подстроки в потоках 50
4.1 Введение . . . 50
4.2 Определения . . . 51
4.3 Инструменты . . . 53
4.3.1 Хэши, фреймы, чекпойнты . . . 53
4.3.2 Видимые периодические строки . . . 54
4.3.3 Свежие и чёрствые вхождения . . . 56
4.3.4 Структуры данных . . . 57
4.4 Алгоритм . . . 58
4.4.1 Удаление чекпойнтов . . . 59
4.4.2 Обновление групп . . . 60
4.4.3 Обнаружение периодических подстрок . . . 61
4.4.4 Обновление списка отслеживания . . . 64

Заключение 68

Список литературы 70

Список иллюстраций 75

Глава 1

Введение

Актуальность темы

Строка — один из центральных объектов компьютерных наук. Любая последовательность данных может рассматриваться как строка (последовательность символов). Такое абстрактное представление, не учитывающее структуру обрабатываемых данных, имеет очень широкую область применений — от поиска в базах данных до анализа структуры ДНК и белков. Раздел компьютерных наук, занимающийся алгоритмами работающими со строками, называется *стрингология* (англ. stringology от string — строка). Название было предложено в 1985 в работе [27]. Этой динамически развивающейся в настоящее время области посвящен ряд монографий [18, 22, 34, 55] и специализированных конференций с высоким уровнем цитируемости: Combinatorial Pattern Matching (CPM), String Processing and Information Retrieval (SPIRE) и др. Задачи, рассматриваемые в стрингологии, можно сгруппировать в несколько кластеров: поиск по образцу (pattern matching), индексирование данных, сжатие данных и алгоритмы на сжатых представлениях, а также поиск регулярных структур, таких как повторы, палиндромы, периодические фрагменты и различные их обобщения. Последнему кластеру принадлежат задачи, рассматриваемые в диссертации.

Одной из активно исследуемых областей являются строковые алгоритмы в модели потока данных. В этой модели элементы последовательности данных поступают один за одним и не могут быть сохранены в связи с малым объёмом доступной памяти. Первые работы, рассматривающие подобные ограничения, появились в 1970-е годы [52]. При этом понятие модели потока данных было сформулировано в работе [1], авторы которой получили в 2005 г. премию Гёделя за основополагающие исследования в области потоковых алгоритмов. Интересной особенностью модели потока данных является то, что в ней зачастую удаётся получить нижние оценки на вычислительную сложность задач, в первую очередь на необходимый объём памяти.

Первый значительный результат о строковых алгоритмах в модели потока данных был получен в [53]. В последнее десятилетие модель потока данных является заметным трендом в стрингологии. В настоящей работе рассматриваются

строковые задачи, связанные с поиском регулярных структур в потоке данных.

1.1 Предварительные сведения

1.1.1 Строки

Строка длины n — это отображение $\{1, \ldots, n\} \to \Sigma$, где Σ — это некоторое конечное множество, называемое алфавитом. Элементы алфавита Σ называются символами. Длина строка w обозначается как $|w|$. В данной работе мы ограничимся рассмотрением целочисленных алфавитов полиномиального размера от длины строки[1], то есть $\Sigma = \{1, 2, \ldots, \sigma\}$, и при этом $\sigma \in \mathcal{O}(\mathrm{poly}(n))$. Через $w[1], w[2], \ldots, w[n]$ обозначаются 1-й, 2-й, …, n-й символы строки w соответственно. Для произвольных целых чисел i и j обозначим $w[i..j] = w[i]w[i+1]\ldots w[j]$; в частности, $w = w[1..n]$. Строка $\overleftarrow{w} = w[n]\ldots w[2]w[1]$ называется строкой, *обратной* к w.

Строка $p = w[i..j]$ называется *подстрокой* строки w. В этом случае строка p *имеет вхождение* в позиции i. Подстроки вида $w[1..j]$ называются *префиксами* строки, а подстроки вида $w[i..n]$ называются *суффиксами*. Конкатенацией строк v и w называется строка $vw = v[1]v[2]\ldots v[|v|]w[1]w[2]\ldots w[|w|]$. Для $k > 1$, k-ой степенью строки w называется строка w^k, равная конкатенации k копий строки w. Строка называется *примитивной*, если она не является степенью более короткой строки.

Палиндромы

Палиндромом называется строка совпадающая с обратной к себе ($w = \overleftarrow{w}$). Ясно, что если строка w является палиндромом и $|w| > 2$, то строка $w[2..|w|-1]$ также является палиндромом. Рассматривая подстроки w, являющиеся палиндромами, будем говорить, что $w[i..j]$ и $w[h..k]$ имеют один и тот же *центр*, если $i+j = h+k$. Так для заданного центра можно определить наибольший палиндром, и тогда все меньшие подстроки, имеющие тот же центр, будут палиндромами.

Повторы

Повтор в строке w — это пара равных подстрок $w[i..i+l-1] = w[j..j+l-1]$, где $i < j$. Такой повтор будем обозначать тройкой (i, j, l). *Обратный повтор* — это пара подстрок, обратных друг к другу, т.е. $w[i..i+l-1] = \overleftarrow{w[j..j+l-1]}$, где $i \le j$. Такой обратный повтор также будем индексировать тройкой (i, j, l). Заметим, что палиндром $w[i..i+l-1]$ является обратным повтором (i, i, l).

[1] Такой тип алфавита используется в строковых алгоритмах наиболее часто; полиномиальное ограничение выполняется для большинства практических видов данных и позволяет реализовывать некоторые алгоритмы быстрее, чем в общем случае (например, сортировку массива за линейное время).

Периодические подстроки

У строки w есть *период* p, если $w[i] = w[i+p]$ для всех подходящих i ($1 \le i \le |w| - p$). Особое значение имеет *минимальный период*. Характеристика периодичности строки, *экспонента*, определяется как отношение длины строки к минимальному периоду. Строка называется *периодической*, если её экспонента не меньше 2.

При конкатенации периодической строки и одного символа (слева или справа), строка либо остаётся периодической с тем же периодом, либо перестаёт быть периодической; этот факт следует из теоремы Файна–Вильфа [24]. Таким образом естественно определяются максимальные по включению периодические подстроки. Для краткости, для максимальных периодических подстрок будем использовать термин *ран* (кальку общепринятого английского термина run).

1.1.2 Асимптотические оценки сложности

Для асимптотической оценки вычислительной сложности алгоритмов в работе используются общепринятые обозначения: $\mathcal{O}$, Θ, o, Ω. Напомним их определения. Через $\mathcal{F}$ обозначим множество неотрицательных функций целочисленного аргумента. Пусть $f \in \mathcal{F}$. Тогда $\mathcal{O}(f)$ это множество всех таких $g \in \mathcal{F}$, что $g(n) \le c_g f(n)$ для всех $n > N_g$, где c_g и N_g — некоторые константы, зависящие от g. Тогда $\Omega(f) = \{g \in \mathcal{F} \colon f \in O(g)\}$ и $\Theta(f) = \Omega(f) \cap O(f)$. Через $o(f)$ обозначим множество всех $g \in \mathcal{F}$ таких, что $g(n) = \alpha_g(n) f(n)$ для всех $n > N_g$, где N_g — некоторое число, зависящее от g, а $\alpha_g(n)$ — некоторая функция, зависящая от g, такая, что $\lim\limits_{n\to\infty} \alpha_g(n) = 0$. Вместо $g \in O(f)$, $g \in \Omega(f)$, $g \in \Theta(f)$, $g \in o(f)$ часто пишут $g = O(f)$, $g = \Omega(f)$, $g = \Theta(f)$, $g = o(f)$ соответственно.

Также нам понадобятся аналоги введённых обозначений для случая неотрицательных функций с двумя и тремя параметрами. Например, для данной неотрицательной функции двух целочисленных аргументов $f(m,n)$ через $\mathcal{O}(f(m,n))$ будем обозначить множество неотрицательных функций $g(m,n)$ таких, что $g(m,n) \le c_g f(m,n)$ для всех $m > M_g$ и $n > N_g$, где c_g, M_g, N_g — некоторые константы, зависящие от g. Остальные определения аналогичны. Детальное описание способов работы с этими обозначениями можно найти, например, в учебнике [17].

1.1.3 Алгоритмы

Мы используем в алгоритмах стандартный набор операций, как, например, в языке C. Таким образом, за исключением представления входных данных в виде потока (см. раздел 1.1.7), используется обычная RAM-модель вычислений. В главе 4 отдельно оговаривается использование операций для работы со словарями. Память измеряется в машинных словах, имеющих размер $\mathcal{O}(\log n)$ бит для входа длины n.

Приближённый алгоритм для задачи максимизации имеет *аддитивную погрешность* E (соотв., *мультипликативную погрешность* ε) если он находит решение со стоимостью не менее $OPT - E$ (соотв., $\frac{OPT}{1+\varepsilon}$), где OPT — это стоимость оптимального решения; здесь E и ε могут быть функциями от размера входа.

Вероятностные алгоритмы

В рассматриваемых задачах будет показываться неприменимость детерминированных алгоритмов. Вместо этого будут использоваться алгоритмы, совершающие случайный выбор в процессе работы. Такие алгоритмы называются *рандомизированными*, и существует два основных типа рандомизированных алгоритмов:

- *алгоритм типа Монте-Карло* возвращает корректный ответ с высокой вероятностью (больше чем $1 - 1/n$) и имеет детерминированные время работы и объём используемой памяти;
- *алгоритм типа Лас-Вегас* всегда возвращает корректный ответ, но его время работы и используемая память на входах размера n являются случайными величинами.

Мы будем использовать в первую очередь алгоритмы типа Монте-Карло. При этом для алгоритмов типа Лас-Вегас в рассматриваемых задачах будут доказываться линейные нижние оценки на необходимую память.

1.1.4 Хэш Карпа–Рабина

Хэш функции, известные как *хэш Карпа–Рабина* [39], используются в большинстве результатов по строковым алгоритмам в модели потока данных. В диссертации все алгоритмы также будут активно использовать хэши Карпа–Рабина. Далее мы опишем эту функцию и её свойства.

Пусть p — фиксированное простое число из отрезка $[n^\alpha, n^{1+\alpha}]$ для некоторого $\alpha > 1$, и r — фиксированное целое число выбранное случайно и равновероятно из множества $\{1, \ldots, p-1\}$. Для строки S над целочисленным алфавитом хэш определяется как значение в точке r полинома над полем $\mathbb{Z}_p$ с коэффициентами, равными символам строки:

$$\phi(S) = \Big(\sum_{i=1}^{n} S[i] \cdot r^i \Big) \bmod p \tag{1.1}$$

Использование хэшей позволяет заменить сравнение строк сравнением их хэшей (которое требует константного времени). При сравнении возможны ложноположительные результаты, соответствующие совпадениям (коллизиям) хэшей,

но невозможны ложноотрицательные. Вероятность совпадения хэшей для двух различных строк длины m не превосходит m/p. Вычисление хэшей производится очень быстро, как показывает следующая лемма. Для строки A *фреймом* назовем кортеж $(|A|, \phi(A), r^{|A|} \bmod p, r^{-|A|} \bmod p)$.

Лемма 1.1.1 ([11])**.** Если фреймы любых двух строк из A, B, AB известны, фрейм третьей строки может быть вычислен за время $\mathcal{O}(1)$.

Для демонстрации работы с хэшами Карпа–Рабина разберём один из трёх случаев леммы 1.1.1, остальные доказываются аналогично. Пусть известны фреймы для AB и B, вычислим фрейм для A:

- $|A| = |AB| - |B|$
- $r^{|A|} \bmod p = r^{|AB|} r^{-|B|} \bmod p$
- $r^{-|A|} \bmod p = r^{-|AB|} r^{|B|} \bmod p$
- $\phi(A) = \left(\sum_{i=1}^{|A|} S[i] \cdot r^i + \sum_{i=1}^{|B|} S[i] \cdot r^{|A|+i} - \sum_{i=1}^{|B|} S[i] \cdot r^{|A|+i}\right) \bmod p = (\phi(AB) - \phi(B) r^{|A|}) \bmod p$ □

Когда входной поток S зафиксирован, мы пишем $I(i)$ для обозначения фрейма подстроки $S[1..i-1]$. Из леммы 1.1.1 следует, в частности, что $I(i+1)$ может быть вычислен за $\mathcal{O}(1)$ из $I(i)$ и $S[i]$.

1.1.5 Словари

Словарь — это структура данных, в которой хранятся пары (ключ, значение), и которая поддерживает операции вставки, удаления и поиска по ключу. В диссертации мы будем использовать *хэш-таблицы* — реализации словаря, основанные на вычислении хэш-функции.

Пусть K — это множество возможных ключей, $M = \{1..m\}$ — это множество позиций в хэш-таблице. Используемая хэш-функция по ключу вычисляет позицию $f : K \to M$. *Коллизия* хэш функции — это пара ключей $k_1 \neq k_2$, таких что $f(k_1) = f(k_2)$. Хэш-функция, которая не имеет коллизий, называется *совершенной*.

Таким образом, если хэш-функция не совершенна, в некоторых позициях хэш-таблицы будет храниться больше одной пары (ключ, значение). В случае использования списков для хранения всех пар с одинаковым значением хэш-функции от ключа, время операции пропорционально размеру списка.

Нам интересны хэш-таблицы с лучшими гарантиями времени выполнения каждой операции.

Хэш-таблица, описанная в [23], с высокой вероятностью выполняет все запросы за $\mathcal{O}(1)$, когда количество запросов полиномиально от размера хэш-таблицы. Описанная хэш-таблица основана на системе из двух уровней. По значению

хэш-функции первого уровня определяется номер хэш-таблицы второго уровня. При этом на втором уровне используются совершенные хэш-функции.

Такие же гарантии (все запросы за $\mathcal{O}(1)$ с высокой вероятностью, когда количество запросов полиномиально от размера хэш-таблицы) имеет хэш-таблица, описанная в [6].

Кроме хэш-таблиц, мы будем использовать детерминированные словари Андерсона–Торапа [3], выполняющие каждую операцию за $\mathcal{O}\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ времени, где n — это количество пар (ключ, значение) в словаре перед операцией.

1.1.6 Принцип Яо

Для доказательства нижних оценок на вычислительную сложность задач будем использовать принцип Яо [59]. С его помощью можно получать оценки на ресурсы, необходимые рандомизированному алгоритму.

Теорема 1.1.1 (Принцип Яо)**.** Пусть $\mathcal{X}$ — множество входов некоторой задачи, $\mathcal{A}$ — множество всех детерминированных алгоритмов, решающих её, и $c(a, x) \geq 0$ — стоимость запуска алгоритма $a \in \mathcal{A}$ на $x \in \mathcal{X}$.

Пусть p — распределение вероятностей над $\mathcal{A}$, и пусть A обозначает алгоритм, выбранный случайно в соответствии с распределением p. Пусть q — распределение вероятностей над $\mathcal{X}$, а X обозначает вход, выбранный случайно в соответствии с распределением q. Тогда

$$\max_{x \in \mathcal{X}} \mathbf{E}[c(A, x)] \geq \min_{a \in \mathcal{A}} \mathbf{E}[c(a, X)] \tag{1.2}$$

Для использования принципа определяется функция стоимости $c(a, x)$ запуска детерминированного алгоритма a на тесте x. При этом рандомизированный алгоритм рассматривается как алгоритм детерминированный, случайно выбранный из некоторого множества алгоритмов, решающих данную задачу, согласно заданному распределению. Это соответствует тому, что все случайные выборы, которые должны быть выполнены в процессе работы алгоритма, сделаны заранее. Таким образом, неравенство (1.2) в принципе Яо говорит о том, что рандомизированный алгоритм на худшем для себя тесте не может быть лучше, чем лучший детерминированный алгоритм, выбранный для фиксированного множества тестов.

Таким образом использование алгоритма Яо заключается в выборе функции стоимости, для алгоритмов типа Лас-Вегас будем использовать потребляемую память, а для алгоритмов типа Монте-Карло стоимость будет индикаторной функцией корректности алгоритма. После этого строится множество тестов $\mathcal{X}$, сложное для любого детерминированного алгоритма, и таким образом получается нижняя оценка для рандомизированных алгоритмов.

1.1.7 Модель потока данных

Мы рассматриваем модель потока данных: входная строка $S[1..n]$ (называемая *потоком*) читается слева направо, по одному символу, и не может быть сохранена в памяти, потому что доступная алгоритму память является сублинейной функцией от n. Мы называем алгоритмы, работающие в приведенной модели, *потоковыми*. Потоковый алгоритм обязательно является онлайн-алгоритмом, т.е. его работу можно представить циклом

```
1: for i = 1 to n do
2:     прочитать S[i]; обработать S[1..i]; выдать ответ для S[1..i]
```

Итерацией потокового алгоритма назовём одну итерацию этого внешнего цикла.

Чаще всего потоковые алгоритмы являются рандомизированными (обычно, типа Монте-Карло) и приближенными; для других типов алгоритмов удается доказать линейные нижние оценки на используемую память, что исключает их из модели потока данных. Все приводимые в диссертации алгоритмы являются приближенными рандомизированными алгоритмами типа Монте-Карло.

Самыми важными характеристиками вычислительной сложности потоковых алгоритмов являются объём используемой памяти и время обновления (максимальное время, требуемое для выполнения одной итерации); общее время работы алгоритма менее важно. Отличительной чертой приближенных алгоритмов является то, что используемые алгоритмом ресурсы (в особенности память) обычно зависит от погрешности; эта зависимость формирует «трейд-офф» между точностью алгоритма и потребляемыми им ресурсами.

Степень разработанности

В одной из первых работ в потоковой модели [52] Мунро и Патерсон рассмотрели задачи сортировки потока и вычисления k-ой порядковой статистики, а также медианы как важного частного случая. Среди прочих результатов в статье показывается, что для вычисления медианы детерминированному алгоритму требуется линейная память, при этом представлен рандомизированный алгоритм, вычисляющий медиану с высокой вероятностью и использующий память объёмом $\mathcal{O}(\sqrt{n})$. Вообще, необходимость рандомизированных алгоритмов является характерной чертой модели потока данных.

В ключевой в области работе Алона, Матиаса и Сегеди [1] рассматривалась задача вычисления k-х частотных моментов потока, где k-й частотный момент равен сумме k-х степеней частот встретившихся в потоке символов. В частности, 0-й частотный момент — это количество различных символов, 1-й — это длина потока, 2-й — это квадрат евклидовой нормы вектора частот. В статье показано, что для приближения k-х частотных моментов при $k \geq 6$ требуется линейная память. А вот 0-й, 1-й и 2-й моменты могут быть вычислены с логарифмической.

При этом 0-й и 2-й моменты могут быть вычислены только приближённо и только с использованием рандомизированных алгоритмов. Рандомизированному алгоритму, вычисляющему точное значение, или детерминированному алгоритму, решающему задачу приближённо, уже понадобится линейная память.

Для задачи приближения количества различных элементов в потоке Бар-Йосеф и др. [8] представили позже более эффективный рандомизированный алгоритм.

Другие интересные результаты, полученные в модели потока данных, относятся к задачам на графах. В этом случае элементы потока рассматриваются как новые или удаляемые рёбра графа. Например, Кейн и др. [37] рассматривали задачу приближённого подсчёта числа вхождений фиксированного подграфа константного размера в большой граф, заданный потоком. Обзор задач на графах в модели потока данных представлен в работе Макгрегора [49].

Изучение алгоритмов на строках в модели потока данных началось со статьи Пората и Пората [53], в которой рассматривалась задача поиска по образцу в модели потока данных. В ней был представлен алгоритм, который предварительно обрабатывает образец, а затем, используя $\mathcal{O}(\log m)$ памяти (где m это длина образца), обрабатывает каждый символ текста за $\mathcal{O}(\log m)$, и с высокой вероятностью находит все вхождения образца в текст. И уже через два года Галил и Бреслауэр [11] представили алгоритм реального времени, то есть обрабатывающий каждый символ потока за $\mathcal{O}(1)$, который решает задачу поиска по образцу в полностью потоковой постановке задачи (во входном потоке сначала записан образец и после него через разделитель записан текст) с использованием $\mathcal{O}(\log m)$ памяти.

Более сложные задачи поиска также рассматриваются в модели потока данных. Один из примеров — это задача поиска с k ошибками. В этой задаче позиция считается вхождением образца, если расстояние Хэмминга между образцом и соответствующей подстрокой текста не превосходит k. Первый результат по этой задаче был представлен в статье [53]. После серии серии улучшений Клиффорд, Кочумака и Порат [16] представили алгоритм для полностью потоковой версии задачи, использующий память объёмом $\mathcal{O}(k \cdot \mathrm{poly}(\log n))$ и обрабатывающий один символ потока за $\mathcal{O}(\sqrt{k} \cdot \mathrm{poly}(\log n))$. Сходными задачами рассмотренными в потоковой модели является поиск многих образцов [14], приближенный поиск образца [15] и параметризованный поиск образца [36].

Кроме того, в потоковой модели рассматривались задачи поиска регулярных структур в строках. Так, отправной точкой для работы над данной диссертацией послужила статья Беренбринк и др. [9], авторы которой представили потоковые алгоритмы для приближенного поиска палиндромов в строке.

1.2 Обзор диссертации

1.2.1 Цели и задачи диссертации

Целью диссертации является анализ вычислительной сложности поиска регулярных структур в строках в модели потока данных. Для достижения этой цели в диссертации рассмотрены следующие комбинаторные задачи:

- Задача LPS (длиннейшая палиндромная подстрока, longest palindromic substring): дан поток S, найти длину и начальную позицию самой длинной подстроки в S, являющейся палиндромом.
- Задача LRS (длиннейший повтор, longest repeating substring): дан поток S, найти длину наибольшей строки w, которая имеет больше одного вхождения в S, и начальные позиции её двух вхождений.
- Задача LRRS (длиннейший обратный повтор, longest repeating reversed substring): дан поток S, найти длину наибольшей строки w, такой что w и $\overleftarrow{w}$ имеют вхождения в S, и начальные позиции вхождений w и $\overleftarrow{w}$.
- Задача Runs (максимальные периодические подстроки): дан поток S, найти все максимальные периодические подстроки в S.

1.2.2 Основные методы исследования

В работе используются методы теории алгоритмов, теории сложности, разнообразные структуры данных.

Исследование вычислительных возможностей модели состоит из двух частей: поиск нижних оценок, то есть нахождение ограничений при которых задача не имеет решения, и поиск верхних оценок, то есть построение эффективных алгоритмов. Задача может считаться полностью исследованной, когда верхняя и нижняя оценки совпадают, то есть для каждого возможного набора ограничений или приведён удовлетворяющий им алгоритм, или доказано его отсутствие.

Доказательство нижних границ в работе основано на принципе Яо (теорема 1.1.1). Для построения эффективных алгоритмов в работе используются хэши Карпа–Рабина, а также различные структуры данных, в том числе словари, суффиксные деревья, бинарные индексные структуры [2] и динамическое взвешенное дерево предков [43]. При доказательстве корректности и вычислительной сложности алгоритмов активно используется комбинаторика слов.

1.2.3 Структура диссертации и организация текста

Диссертация состоит, помимо введения, из трех глав, заключения и списка литературы. В главе 2 рассматривается задача LPS, в главе 3 — родственные

задачи LRS и LRRS, а в главе 4 — задача Runs. В заключении подводятся итоги работы и описываются направления дальнейшей работы по проблематике диссертации.

1.2.4 Апробация и публикации

Все представленные результаты опубликованы в статьях автора [32, 33, 50, 51] (в соавторстве с Арсением Михайловичем Шуром, две из них в соавторстве с Павлом Гавриховским и Пшемыславом Узнаньским) в журнале Алгоритмика (Algorithmica, Springer) и в сборниках следующих международных конференций: 27-й ежегодный симпозиум по комбинаторному поиску образцов (CPM 2016, Тель-Авив, Израиль), 30-й ежегодный симпозиум по комбинаторному поиску образцов (CPM 2019, Пиза, Италия), 26-й международный симпозиум по обработке строк и извлечению информации (SPIRE 2019, Сеговия, Испания). Все названные сборники вышли в сериях Leibniz International Proceedings in Informatics (LIPIcs), изд-во Schloss Dagstuhl – Leibniz-Zentrum für Informatik, и Lecture Notes in Computer Science (LNCS), изд-во Springer; они удовлетворяют достаточному условию ВАК для опубликования результатов диссертации. Кроме указанных конференций, результаты также докладывались на конференции «Проблемы теоретической информатики 2019» (ВШЭ совместно с ПОМИ РАН, Москва), конкурсе студенческих работ по теоретической информатике и дискретной математике им. Алана Тьюринга (СПбАУ РАН, Санкт-Петербург, 2016, 1 место) и на семинаре «Алгебраические системы» (УрФУ, рук. Шеврин Л. Н. 2019–2020). В следующем разделе после краткого изложения каждого результата указывается, в какой именно статье он опубликован.

Вклад автора диссертации

Из работ [32, 33] в диссертацию включены алгоритмы A, M, M’ и теоремы об их корректности и вычислительной сложности, принадлежащие автору. А.М. Шуру в этих статьях принадлежит алгоритм E, не вошедший в диссертацию, а П. Гавриховскому и П. Узнаньскому — доказательство нижних границ. В статьях [50, 51] основные результаты принадлежат автору (А.М. Шуру принадлежат постановка задач и оптимизация некоторых доказательств).

1.2.5 Основные результаты диссертации

Палиндромы в потоках

Палиндромы (см. определение в подразделе 1.1.1) — одна из базовых регулярных структур в тексте, и они хорошо изучены в классической модели вычислений. Так линейный алгоритм, вычисляющий все палиндромы в строке, был представлен в [48], а в [26] показано, как преобразовать его в алгоритм реального

времени. Другие результаты, ставшие классическими, были доказаны в [4, 28, 40]. Активное изучение задач, связанных с палиндромами, не прекращается; например, в статье [54] была приведена структура данных, являющаяся эффективным представлением множества палиндромов в строке. Другие свежие результаты включают [10, 13, 46]. Одной из причин активного изучения палиндромов является практическая значимость в вычислительной биологии «инволютивных» палиндромов, см., например, [38]. В модели потока данных задачи поиска палиндромов были впервые рассмотрены в [9].

В главе 2 мы рассматриваем задачу LPS о наибольшем палиндроме в модели потока данных (результаты опубликованы в [32, 33]). В работах [32] и [33] соавторами доказано, что точных потоковых алгоритмов и приближенных рандомизированных потоковых алгоритмов типа Лас-Вегас для этой задачи не существует, а для приближенных рандомизированных алгоритмов типа Монте-Карло доказаны нижние оценки на используемую память. Эти оценки равны $\Omega(M \log \min\{|\Sigma|, M\})$ бит памяти, где $M = n/E$ для приближения ответа с аддитивной погрешностью E и $M = \log n / \log(1 + \varepsilon)$ для приближения ответа с мультипликативной погрешностью $(1 + \varepsilon)$.

В диссертации построены три алгоритма реального времени для задачи LPS. Это приближенные алгоритмы типа Монте-Карло для аддитивной погрешности, «маленькой» и «большой» мультипликативной погрешности, соответственно. Каждый из алгоритмов использует $\mathcal{O}(M)$ слов памяти, где M определено выше. Таким образом, алгоритмы достигают нижней оценки с точностью до логарифмического множителя (а при большинстве значений параметров — с точностью до константного множителя).

Повторы и обратные повторы в потоках

В главе 3 речь идёт о тесно связанных задачах LRS и LRRS. Поиск наибольшей повторяющейся подстроки в классической модели вычислений является известным приложением суффиксных деревьев, упомянутым в пионерской статье о них [57]; задача LRRS в классической модели решается аналогично.

В главе 3 мы рассматриваем задачи LRS и LRRS в модели потока данных; результаты опубликованы в статье [50]. Вначале мы показываем, что точных алгоритмов и приближенных рандомизированных потоковых алгоритмов типа Лас-Вегас для этих задач не существует, а для приближенных рандомизированных алгоритмов типа Монте-Карло для обеих задач справедливы нижние оценки в $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ бит памяти, для приближения ответа с аддитивной погрешностью E.

Для обеих задач в диссертации представлены линейные алгоритмы типа Монте-Карло, использующие $\mathcal{O}(E + \frac{n}{E})$ слов памяти, где E — это аддитивная ошибка приближения. С теми же ограничениями на память представлены алгоритмы, близкие к реальному времени, тратящие $\mathcal{O}(\log n)$ времени на один символ и $\mathcal{O}(n + \frac{n}{E} \log n)$ времени на обработку всего потока. Используемая память точно

совпадает с нижними оценками при условиях $E = \mathcal{O}(\sqrt{n})$ и $|\Sigma| = \Omega(n^{0.01})$. В представленных алгоритмах используются суффиксные деревья и некоторые технически сложные структуры данных.

Максимальные периодические подстроки в потоках

В главе 4 мы рассматриваем задачу о максимальных периодических подстроках (см. определение в подразделе 1.1.1). Максимальные периодические подстроки хорошо изучены и в алгоритмической и в комбинаторной постановках. В частности, гипотеза Колпакова–Кучерова [41] о том, что их количество не превосходит длину строки, доказана в [7]. Известны эффективные алгоритмы для нахождения всех максимальных периодических подстрок как в случае полиномиального алфавита [41], так и в случае общего алфавита [20, 30, 44, 45]. Задача поиска всех подстрок, обладающих определённой регулярной структурой, в некотором смысле нехарактерна для потоковой модели.

В диссертации мы показываем что потоковый алгоритм, в том числе рандомизированный алгоритм типа Монте-Карло, не может точно найти (и даже посчитать) все квадраты в строке, а следовательно, и все максимальные периодические подстроки. Таким образом, для задачи Runs в модели потока данных необходимо вначале сформулировать подходящую приближенную задачу. Эта задача (approxRuns, приближение максимальных периодических подстрок) нами сформулирована следующим образом: дан поток S и параметр допустимой погрешности ε, для каждой максимальной периодической подстроки в S с экспонентой $\beta \geq 2 + \varepsilon$, выдать одну подстроку с тем же периодом и экспонентой не меньше $\beta - \varepsilon$; для максимальных периодических подстрок с экспонентой меньше $2 + \varepsilon$ — выдать одну или ноль подстрок с тем же периодом и экспонентой не меньше 2.

Основной результат главы — это рандомизированный алгоритм типа Монте-Карло, решающий задачу approxRuns. Алгоритм использует $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$ слов памяти и обрабатывает каждый символ потока за $\mathcal{O}(\log n)$ операций, включая операции со словарём. Представленный алгоритм использует ряд комбинаторных свойств строк для повышения эффективности работы.

Положения, выносимые на защиту

- Алгоритмы типа Монте-Карло приближенного решения задачи LPS и доказательства их корректности и вычислительной сложности (алгоритмы A, M и M'; теоремы 2.3.1, 2.3.2 и 2.3.3).

- Теорема о нижних оценках для точных и приближенных алгоритмов типа Лас-Вегас для задачи LRS (теорема 3.6.1) и теорема о нижней оценки для алгоритма типа Монте-Карло для приближенного решения задачи LRS с аддитивной погрешностью (теорема 3.6.2).

- Алгоритмы типа Монте-Карло приближенного решения задач LRS и LRRS и доказательства их корректности и вычислительной сложности (алгоритмы ARR, FastARR и FastAR; теоремы 3.4.1, 3.4.2 и 3.5.2).
- Теорема о нижней оценке для точного потокового алгоритма, решающего задачу midSquare (теорема 4.2.1).
- Алгоритм типа Монте-Карло решения приближенной задачи approxRuns и доказательство его корректности и вычислительной сложности (алгоритм R; теорема 4.1.1).

Научная новизна

Все представленные в диссертации результаты являются новыми.

Степень достоверности

Все представленные в диссертации результаты снабжены строгими математическими доказательствами, обеспечивающими их достоверность.

Теоретическая и практическая значимость

Диссертация является теоретической работой; ее результаты могут применяться для дальнейших исследований в области строковых алгоритмов, для разработки алгоритмов обработки реальных потоков данных, а также для чтения специальных курсов по алгоритмам и структурам данных.

Глава 2

Палиндромы в потоках

2.1 Введение

Мы рассматриваем задачу вычисления наибольшего палиндрома в потоковой модели, где палиндром — это фрагмент, который читается одинаково в обоих направлениях (см. определение в разделе 1.1.1). Палиндромы — одна из базовых регулярных структур в тексте и она хорошо изучена в классической модели, см. [4, 28, 40, 48]. Определение палиндромов, с небольшим изменением, имеет большое значение в вычислительной биологии, где рассматриваются строки над алфавитом $\{A, T, C, G\}$, и строка является палиндромом, если она обратна к своему дополнению (дополнение взаимно заменяет символы A и T, а также C и G); см. [31] и ссылки внутри для обсуждения алгоритмических аспектов. Все наши результаты тривиально обобщаются на биологические палиндромы.

Напомним, что задача LPS формулируется так: *в заданной строке S найти наибольшую длину палиндрома $L(S)$ и стартовую позицию палиндрома такой длины.* Впервые задача LPS в потоковой модели рассмотрена в [9], где авторы предложили рассматривать приближённые решения с аддитивной и мультипликативной погрешностью. При этом погрешность приближения (см. определения в разделе 1.1.3) было предложено рассматривать как параметр задачи, от которого зависят используемые ресурсы (память и время работы). В [9] представлены алгоритмы, решающие задачу LPS

(i) за $\mathcal{O}\left(\frac{n\sqrt{n}}{E}\right)$ времени и $\mathcal{O}\left(\frac{n}{E}\right)$ памяти с аддитивной ошибкой $E \in [1, \sqrt{n}]$;

(ii) за $\mathcal{O}\left(\frac{n \log n}{\varepsilon \log(1+\varepsilon)}\right)$ времени и $\mathcal{O}\left(\frac{\log n}{\varepsilon \log(1+\varepsilon)}\right)$ памяти с мультипликативной ошибкой $(1+\varepsilon)$, где $\varepsilon < 1$;

(iii) за $\mathcal{O}(n)$ времени и $\mathcal{O}(\sqrt{n})$ памяти точно, *в случае если наибольший палиндром короче чем $\sqrt{n}$.*

Все представленные алгоритмы — это алгоритмы типа Монте-Карло. В статье также показано, что любой алгоритм типа Лас-Вегас имеющий аддитивную погрешность E должен использовать $\Omega\left(\frac{n}{E} \log |\Sigma|\right)$ бит памяти. Эти результаты

оставляют множество открытых вопросов как про более эффективные алгоритмы (например, только алгоритм (iii) и особый случай алгоритма (i) линейные) так и про более точные нижние оценки на необходимую память, в частности, для алгоритмов типа Монте-Карло. В данном разделе мы отвечаем на все эти вопросы, по сути определяя временную сложность и необходимую память для задачи LPS.

Сначала мы приведём формулировки теорем, задающих нижние оценки на вычислительную сложность задачи LPS в модели потока данных. Эти теоремы были доказаны в работах [32, 33] соавторами (Гавриховским и Узнаньским). Сначала показывается, что алгоритмы типа Лас-Вегас не могут достичь сублинейной памяти; а следовательно в потоковой модели задача LPS может быть решена только с высокой вероятностью. Далее доказывается нижняя оценка в $\Omega(M \log \min\{|\Sigma|, M\})$ бит памяти для алгоритмов типа Монте-Карло; здесь $M = n/E$ для приближенного решения с аддитивной погрешностью $E \in [1, n]$ и $M = \frac{\log n}{\log(1+\varepsilon)}$ для приближенного решения с мультипликативной погрешностью $(1 + \varepsilon)$, где $\varepsilon > n^{-0.99}$.

После этого, мы представим три линейных алгоритма типа Монте-Карло, более того, являющихся алгоритмами реального времени, у которых используемая память совпадает с нижними оценками с точностью до логарифмического множителя (а для широких диапазонов задействованных параметров совпадает точно).

- Алгоритм A решает LPS с аддитивной погрешностью $E \in [1, n]$ использует $\mathcal{O}(n/E)$ машинных слов памяти. По сравнению с (i), он использует столько же памяти, но работает быстрее в случае $E = o(\sqrt{n})$, и убирает ограничения на E; другим преимуществом является отсутствие зависимости времени работы от допустимой погрешности. Используемая память точно совпадает с нижней оценкой при разумных допущениях $|\Sigma| > n^{0.01}, E < n^{0.99}$.

- Алгоритм M решает LPS с мультипликативной погрешностью $\varepsilon \in (0, 1]$, используя $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$ машинных слов памяти. По сравнению с (ii), используемая память уменьшается не менее чем в ε^{-1} раз (так как $\log(1 + \varepsilon)$ эквивалентен ε при $\varepsilon < 1$), ограничение по времени уменьшено на множитель $\varepsilon^{-2} \cdot \log n$. Используемая память совпадает с нижней оценкой с точностью до логарифмического множителя в худшем случае (если ε это константа и $|\Sigma| = \mathcal{O}(\log n)$); совпадение точное если $\varepsilon < n^{-0.01}, |\Sigma| > n^{0.01}$.

- Алгоритм M' решает LPS с мультипликативной погрешностью $\varepsilon \in (1, n]$ использует $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ слов памяти. Он не имеет аналогов в [9] и используемая память совпадает с нижней оценкой с точностью до логарифмического множителя.

Глава организована следующим образом: мы приводим теоремы о нижних оценках без доказательства в разделе 2.2, а потоковые алгоритмы A, M, и M' в

разделе 2.3. Через $S[1..n]$ мы обозначаем обрабатываемую строку (поток) длины n над целочисленным полиномиальным алфавитом $\Sigma = \{1, \ldots, \sigma\}$. Символ log используется для двоичного логарифма.

2.2 Нижние оценки

Приведём формулировки принадлежащих соавторам работ [32, 33] теорем о нижних оценках на необходимую память в задаче LPS в потоковой модели, где длина n и алфавит Σ входного потока заданы. Обозначим эту задачу $\mathsf{LPS}_\Sigma[n]$.

Теорема 2.2.1 (Приближение алгоритмом типа Лас-Вегас)**.** Пусть A — алгоритм типа Лас-Вегас, решающий $\mathsf{LPS}_\Sigma[n]$ с аддитивной погрешностью $E \le 0.99n$ или мультипликативной погрешностью $(1+\varepsilon) \le 100$, и использующий $s(n)$ бит памяти. Тогда $\mathbf{E}[s(n)] = \Omega(n \log |\Sigma|)$.

Теорема 2.2.2 (Аддитивное приближение алгоритмом типа Монте-Карло)**.** Пусть A — потоковый алгоритм типа Монте-Карло решающий $\mathsf{LPS}_\Sigma[n]$ с аддитивной погрешностью E с вероятностью $1 - \frac{1}{n}$. Если $E \le 0.99n$, то A использует $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ бит памяти.

Теорема 2.2.3 (Мультипликативное приближение алгоритмом типа Монте-Карло)**.** Пусть A — потоковый алгоритм типа Монте-Карло, решающий $\mathsf{LPS}_\Sigma[n]$ с мультипликативной погрешностью $(1+\varepsilon)$ с вероятностью $1 - \frac{1}{n}$. Если $n^{-0.98} \le \varepsilon \le n^{0.49}$, то A использует $\Omega(\frac{\log n}{\log(1+\varepsilon)} \log \min\{|\Sigma|, \frac{\log n}{\log(1+\varepsilon)}\})$ бит памяти.

2.3 Алгоритмы реального времени

В этом разделе мы приводим потоковые алгоритмы реального времени типа Монте-Карло, использующие память, совпадающую с нижними оценками из раздела 2.2 с точностью до множителя, ограниченного $\log n$. Алгоритмы используют хэш Карпа–Рабина (см. раздел 1.1.4). Приведем необходимые детали использования хэша в задаче LPS.

Пусть p — фиксированное простое из отрезка $[n^{3+\alpha}, n^{4+\alpha}]$ для некоторого $\alpha > 0$, и r — фиксированное целое число, выбранное случайно и равновероятно из $\{1, \ldots, p-1\}$. Для строки S определим, наряду с «прямым» хэшем (1.1), «обратный» хэш:

$$\phi^F(S) = \phi(S) = \Big(\sum_{i=1}^{n} S[i] \cdot r^i\Big) \bmod p\,, \quad \phi^R(S) = \Big(\sum_{i=1}^{n} S[i] \cdot r^{n-i+1}\Big) \bmod p\,.$$

Очевидно, прямой хэш строки совпадает с обратным хэшем от обратной строки. Следовательно, если u это палиндром, то $\phi^F(u) = \phi^R(u)$. Обратное также верно с точностью до (маловероятной) коллизии хэшей, потому что для двух строк

$u \neq v$ длины m вероятность того, что $\phi^F(u) = \phi^F(v)$, не больше m/p. Это свойство позволяет определять палиндромы с высокой вероятностью сравнением хэшей. (Этот подход несколько проще, чем использованный предшественниками в [9]; в частности, нам не нужны «пары хэшей», использованные там.)

Заметим, что алгоритм реального времени совершает $\mathcal{O}(n)$ сравнений хэшей для строк длины, не превосходящей n; значит, общая вероятность коллизии есть $\mathcal{O}(n^{-1-\alpha})$ по выбору p. Поскольку алгоритмы типа Монте-Карло допускают вероятность ошибки $\mathcal{O}(1/n)$, мы во всех дальнейших рассуждениях полагаем, что коллизий не произошло. Для входного потока S, мы определяем $F^F(i,j) = \phi^F(S[i..j])$ и $F^R(i,j) = \phi^R(S[i..j])$. Хэш любой подстроки может быть вычислен за константное время из хэшей двух префиксов по лемме 1.1.1.

Мы добавим в определение фрейма из раздела 1.1.4 обратный хэш. Все алгоритмы этой главы хранят фреймы только от префиксов входного потока; i-м фреймом потока S будем называть кортеж $I(i) = (i, F^F(1,i{-}1), F^R(1,i{-}1), r^{-(i-1)} \bmod p, r^i \bmod p)$. Утверждение ниже следует непосредственно из определений и леммы 1.1.1.

Предложение 2.3.1. *1) По заданному фрейму $I(i)$ и символу $S[i]$ фрейм $I(i{+}1)$ может быть вычислен за время $\mathcal{O}(1)$.*
2) По заданным фреймам $I(i)$ и $I(j{+}1)$ можно проверить за время $\mathcal{O}(1)$, является ли подстрока $S[i..j]$ палиндромом.

Все алгоритмы в этом разделе используют общую схему. Внешний цикл выполняет $n = |S|$ итераций; на i-й итерации обрабатывается символ $S[i]$. Каждый алгоритм вычисляет все фреймы, а хранит только некоторые из них, основываясь на объёме доступной памяти. После чтения $S[i]$, каждый алгоритм проверяет некоторые суффиксы строки $S[1..i]$ — являются ли они палиндромами длины большей, чем наибольший найденный ранее палиндром, и обновляет наибольший палиндром соответствующим образом. Какие именно суффиксы доступны для проверки, зависит от хранимых фреймов; каждая проверка занимает $\mathcal{O}(1)$ времени по предложению 2.3.1 (2). При помощи комбинаторных лемм доказывается, что на каждой итерации достаточно проверить лишь константное число суффиксов.

Допустим что $S[i..j]$ — наибольший палиндром в S. Очень вероятно, что i-й фрейм будет недоступен после чтения символа $S[j]$. Однако, мы покажем, что во всех случаях некоторый «достаточно близкий» $(i{+}k)$-й фрейм будет доступен после чтения $S[j{-}k]$. Тогда палиндром $S[i{+}k..j{-}k]$ будет найден на $(j{-}k)$-й итерации, обеспечивая достаточную точность приближения наибольшего палиндрома.

Техническая часть алгоритмов заключается в поддерживании списка сохранённых фреймов в таком виде, что (a) обеспечивается требуемое качество приближения; (b) обеспечивается доступ к фреймам, нужным на определённой итерации, за константное время; (c) список может быть обновлен на каждой итерации за константное время.

2.3.1 Аддитивная погрешность

Теорема 2.3.1. Существует потоковый алгоритм реального времени типа Монте-Карло, решающий задачу $\mathsf{LPS}(S)$ с аддитивной погрешностью $E = E(n)$ и использующий $\mathcal{O}(n/E)$ машинных слов памяти, где $n = |S|$.

Сначала мы представим простой (и медленный) алгоритм, который решает поставленную задачу, т.е. находит в S палиндром длины $\ell(S) \geq L(S) - E$, где $L(S)$ — длина наибольшего палиндрома в S. Затем мы преобразуем его в алгоритм реального времени. Мы храним фреймы $I(j)$ для некоторых позиций j в двусвязном списке SP в порядке убывания значений j. Наибольший палиндром, найденный на текущий момент, хранится как пара $answer = (pos, len)$, где pos — позиция вхождения палиндрома, а len — его длина. Пусть $t_E = \lfloor \frac{E}{2} \rfloor$.

В алгоритме ABasic мы добавляем $I(j)$ в список SP для всех j, делящихся на t_E. Это позволяет нам на i-й итерации проверять на палиндромность все подстроки вида $S[kt_E..i]$. Мы подразумеваем, что в начале i-й итерации фрейм $I(i)$ хранится в переменной I.

Алгоритм 2.1. Алгоритм ABasic, i-я итерация

1: **if** $i \bmod t_E = 0$ **then**
2: add I to the beginning of SP
3: read $S[i]$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
4: **for** all elements v of SP **do**
5: **if** $S[v.i..i]$ is a palindrome and $answer.len < i - v.i + 1$ **then**
6: $answer \leftarrow (v.i, i - v.i + 1)$

Предложение 2.3.2. *Алгоритм ABasic находит в S палиндром длины $\ell(S) \geq L(S) - E$, использует $\mathcal{O}(n/E)$ памяти и имеет время обновления $\mathcal{O}(n/E)$.*

Доказательство. Ограничения на время и на память исходят из размера списка SP, который ограничен $n/t_E = \mathcal{O}(n/E)$ элементами; по предложению 2.3.1, количество операций на итерации пропорционально размеру списка.

Рассмотрим наибольший палиндром $S[i..j]$ в S. Пусть $k = \lceil \frac{i}{t_E} \rceil t_E$. Тогда $i \leq k < i + t_E$. На k-й итерации $I(k)$ добавляется в SP; значит, на итерации $j - (k - i)$ будет найден палиндром $S[k..j-(k-i)]$. Его длина

$$j - (k - i) - k + 1 = j - i + 1 - 2(k - i) > L(S) - 2t_E \geq L(S) - E,$$

как и требуется. □

Ускорение алгоритма ABasic основывается на следующем наблюдении:

Лемма 2.3.1. За одну итерацию длина $answer.len$ увеличивается не более чем на $2 \cdot t_E$.

Доказательство. Пусть $S[j..i]$ — наибольший палиндром, найденный на итерации i. Если $i - j + 1 \leq 2t_E$, то утверждение очевидно выполняется. Иначе, рассмотрим подстроку $S[j+t_E..i-t_E]$ — это тоже палиндром, его длина $i-j+1-2t_E$, и он должен был быть найден ранее (на $(i-t_E)$-й итерации). Значит, и в этом случае утверждение выполняется. □

Из леммы 2.3.1 следует, что на каждой итерации только два доступных суффикса могут обновить ответ (см. рис. 2.1). Таким образом, получаем следующий алгоритм A.

Рис. 2.1: Поиск палиндрома, который может обновить ответ. Квадраты обозначают позиции j, такие что фрейм $I(j)$ сохранён; скобки показывают подстроки, которые могут быть проверены на палиндромность. По лемме 2.3.1, только подстроки-«кандидаты» могут быть палиндромами длины $> answer.len$.

Алгоритм 2.2. Алгоритм A, i-я итерация

```
1: if i mod t_E = 0 then
2:     add I to the beginning of SP
3:     if i = t_E then
4:         sp ← first(SP)
5: read S[i]; compute I(i+1) from I; I ← I(i+1)
6: sp ← previous(sp)                          ▷ если существует
7: while i − sp.i + 1 ≤ answer.len and (sp ≠ last(SP)) do
8:     sp ← next(sp)
9: for all existing v in {sp, next(sp)} do
10:    if S[v.i..i] is a palindrome and answer.len < i−v.i+1 then
11:        answer ← (v.i, i−v.i+1)
```

Из леммы 2.3.1 следует, что цикл в строках 9–11 алгоритма A вычисляет ту же последовательность значений переменной $answer$, что и цикл в строках 4–6 алгоритма ABasic. Следовательно, по предложению 2.3.2, он находит палиндром требуемой длины. Очевидно, объём памяти, используемый двумя алгоритмами, отличается на константу. Чтобы доказать, что итерация алгоритма A выполняется за $\mathcal{O}(1)$ времени, достаточно заметить, что цикл в строках 7–8 совершает не более двух итераций. Теорема 2.3.1 доказана.

2.3.2 Мультипликативная погрешность $\varepsilon \leq 1$

Теорема 2.3.2. Существует потоковый алгоритм реального времени типа Монте-Карло, решающий задачу $\mathsf{LPS}(S)$ с мультипликативной погрешностью $\varepsilon = \varepsilon(n) \in (0, 1]$ и использующий $\mathcal{O}\left(\frac{\log(n\varepsilon)}{\varepsilon}\right)$ машинных слов памяти, где $n = |S|$.

Как и в прошлом разделе, мы сначала представим более простой алгоритм 2.3 с нелинейным временем работы и затем улучшим его до алгоритма реального времени. Алгоритм должен найти палиндром длины $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$. Следующая очевидная лемма даёт более удобное выражение.

Лемма 2.3.2. При $\varepsilon \in (0, 1]$ из условия $\ell(S) \geq L(S)(1 - \frac{\varepsilon}{2})$ следует $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$.

Определим $q_\varepsilon = \left\lceil \log \frac{2}{\varepsilon} \right\rceil$. Основное отличие в построении алгоритма с мультипликативной погрешностью от алгоритма с аддитивной погрешностью заключается в том, что здесь *каждый* фрейм добавляется в список SP, но затем после некоторого числа итераций удаляется из списка. Количество итераций, которое фрейм $I(i)$ хранится в SP, определяется функцией времени жизни $\mathsf{ttl}(i)$, заданной ниже. Эта функция отвечает за корректность алгоритма и за объём используемой памяти.

Алгоритм 2.3. Алгоритм MBasic, i-я итерация

```
1: add I to the beginning of SP
2: for all v in SP do
3:     if v.i + ttl(v.i) = i then
4:         delete v from SP
5: read S[i]; compute I(i+1) from I; I ← I(i+1)
6: for all v in SP do
7:     if S[v.i..i] is a palindrome and answer.len < i−v.i+1 then
8:         answer ← (v.i, i−v.i+1)
```

Пусть $\beta(i)$ — это позиция самой правой единицы в двоичном представлении числа i (позиция 0 соответствует наименее значимому биту). Определим

$$\mathsf{ttl}(i) = 2^{q_\varepsilon + 2 + \beta(i)}\,. \tag{2.1}$$

Это определение проиллюстрировано на рис. 2.2. Далее покажем некоторые свойства списка SP.

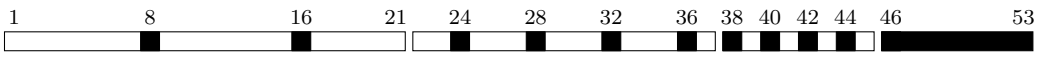

Рис. 2.2: Состояние списка SP после итерации $i = 53$ (подразумевается $q_\varepsilon = 1$). Чёрные квадраты обозначают числа j, такие что фреймы $I(j)$ сохранены в текущий момент. К примеру, из (2.1) следует что $\mathsf{ttl}(28) = 2^{1+2+2} = 32$, так что $I(28)$ будет сохранено в SP до итерации $28 + 32 = 60$.

Лемма 2.3.3. Для любых целых чисел $a \geq 1$ и $b \geq 0$ существует единственное целое число $j \in [a, a + 2^b)$, такое что $\mathsf{ttl}(j) \geq 2^{q_\varepsilon + 2 + b}$.

Доказательство. Из (2.1) получаем, что $\mathsf{ttl}(j) \geq 2^{q_\varepsilon+2+b}$ тогда и только тогда, когда $\beta(j) \geq b$, т.е. j делится на 2^b (по определению β). Среди любых 2^b последовательных целых чисел ровно одно обладает этим свойством. □

Рисунок 2.2 показывает разбиение отрезка $[1, i]$ на полуинтервалы, длины которых являются степенями двойки (за исключением самого левого полуинтервала). В общем случае, это разбиение состоит из $m - q_\varepsilon$ полуинтервалов, которые, справа налево, равны

$$(i - 2^{q_\varepsilon+2}, i], (i - 2^{q_\varepsilon+3}, i - 2^{q_\varepsilon+2}], \ldots, (i - 2^m, i - 2^{m-1}], (0, i - 2^m], \qquad (2.2)$$

где $m = \lceil \log i \rceil - 1$ (если $m \le q_\varepsilon$, то полуинтервал один). Из леммы 2.3.3 и (2.1) следует следующая лемма о распределении элементов SP.

Лемма 2.3.4. После каждой итерации, первый полуинтервал (соотв., последний полуинтервал; каждый из оставшихся полуинтервалов) в (2.2) содержит $2^{q_\varepsilon+2}$ (соотв., не более $2^{q_\varepsilon+1}$; ровно $2^{q_\varepsilon+1}$) позиций, для которых фреймы сохранены в списке SP.

Количество полуинтервалов в (2.2) есть $\mathcal{O}(\log(n\varepsilon))$, а значит из леммы 2.3.4 и определения q_ε получаем следующий результат.

Лемма 2.3.5. После каждой итерации размер списка SP есть $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$.

Предложение 2.3.3. *Алгоритм 2.3 находит палиндром длины $\ell(S) \ge \frac{L(S)}{1+\varepsilon}$, используя $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$ памяти, со временем обновления $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$.*

Доказательство. Оценки времени обновления и объёма памяти доминируются размером списка SP. А значит, требуемые оценки на сложность следуют из леммы 2.3.5. Теперь покажем корректность. Пусть $S[i..j]$ — палиндром длины $L(S)$, и пусть $d = \lfloor \log L(S) \rfloor$.

Если $d < q_\varepsilon + 2$, то палиндром $S[i..j]$ будет найден точно, так как фрейм $I(i)$ хранится в SP на jой итерации:

$$i + \mathsf{ttl}(i) \ge i + 2^{q_\varepsilon+2} \ge i + 2^{d+1} > i + L(S) > j\,.$$

Иначе, по лемме 2.3.3 существует единственное целое число $k \in [i, i + 2^{d-q_\varepsilon-1})$, такое что $\mathsf{ttl}(k) \ge 2^{d+1}$. А значит, палиндром $S[i+(k-i)..j-(k-i)]$ будет найден на итерации $j - (k - i)$, так как фрейм $I(k)$ хранится в SP на этой итерации:

$$k + \mathsf{ttl}(k) \ge i + \mathsf{ttl}(k) \ge i + 2^{d+1} > j \ge j - (k - i)\,.$$

Длина этого палиндрома удовлетворяет требованиям замечания:

$$\begin{aligned} j - (k - i) - (i + (k - i)) + 1 = L(S) - 2(k - i) &\ge L(S) - 2^{d-q_\varepsilon} \\ &\ge L(S) - L(S)/2^{q_\varepsilon} \ge L(S)(1 - \varepsilon/2)\,. \end{aligned}$$

Ссылка на лемму 2.3.2 завершает доказательство. □

Теперь мы ускорим алгоритм 2.3. Он имеет две медленные части: удаление из списка SP и проверка палиндромов. Леммы 2.3.6 и 2.3.7 показывают что, аналогично разделу 2.3.1, достаточно $\mathcal{O}(1)$ проверок на каждой итерации.

Лемма 2.3.6. Пусть на некоторой итерации список SP содержит последовательные элементы $I(d), I(c), I(b), I(a)$. Тогда $b - a \le d - b$.

Доказательство. Пусть j — номер рассматриваемой итерации. Заметим, что $a < b < c < d$. Рассмотрим полуинтервал в (2.2), содержащий a. Если $a \in (j - 2^{q_\varepsilon+2}, j]$, то $b - a = 1$ и $d - b = 2$, так что требуемое неравенство выполняется. Иначе, пусть $a \in (j - 2^{q_\varepsilon+2+x}, j - 2^{q_\varepsilon+2+x-1}]$. Тогда, по (2.1), $\beta(a) \ge x$; более того, любой фрейм $I(k)$, такой что $a < k \le j$ и $\beta(k) \ge x$, хранится в SP. Следовательно $b - a \le 2^x$. По лемме 2.3.4, каждый полуинтервал, за исключением самого правого, содержит не менее $2^{q_\varepsilon+1} \ge 4$ элементов. Значит, все числа b, c, d принадлежат или тому же полуинтервалу, что и a, или предыдущему полуинтервалу $(j - 2^{q_\varepsilon+2+x-1}, j - 2^{q_\varepsilon+2+x-2}]$. Опять по (2.1), получаем $\beta(b), \beta(c), \beta(d) \ge x - 1$. А значит, $c-b, d-c \ge 2^{x-1}$, из чего следует требуемое неравенство. □

Мы называем фрейм $I(a)$, хранимый в SP, *значимым на i-й итерации*, если $i - a + 1 > answer.len$ и $S[a..i]$ может быть палиндромом (то есть алгоритм 2.3 не хранит достаточно информации, чтобы предсказать, что для $v = I(a)$ условие в его строке 7 ложно).

Лемма 2.3.7. На каждой итерации SP содержит не более трёх значимых фреймов. Более того, если $I(d'), I(d)$ — последовательные элементы SP, такие что $i - d' < answer.len \le i - d$, где i — номер текущей итерации, то значимые фреймы являются последовательными элементами SP, начиная с $I(d)$.

Доказательство. Пусть d определено как в условии леммы. Если в SP за $I(d)$ следует не более двух фреймов, утверждение верно. Иначе, пусть следующие три фрейма — это $I(c), I(b)$ и $I(a)$ соответственно. Если $S[a..i]$ — палиндром, то $S[a+(b-a)..i-(b-a)]$ — тоже палиндром. На итерации $i-(b-a)$ фрейм $I(b)$ хранился в SP, значит, этот палиндром был найден. Следовательно, на iй итерации значение $answer.len$ не меньше длины этого палиндрома, то есть не меньше $i - a + 1 - 2(b - a)$. По лемме 2.3.6, $b - a \le d - b$, из чего следует, что $answer.len \ge i-a+1-(b-a)-(d-b) = i-d+1$. Это неравенство противоречит определению d; следовательно, $S[a..i]$ не является палиндромом. Аналогично показывается, что фреймы, следующие за $I(a)$ в SP, также не образуют палиндромов. А значит, только фреймы $I(d), I(c), I(b)$ являются значимыми. □

Лемма 2.3.7 говорит нам, что строки 7–8 алгоритма 2.3 достаточно выполнить для не более чем трёх последовательных элементов SP (как на рис. 2.1, но с не более чем тремя «кандидатами»). Теперь перейдём к удалениям. Функция $\mathsf{ttl}(x)$ обладает следующим полезным свойством.

Лемма 2.3.8. Функция $x \to x + \mathsf{ttl}(x)$ инъективна.

Доказательство. Заметим что $\beta(x + \mathsf{ttl}(x)) = \beta(x)$ по определению ttl. Следовательно из равенства $x + \mathsf{ttl}(x) = y + \mathsf{ttl}(y)$ следует, что $\beta(x) = \beta(y)$. Тогда, согласно (2.1), $\mathsf{ttl}(x) = \mathsf{ttl}(y)$, и, наконец, $x = y$. □

Из леммы 2.3.8 следует, что на каждой итерации из SP удаляется не более одного элемента. Нам понадобится дополнительная структура данных для выполнения этого удаления за $\mathcal{O}(1)$ времени. За $BS(x)$ мы будем обозначать связный список максимальных отрезков из единиц в двоичном представлении числа x. К примеру, двоичное представление числа $x = 12345$ и $BS(x)$ выглядят следующим образом:

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	1	0	0	1

$$BS(12345) = \{[0, 0], [3, 5], [10, 10], [12, 13]\}$$

Очевидно, список $BS(x)$ занимает $\mathcal{O}(\log x)$ памяти.

Лемма 2.3.9. Зная $BS(x)$, можно вычислить $\beta(x)$ и $BS(x + 1)$ за время $\mathcal{O}(1)$.

Доказательство. Пусть $[a, b]$ — это первый отрезок в $BS(x)$. Тогда $a = \beta(x)$ по определению $\beta(x)$. Построим $BS(x+1)$. Если $a > 1$, то $BS(x+1) = [0, 0] \cup BS(x)$. Если $a = 1$, то $BS(x + 1) = [0, b] \cup (BS(x) \backslash [1, b])$. Теперь пусть $a = 0$. Если $BS(x) = \{[0, b]\}$ то $BS(x + 1) = \{[b+1, b+1]\}$. Иначе пусть второй отрезок в $BS(x)$ это $[c, d]$. Если $c > b + 2$, то $BS(x + 1) = [b+1, b+1] \cup (BS(x) \backslash [0, b])$. Наконец, если $c = b + 2$, то $BS(x + 1) = [b+1, d] \cup (BS(x) \backslash \{[0, b], [c, d]\})$. □

Таким образом, если мы поддерживаем список BS, равный $BS(i)$ в конце i-й итерации, то у нас есть значение $\beta(i)$. Если фрейм $I(a)$ должен быть удалён из SP на этой итерации, то $i = a + \mathsf{ttl}(a)$, и следовательно $\beta(a) = \beta(i)$ (см. лемму 2.3.8). Следующая лемма тривиальна.

Лемма 2.3.10. Если $a < b$ и $\mathsf{ttl}(a) = \mathsf{ttl}(b)$, то $I(a)$ удаляется из SP раньше, чем $I(b)$.

По лемме 2.3.10, информация о позициях с одинаковым ttl (другими словами, с одинаковым значением β) добавляется и удаляется из SP в одном и том же порядке. А значит, можно поддерживать очередь $QU(x)$ указателей на все элементы SP, соответствующие позициям j, для которых $\beta(j) = x$. Такие очереди для каждого $x \in \{0, \ldots, \lfloor \log n \rfloor\}$ образуют последний ингредиент алгоритма M реального времени, представленного далее.

Доказательство теоремы 2.3.2. После каждой итерации алгоритм M имеет тот же список SP (см. рис. 2.2), что и алгоритм 2.3, потому что эти алгоритмы добавляют и удаляют одни и те же элементы. По лемме 2.3.7, алгоритм M возвращает тот же ответ, что и алгоритм 2.3. Следовательно, по предложению 2.3.3,

Алгоритм 2.4. Алгоритм M, i-я итерация

1: add I to the beginning of SP
2: **if** $i = 1$ **then**
3: $\quad sp \leftarrow first(SP)$
4: compute $BS[i]$ from BS; $BS \leftarrow BS[i]$; compute $\beta(i)$ from BS
5: **if** $QU(\beta(i))$ is not empty **then**
6: $\quad v \leftarrow$ element of SP pointed by $first(QU(\beta(i)))$
7: $\quad$ **if** $v = sp$ **then**
8: $\quad\quad sp \leftarrow next(sp)$
9: $\quad$ delete v; delete $first(QU(\beta(i)))$
10: add pointer to $first(SP)$ to $QU(\beta(i))$
11: read $S[i]$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
12: $sp \leftarrow previous(sp)$ ▷ если существует
13: **while** $i - sp.i + 1 \le answer.len$ and $sp \neq last(SP)$ **do**
14: $\quad sp \leftarrow next(sp)$
15: **for** all existing v in $\{sp, next(sp), next(next(sp))\}$ **do**
16: $\quad$ **if** $S[v.i..i]$ is a palindrome and $answer.len < i{-}v.i{+}1$ **then**
17: $\quad\quad answer \leftarrow (v.i, i{-}v.i{+}1)$

алгоритм M находит палиндром требуемой длины. Кроме этого, алгоритм M поддерживает список BS размера $\mathcal{O}(\log n)$ и массив QU содержащий $\mathcal{O}(\log n)$ очередей, суммарный размер которых равен размеру SP. Следовательно, он использует суммарно $\mathcal{O}(\frac{\log(n\varepsilon)}{\varepsilon})$ памяти, согласно лемме 2.3.5. Цикл в строках 13–14 совершает не более трёх итераций. Действительно, пусть значение sp после предыдущей итерации равно z. Тогда цикл стартует с $sp = previous(z)$ (или с $sp = z$, если z — первый элемент в SP), и заканчивается обработкой $sp = next(next(z))$. По лемме 2.3.9, и $BS(i)$, и $\beta(i)$ могут быть вычислены за $\mathcal{O}(1)$ времени. Следовательно, каждая итерация выполняется за $\mathcal{O}(1)$ времени. □

Замечание. Так как для $n^{-0.99} \le \varepsilon \le 1$ классы $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ и $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$ совпадают, алгоритм M использует память, отличающуюся на множитель, не превосходящий $\log n$. Более того, для произвольной медленно растущей функции φ алгоритм M использует $o(n)$ памяти при $\varepsilon = \frac{\varphi(n)}{n}$.

2.3.3 Мультипликативная погрешность $\varepsilon > 1$

Теорема 2.3.3. Существует потоковый алгоритм реального времени типа Монте-Карло, решающий задачу $\mathsf{LPS}(S)$ с мультипликативной погрешностью $\varepsilon = \varepsilon(n) \in (1, n]$ и использующий $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ машинных слов памяти, где $n = |S|$.

Наша цель заключается в преобразовании алгоритма M в алгоритм реального времени M', который решает $\mathsf{LPS}(S)$ с мультипликативной погрешностью $\varepsilon > 1$, используя $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ памяти. Основная идея преобразования заключа-

ется в замене всех двоичных представлений представлениями с основанием, пропорциональным $1+\varepsilon$, и таким образом в сжатии размера списков SP и BS.

Допустим без потери общности, что $\varepsilon \geq 7$, так как иначе мы можем положить $\varepsilon = 1$ и применить алгоритм M. Зафиксируем $k \leq \frac{1}{2}(1+\varepsilon)$ как наибольшее чётное число с этим свойством (в частности, $k \geq 4$). Пусть $\beta'(i)$ — это позиция самой правой отличной от нуля цифры в k-ичном представлении числа i. Определим

$$\mathsf{ttl}'(i) = \begin{cases} \frac{9}{2} \cdot k^{\beta'(i)} & \text{если } \beta'(i) > 0, \\ 4 & \text{иначе.} \end{cases} \tag{2.3}$$

Список SP' является аналогом списка SP из раздела 2.3.2. Он содержит, после i-й итерации, фреймы $I(j)$ для всех позиций $j \leq i$, таких что $j + \mathsf{ttl}'(j) > i$. Аналогично (2.2), мы определяем разбиение отрезка $[1; i]$ на полуинтервалы и затем считаем количество позиций фреймов из SP' в этих полуинтервалах. Полуинтервалы, справа налево, равны

$$(i-4, i], \left(i - \tfrac{9}{2}k, i-4\right], \left(i - \tfrac{9}{2}k^2, i - \tfrac{9}{2}k\right], \ldots, \left(i - \tfrac{9}{2}k^m, i - \tfrac{9}{2}k^{m-1}\right], \left(0, i - \tfrac{9}{2}k^m\right], \tag{2.4}$$

где $m = \left\lceil \log_k \frac{2i}{9} \right\rceil - 1$. Мы нумеруем интервалы с 0 до $m{+}1$.

Лемма 2.3.11. Каждый полуинтервал в (2.4) содержит не более 5 позиций фреймов, сохранённых в SP'. Каждый из полуинтервалов $0, ..., m$ содержит не менее 3 таких позиций.

Доказательство. Все 4 фрейма с позициями в 0-м полуинтервале хранятся в SP' по определению, см. (2.3). Для любого $j = 1, \ldots, m{+}1$, фрейм с позицией в j-м полуинтервале хранится в SP' тогда и только тогда, когда его позиция делится на k^j; см. (2.3). Длина этого полуинтервала меньше, чем $\frac{9}{2}k^j$, что даёт нам верхнюю оценку в $\left\lceil \frac{9}{2} \right\rceil = 5$ элементов. Аналогично, если $j \neq m{+}1$, то jй полуинтервал имеет длину $\frac{9}{2}k^j - \frac{9}{2}k^{j-1}$, а следовательно, содержит не менее $\left\lfloor \frac{9}{2}\frac{k-1}{k} \right\rfloor$ позиций фреймов, хранящихся в SP'. Так как $k \geq 4$, утверждение доказано. □

Теперь мы модифицируем алгоритм 2.3, заменив ttl на ttl' и SP на SP'.

Предложение 2.3.4. *Модифицированный алгоритм 2.3 находит палиндром длины $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$, используя $\mathcal{O}\left(\frac{\log n}{\log(1+\varepsilon)}\right)$ памяти.*

Доказательство. Пусть $S[i..j]$ — палиндром длины $L(S)$ и пусть $d = \left\lfloor \log_k \frac{L(S)}{4} \right\rfloor$. Без потери общности допустим, что $d \geq 0$, так как иначе $L(S) < 4 \leq \mathsf{ttl}'(i)$ и палиндром $S[i..j]$ будет найден точно. Так как $L(S) \geq 4k^d$, обозначим за $a_1 < a_2 < a_3 < a_4 < a_5$ последовательные позиции, кратные k^d (т.е., $\beta'(a_1), \ldots, \beta'(a_5) \geq d$), такие что $a_2 \leq \frac{i+j}{2} < a_3$. Тогда в частности $i < a_1$,

и существует палиндром $S[a_1..(i+j-a_1)]$, такой что $a_3 \le (i+j-a_1) < a_5$. Так как $a_1 + \mathsf{ttl}'(a_1) \ge a_5$, этот конкретный палиндром будет найден модифицированным алгоритмом 2.3; следовательно, $\ell(S) \ge a_3 - a_1 = 2k^d$. С другой стороны, мы знаем, что $L(S) < 4k^{d+1}$, то есть $\frac{L(S)}{\ell(S)} < 2k \le (1+\varepsilon)$.

Оценка на объём используемой памяти следует из оценки на размер списка SP', в котором не более $5\lceil \log_k \frac{2n}{9} \rceil + 1 = \mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ элементов. □

Далее мы опишем ускорение проверки палиндромов, аналогичное случаю $\varepsilon \le 1$. Мы будем использовать такое же определение значимых на i-й итерации фреймов, как в разделе 2.3.2. Сначала нам понадобится следующее свойство, являющееся более общим аналогом леммы 2.3.6; затем с его помощью мы докажем аналог леммы 2.3.7.

Лемма 2.3.12. Пусть на некоторой итерации список SP' содержит последовательные элементы $I(d), I(c)$, и при этом $d \le i - answer.len$, где i — номер текущей итерации. Пусть, кроме того, $I(a)$ — это другой элемент SP' на той же итерации, такой что $a < c$. Если c, d принадлежат одному полуинтервалу (2.4), то фрейм $I(a)$ не является значимым.

Доказательство. Пусть c, d принадлежат j-му полуинтервалу. Тогда они делятся на k^j и $d - c = k^j$. Так как $a < c$, a также делится на k^j. Одно из чисел $\frac{d+a}{2}, \frac{c+a}{2}$ делится на k^j; назовём это число b. Пусть $\delta = b - a$. Если $S[a..i]$ это палиндром, то $S[b..i-\delta]$ — также палиндром. Так как на i-й итерации левая граница j-го полуинтервала меньше чем c, то на $(i-\delta)$-й итерации эта левая граница была меньше чем b; следовательно, фрейм $I(b)$ был в SP' на $(i-\delta)$-й итерации, и палиндром $S[b..i-\delta]$ был найден. Его длина равна

$$i - \delta - b + 1 = i + 1 + a - 2b \ge i + 1 + a - (d + a) \ge i - d + 1 > answer.len,$$

что невозможно по определению $answer.len$. Таким образом, $S[a..i]$ не является палиндромом и утверждение доказано. □

Лемма 2.3.13. На каждой итерации список SP' содержит не более трёх значимых фреймов. Более того, если $I(d'), I(d)$ — последовательные элементы SP', такие что $i - d' < answer.len \le i - d$, где i — номер текущей итерации, то значимые фреймы являются последовательными элементами SP', начиная с $I(d)$.

Доказательство. Пусть $a < b < c < d$ такие, что элементы $I(d), I(c), I(b)$ последовательны в SP', и $I(a)$ хранится в SP'. Тогда или b, c, или c, d находятся в одном полуинтервале (2.4), и следовательно, a не является значимым по лемме 2.3.12. □

Для завершения доказательства, перейдём к операциям удаления из списка и докажем следующий аналог леммы 2.3.8.

Лемма 2.3.14. Функция $h(x) = x + \mathsf{ttl}'(x)$ отображает не более двух различных x в одно значение. Более того, если $h(x) = h(y)$ и $\beta'(x) \geq \beta'(y)$, то $\beta'(x) = \beta'(h(x)) + 1$ и $\beta'(y) = 0$.

Доказательство. Пусть $h(x) = h(y)$. Если $\beta'(x) = \beta'(y)$, то $\mathsf{ttl}'(x) = \mathsf{ttl}'(y)$ по (2.3), из чего следует $x = y$. Следовательно, все прообразы $h(x)$ имеют разные значения β'. Допустим что $\beta'(x) > \beta'(y)$. Тогда, для некоторого целого j, $x = j \cdot k^{\beta'(x)}$ и $h(x) = (j+4)k^{\beta'(x)} + \frac{k}{2} \cdot k^{\beta'(x)-1}$ по (2.3). Так как k чётное, получаем $\beta'(h(x)) = \beta'(x) - 1$. Если $\beta'(y) > 0$ — повторим то же рассуждение и получим $\beta'(x) = \beta'(y)$, что противоречит допущению. Следовательно, $\beta'(y) = 0$. □

Определим список $BS'(x)$, который содержит RLE-код[1] k-ичного представления x. Список $BS'(x)$ имеет размер $\mathcal{O}(\log_k n)$, может быть обновлён до $BS'(x+1)$ за $\mathcal{O}(1)$ времени и позволяет вычислить значение $\beta'(x)$ за $\mathcal{O}(1)$ времени (мы опустим доказательство, потому что оно аналогично доказательству леммы 2.3.9). Далее, лемма 2.3.10 выполняется для функции ttl', так что мы введём очереди $QU'(x)$ тем же образом, как были введены $QU(x)$ в разделе 2.3.2. Имея все эти ингредиенты, построим алгоритм M', ускоряющий модифицированный алгоритм 2.3 до алгоритма реального времени при помощи лемм 2.3.13 и 2.3.14. Алгоритмы M и M' очень похожи; единственное значительное отличие между ними заключается в процедуре удаления фреймов из списка (строки 5–9 алгоритма M и строки 5–15 алгоритма M'). Поскольку алгоритм M' находит тот же палиндром, что и модифицированный алгоритм 2.3, мы таким образом получаем результат теоремы 2.3.3.

[1]Кодирование длин серий (англ. run-length encoding, RLE) — это алгоритм сжатия данных, в котором каждая максимальная по включению последовательность (серия) из идущих подряд одинаковых символов заменяются на группу. Группа задаётся символом и длиной серии. *Пример: RLE код для строки* 00332222122 *равен* $(0,2)(3,2)(2,4)(1,1)(2,2)$.

Алгоритм 2.5. Алгоритм M', i-я итерация

add I to the beginning of SP'
if $i = 1$ **then**
    $sp \leftarrow first(SP')$
compute $BS'[i]$ from BS'; $BS' \leftarrow BS'[i]$; compute $\beta'(i)$ from BS'
if $QU'(\beta'(i)+1)$ is not empty **then**
    $v \leftarrow$ element of SP' pointed by $first(QU'(\beta'(i)+1))$
    if $v.i + ttl'(v.i) = i$ **then**
        if $v = sp$ **then**
            $sp \leftarrow next(sp)$
        delete v; delete $first(QU'(\beta'(i)+1))$
$v \leftarrow$ element of SP' pointed by $first(QU'(0))$
if $v.i + ttl'(v.i) = i$ **then**
    if $v = sp$ **then**
        $sp \leftarrow next(sp)$
    delete v; delete $first(QU'(0))$
add pointer to $first(SP')$ to $QU'(\beta'(i))$
read $S[i]$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
$sp \leftarrow previous(sp)$ ▷ если существует
while $i - sp.i + 1 \leq answer.len$ and $sp \neq last(SP')$ **do**
    $sp \leftarrow next(sp)$
for all existing v in $\{sp, next(sp), next(next(sp))\}$ **do**
    if $S[v.i..i]$ is a palindrome and $answer.len < i - v.i + 1$ **then**
        $answer \leftarrow (v.i, i - v.i + 1)$

Глава 3

Повторы и обратные повторы в потоках

3.1 Введение

Мы рассматриваем задачу поиска самой длинной строки, встречающейся во входном потоке хотя бы два раза (наибольший повтор, LRS) и задачу поиска самой длинной строки, которая имеет прямое и обратное вхождение в поток (наибольший обратный повтор, LRRS); см. раздел 1.2.1. Решение LRS в стандартной RAM-модели за линейное время является хорошо известным приложением суффиксных деревьев, впервые упомянутым в пионерской статье Вейнера [57]. Задача LRRS может быть решена аналогично (например, построением суффиксного дерева над обратной входной строкой). Заметим, что LRS и LRRS также допускают эффективные параллельные алгоритмы [5].

В потоковой модели эти задачи могут быть решены только приближенно и только с высокой вероятностью, как показано ниже в разделе 3.6. Наш основной результат — это алгоритмы типа Монте-Карло для обеих задач, решающие их с аддитивной погрешностью приближения E, с использованием $\mathcal{O}(\frac{n}{E} + E)$ памяти. Мы опишем два алгоритма для каждой задачи; «простой» алгоритм работает за $\mathcal{O}(n)$ времени суммарно, но имеет время обновления $\mathcal{O}(\frac{n}{E})$. Более эффективные версии позволяют делать обновление за $\mathcal{O}(\log n)$ времени, то есть работают «почти» в реальном времени; общее время работы при этом равно $\mathcal{O}(n + \frac{n}{E} \log n)$. После вводных замечаний, мы опишем основную идею получения сублинейной памяти в разделе 3.3, алгоритмы для LRRS в разделе 3.4, алгоритмы для LRS в разделе 3.5, нижние оценки на память в разделе 3.6.

3.2 Определения

Повтор в S — это пара равных подстрок $S[i..i+l-1] = S[j..j+l-1]$, где $i < j$; мы обозначаем повторы тройками (i, j, l) и пишем $L(S)$ для обозначения мак-

симальной длины повтора в S. Аналогично, условие $S[i..i+l-1] = \overleftarrow{S}[j..j+l-1]$, $i \le j$, определяет *обратный повтор* (i, j, l) в S; мы пишем $\overleftarrow{L}(S)$ для обозначения максимальной длины обратного повтора в S. Заметим, что палиндром $S[i..i+l-1]$ является обратным повтором (i, i, l); более того, если $i + l \ge j - 1$ для обратного повтора (i, j, l), то $S[i..j+l-1]$ — это палиндром длины $l + j - i$. Таким образом, длиннейший обратный повтор в S или состоит из неперекрывающихся вхождений («левое вхождение» $S[i..i+l-1]$ заканчивается до того как начинается «правое вхождение» $S[j..j+l-1]$), или является палиндромом.

Наблюдение 3.2.1. Наибольший палиндром в потоке может быть приближен с использованием $\mathcal{O}(\frac{n}{E})$ памяти и временем обновления $\mathcal{O}(1)$ алгоритмом A, что лучше, чем мы можем получить для LRRS. Таким образом, алгоритму, разрабатываемому для решения LRRS, достаточно обрабатывать только *некоторые* обратные повторы в потоке (включая *все непекрывающиеся*); мы подразумеваем, что алгоритм для поиска наибольшего палиндрома запущен параллельно, и возвращается наибольший из двух результатов.

Алгоритмы используют хэш Карпа–Рабина (см. раздел 1.1.4). Приведем необходимые детали использования хэша в задачах LRS и LRRS.

Пусть p — фиксированное простое число из отрезка $[n^{3+\alpha}, n^{4+\alpha}]$ для некоторого $\alpha > 0$, и r — фиксированное целое число, случайно и равновероятно выбранное из отрезка $\{1, \ldots, p-1\}$. Для строки S, её прямой хэш и обратный хэш определены, соответственно, как

$$\phi^F(S) = \Big(\sum_{i=1}^{n} S[i] \cdot r^i\Big) \bmod p \text{ и } \phi^R(S) = \Big(\sum_{i=1}^{n} S[i] \cdot r^{n-i+1}\Big) \bmod p.$$

Очевидно, прямой хэш строки совпадает с обратным хэшом обратной строки; этот факт используется для нахождения обратных повторов. Вероятность коллизии хэшей для двух строк длины m не больше m/p; а значит, для алгоритма с линейным числом сравнений вероятность хотя бы одной коллизии есть $\mathcal{O}(n^{-1-\alpha})$ по выбору p. Все последующие рассуждения подразумевают, что коллизий нет. Для входного потока S мы определяем $F^F(i, j) = \phi^F(S[i..j])$ и $F^R(i, j) = \phi^R(S[i..j])$. Хэши подстрок строки S могут быть получены за константное время из хэшей префиксов строки S, см. предложение 1.1.1.

Для входного потока S, кортеж $I(i) = (i, F^F(1, i-1), F^R(1, i-1), r^{1-i} \bmod p, r^i \bmod p)$ является его i-*м фреймом*. По заданному фрейму $I(i)$ и символу $S[i]$, фрейм $I(i+1)$ может быть вычислен за $\mathcal{O}(1)$ времени, см. Предложение 2.3.1(1).

Мы пишем log для двоичного логарифма и $a \overline{\bmod} b$ вместо $(a - 1) \bmod b + 1$.

Суффиксные деревья

Напомним определение и устройство классической индексной структуры данных, называемой *суффиксным деревом* строки [57]; см. пример на рис. 3.1. *Бор*

множества строк U — это дерево, вершинами которого являются все префиксы строк из U, а ребра помечены и имеют вид $u \xrightarrow{a} ua$, где a — символ алфавита. *Терминальные вершины* бора — это вершины, соответствующие строкам из U; все листья бора являются терминальными вершинами. *Суффиксный бор* строки w — это бор множества всех суффиксов w. Суффиксное дерево $\mathcal{T}(w)$ строки w получается из суффиксного бора этой строки процедурой «сжатия», при которой оставляются только вершины, имеющие более одного сына, и терминальные вершины; цепи между такими вершинами превращаются в рёбра, помеченные подстроками из w. Метки рёбер не хранятся в виде строк; обычный подход заключается в том, чтобы хранить в вершине u ее «строковую глубину» $sd(u)$ в дереве, которая совпадает с длиной строки u, а также последнюю позицию некоторого вхождения u в w. Первая позиция этого вхождения находится по разности строковой глубины самой вершины u и ее родителя.

В итоге, дерево $\mathcal{T}(w)$ имеет $\mathcal{O}(|w|)$ вершин и рёбер, занимает $\mathcal{O}(|w|)$ машинных слов памяти; сама строка w является частью структуры данных. Каждая вершина идентифицируется меткой пути от корня до неё. Не все подстроки w соответствуют вершинам; в общем случае, подстроки адресуются *позициями*. Позиция в суффиксном дереве — это пара $pos = (v, raise)$, где v — вершина и $raise \geq 0$. Эта позиция соответствует префиксу строки v длины $|v| - raise$, и указывает на «локацию» в $\mathcal{T}(w)$ на входящем в v ребре, $raise$ символов над v. Для навигации в суффиксном дереве используются (прямые) *суффиксные ссылки*; для вершины v, суффиксная ссылка $link(v)$ — это наибольший собственный суффикс v. Подобная ссылка для произвольной позиции называются *неявной суффиксной ссылкой* и не хранится; вычисление таких ссылок является ключевым примитивом для работы с суффиксными деревьями.

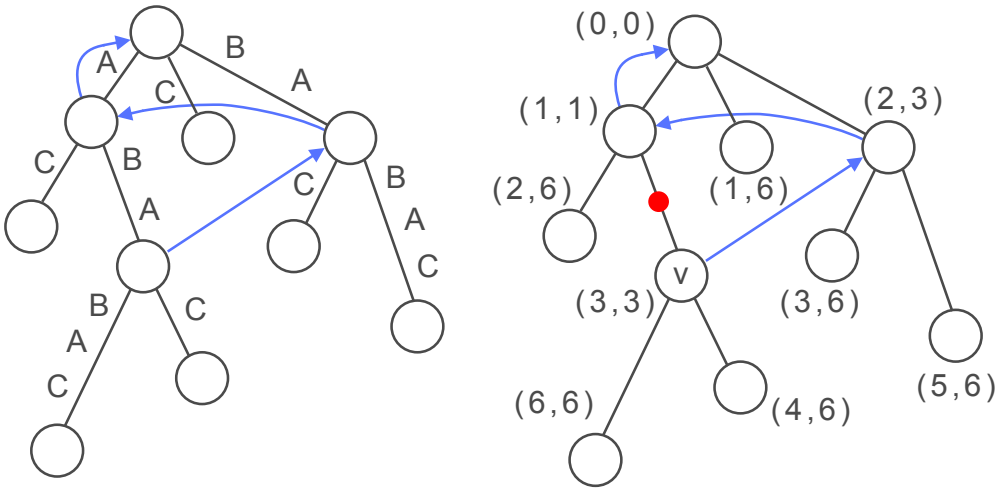

Рис. 3.1: Суффиксное дерево $\mathcal{T}(ABABAC)$. Слева метки на рёбрах записаны в виде строк. Справа, в том же суффиксном дереве, метки записаны в виде (sd, последняя позиция вхождения). На обеих картинках цветом изображены суффиксные ссылки. На правой картинке красная точка соответствует позиции $pos = (v, 1)$.

Существует два основных подхода к построению суффиксного дерева. Алго-

ритм Вейнера [57] (и его модификации) добавляет суффиксы слова в дерево в порядке увеличения их длины. Можно считать, что это онлайн-алгоритм, строящий суффиксное дерево обратной строки. За одну итерацию алгоритм Вейнера добавляет к дереву одну терминальную вершину и не более одной нетерминальной. Алгоритм Укконена [56] (и его модификации) — это онлайн-алгоритм, поддерживающий суффиксное дерево при добавлении символов к концу строки. За одну итерацию алгоритму может потребоваться создать несколько (вплоть до $\Omega(n)$) нетерминальных вершин.

3.3 Сведение к сжатым повторам

Основная сложность задач LRS и LRRS заключается в том, что они не локальны: два вхождения могут быть разделены, например, $\Omega(n)$ символами. Для того, чтобы не хранить весь входной поток, мы вычисляем хэш от подстрок (*блоков*) фиксированного размера b, рассматриваем хэши как новые символы (мы называем их *хэш-буквами*) и ищем повторяющиеся строки из хэш-букв. Мы определяем *прямой след* и *обратный след* потока S, с размером блока b и сдвигом r, как следующие строки над алфавитом $\{0, \dots, p-1\}$:

$$P_{r,b} = F^F(r, r+b-1)F^F(r+b, r+2b-1)\cdots F^F(r+x_{r,b}b, r+(x_{r,b}+1)b-1),$$
$$Q_{r,b} = F^R(r+x_{r,b}b, r+(x_{r,b}+1)b-1)\cdots F^R(r+b, r+2b-1)F^R(r, r+b-1),$$

где $x_{r,b} = \left\lfloor \frac{|S|+1-r}{b} \right\rfloor - 1$.

Зафиксируем b и рассмотрим только следы с размером блока b и сдвигами $r = 1, \dots, b$; будем использовать обозначения P_r, Q_r, x_r вместо $P_{r,b}, Q_{r,b}, x_{r,b}$; следы P_b и Q_b называются *основными*. Будем говорить, что подстрока $S[l..t]$ *содержится* в следах P_r и Q_r, если $l \equiv_b r$ и $t - l + 1 \equiv_b 0$. Это условие означает, что $S[l..t]$ может быть разбита на блоки, которые соответствуют хэш-буквам, и после вычисления прямых (соотв., обратных) хэшей образует подстроку следа P_r (соотв., Q_r).

Мы храним только основной след (P_b для LRS и Q_b для LRRS) и ищем повторы, у которых левое вхождение содержится в основном следе. Мы называем такие повторы *сжимаемыми*. Так как левое вхождение фиксирует длину, правое вхождение сжимаемого повтора автоматически содержится в каком-то следе. Длинные сжимаемые повторы дают приближенное решение для LRS и LRRS.

Наблюдение 3.3.1. Для каждого повтора (обратного повтора) (i, j, l), такого что $l > 2b - 2$, поток S содержит сжимаемый повтор (соотв., сжимаемый обратный повтор) (i', j', l') такой что $l' \geq l - 2b + 2$. При этом параметры этого сжимаемого повтора определяются равенствами $i' = \lceil \frac{i}{b} \rceil b$, $j' = j + i' - i$, $l' = \lfloor \frac{i+l-i'}{b} \rfloor b$.

Перейдем от сжимаемых повторов к их образам в следах. *Сжатый повтор* в строке S, заданный четвёркой (r, i_l, i_r, k), это пара подстрок $P_b[i_l \,..\, i_l+k-1] =$

$P_r[i_r \mathinner{..} i_r{+}k{-}1]$, где $i_l < i_r$ и $1 \le r \le b$ (см Рис. 3.2). Аналогично, *сжатый обратный повтор* — это пара $Q_b[x_r{+}2{-}i_l \mathinner{..} x_r{+}3{-}i_l{-}k] = P_r[i_r \mathinner{..} i_r{+}k{-}1]$.

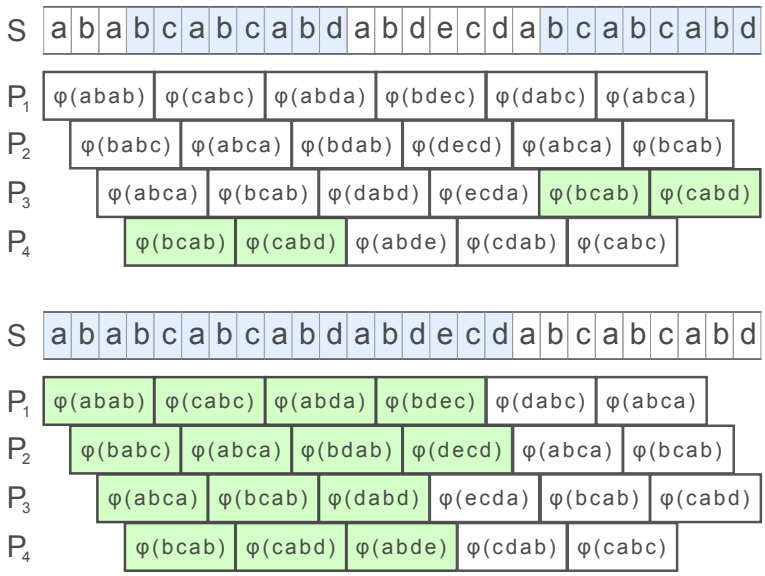

Рис. 3.2: Следы и сжатые повторы. Сверху: сжимаемый повтор $(4, 19, 8)$ *и его сжатый повтор* $(3, 1, 5, 2)$ *(выделено цветом). Снизу: хэш-буквы, доступные во всех прямых следах после прочтения* $S[17]$ *(выделено цветом).*

Наблюдение 3.3.2. Если (i, j, l) — сжимаемый повтор (сжимаемый обратный повтор), то кортеж $(j \bmod b, i/b, \lfloor j/b \rfloor{+}1, l/b)$ является сжатым повтором (соотв., сжатым обратным повтором). И наоборот, если (r, i_l, i_r, k) — сжатый повтор (сжатый обратный повтор), то $(i_l b, (i_r{-}1)b{+}r, kb)$ является сжимаемым повтором (соотв., сжимаемым обратным повтором), с точностью до коллизии хэшей.

С помощью наблюдений 3.3.1 и 3.3.2 мы свели исходную задачу к задаче поиска наибольшего сжатого повтора и наибольшего сжатого обратного повтора в потоке; их решения дадут, с высокой вероятностью, решения задач LRS и LRRS с аддитивной погрешностью меньше $2b$. По наблюдению 3.2.1, алгоритм для сжатых обратных повторов может безопасно пропускать некоторые такие повторы, если соответствующие сжимаемые обратные повторы перекрываются.

Хэш-буква имеет вид $F^F(j, j+b-1)$ или $F^R(j, j+b-1)$. Эта хэш-буква становится *доступной*, когда мы прочитали все символы $S[j], S[j{+}1], \ldots, S[j{+}b{-}1]$, то есть после чтения $S[j{+}b{-}1]$. Следовательно, после каждого чтения, начиная с $S[b]$, в одном из следов и одном из обратных следов становится доступна новая хэш-буква. Для вычисления хэш-буквы $F^F(j, j+b-1)$ или $F^R(j, j+b-1)$ достаточно знать фреймы $I(j{+}b)$ и $I(j)$. Перед чтением $S[i]$ мы храним $I(i{-}b{+}1), I(i{-}b{+}2), \ldots, I(i)$. Номера этих фреймов попарно различны по модулю b, а значит, фреймы могут быть сохранены в циклической очереди длины b. Префикс следа P_r (соотв., суффикс следа Q_r), доступный после чтения $S[i]$, будем обозначать через P_r^i (соотв., Q_r^i). Заметим, что $P_r^i = P_r[1..\lfloor \frac{i-r+1}{b} \rfloor]$, аналогично для Q_r^i.

3.4 Поиск наибольшего обратного повтора

По наблюдению 3.2.1, достаточно проанализировать любое множество обратных повторов, содержащее все неперекрывающиеся. Если обратный повтор — неперекрывающийся, то сжимаемый обратный повтор, получаемый из него согласно наблюдению 3.3.1, также неперекрывающийся.

Определим аналог неперекрывающегося повтора для сжатых повторов. *Удобный повтор* — это сжатый обратный повтор (r, i_l, i_r, k), такой что его левое вхождение является подстрокой в $Q_b^{i_r b-1}$. Данное свойство гарантирует, что все хэш-буквы из левого вхождения повтора станут доступны до того, как будет доступна первая буква правого вхождения.

Лемма 3.4.1. Сжатый обратный повтор, соответствующий неперекрывающемуся сжимаемому обратному повтору, удобен.

Доказательство. Пусть (i, j, l) — неперекрывающийся сжимаемый обратный повтор; по наблюдению 3.3.2 соответствующий сжатый обратный повтор это $(j \bmod b, i/b, \lfloor j/b \rfloor + 1, l/b)$. Его левое вхождение содержится в Q_b^{i+l-1}, и $i+l-1 < j-1 < (\lfloor j/b \rfloor + 1)b - 1$, то есть он удобный по определению. □

По Лемме 3.4.1, для получения желаемого приближения для LRRS достаточно найти наибольший удобный повтор в S. Мы используем алгоритм Вейнера [57] для поддержания динамического суффиксного дерева для Q_b (алгоритм Вейнера поддерживает обратные суффиксные ссылки, но может также поддерживать и прямые суффиксные ссылки, см.[12]). После обработки символа $S[j]$, для каждого $j \le n$, дерево равняется $\mathcal{T}(Q_b^j)$. При обработке символа $S[i]$, мы добавляем новую хэш-букву к следу P_r, где $r = (i+1) \bmod b$, и ищем наибольший удобный повтор, правое вхождение которого является суффиксом P_r^i; мы обозначаем этот суффикс $suff_r^i$ (иными словами, $suff_r^i$ — самый длинный суффикс в P_r^i, входящий в Q_b^{r-1}). Так как повтор удобный, последняя хэш-буква левого вхождения была добавлена в дерево раньше, чем первая хэш-буква правого вхождения стала доступна. Из этого условия следует, что удобный повтор длиннее одной хэш-буквы расширяет некоторый удобный повтор, найденный на итерации с номером $i - b$.

Наблюдение 3.4.1. По определению удобного повтора, из условия $P_r^i = P_r^{i-1}$ следует $suff_r^i = suff_r^{i-1}$. Следовательно $suff_r^{i-1} = suff_r^{i-b}$, если $i + 1 \equiv_b r$.

Другими словами, обновление основного следа Q_b не влияет на множество удобных повторов со вторым вхождением, являющимся суффиксом P_r^i.

Для каждого $r = 1, \ldots, b$ мы поддерживаем позицию строки $suff_r^i$ в дереве $\mathcal{T}(Q_b^i)$. Будем обозначать эту позицию pos_r^i.

Наблюдение 3.4.2. Равенство $suff_r^{i-1} = suff_r^{i-b}$, отмеченное в наблюдении 3.4.1, не обязательно влечёт равенство позиций $pos_r^{i-1} = pos_r^{i-b}$. Последнее равенство

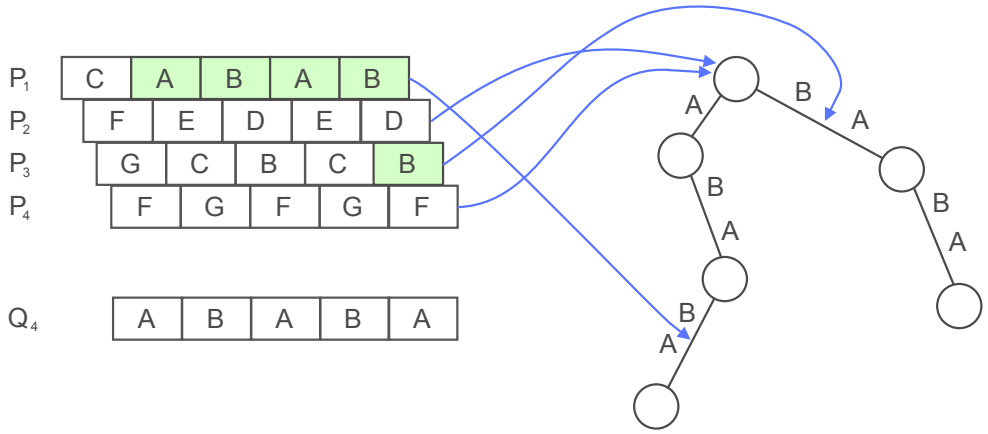

Рис. 3.3: Суффиксное дерево построенное по основному следу Q_4, равное $\mathcal{T}(ABABA)$. В следах $P_1, \ldots, P_4$ цветом выделены $suff_r^i$; стрелками показаны pos_r^i.

выполняется тогда и только тогда, когда пара $pos_r^{i-b} = (v, raise)$ корректно указывает на позицию $w = suff_r^{i-b}$ в дереве $\mathcal{T}(Q_b^{i-1})$. Корректность могла нарушиться из-за новых вершин, появившихся во время последнего обновления дерева. Они могли разделить входящее в v ребро, так что позиция w могла оказаться над непосредственным предком v. При этом, если мы поднимемся вверх на $raise$ символов из вершины v, мы всё равно окажемся в позиции w.

После чтения $S[i]$ нам нужно вычислить pos_r^i, где $r = (i{+}1) \bmod b$. Мы можем предположить, что после чтения $S[i-b]$ мы вычислили pos_r^{i-b}. По наблюдению 3.4.2, мы можем получить pos_r^{i-1} из $pos_r^{i-b} = (v, raise)$ подъёмом по дереву из v до нижнего конца ребра, которое содержит позицию на $raise$ символов выше v.

Предложение 3.4.1. *Если $i + 1 \equiv_b r$, то наибольший собственный суффикс $suff_r^i$ является префиксом $suff_r^{i-1}$.*

Доказательство. Пусть $k \geq 1$ и $(r, i_l, \frac{i-r+1}{b} - k + 1, k)$ — это удобный повтор соответствующий $suff_r^i$. Наибольший собственный префикс $suff_r^i$ соответствует удобному повтору $(r, i_l{+}1, \frac{i-r+1}{b}{-}k{+}1, k{-}1)$; его правое вхождение заканчивается в позиции $\frac{i-r+1}{b} - 1$, то есть является суффиксом P_r^{i-b}. Таким образом, это вхождение является суффиксом $suff_r^{i-b}$ по определению. По наблюдению 3.4.1, $suff_r^{i-b} = suff_r^{i-1}$. □

Из предложения 3.4.1 следует, что мы можем найти позицию pos_r^i в дереве $\mathcal{T}(Q_b^i)$ следующим образом: возьмём наибольший суффикс строки $suff_j^{i-1}$, такой что из его позиции в дереве есть переход по хэш-букве $a = F^F[i{-}b{+}1..i]$; обозначим этот суффикс w, и совершим из w переход по a, полученная позиция и будет pos_r^i; если корень $\mathcal{T}(Q_b^i)$ не имеет перехода по a, pos_r^i совпадает с корнем. Для нахождения w, мы можем перебрать суффиксы $suff_j^{i-1}$ в порядке убывания длины, переходя по суффиксным ссылкам. Если текущая позиция находится на

ребре (u, v), на t символов ниже верхнего конца ребра (u), то его неявная суффиксная ссылка может быть найдена спуском на t символов из вершины $link(u)$. Наконец, если $i \overline{\bmod} b = b - 1$, то новая хэш-буква добавляется к Q_b, и для обновления суффиксного дерева должна быть выполнена итерация алгоритма Вейнера. Теперь сформулируем теорему.

Теорема 3.4.1. Существует потоковый алгоритм типа Монте-Карло, решающий задачу LRRS с заданной аддитивной погрешностью $E = \mathcal{O}(n^{0.99})$, использующий $\mathcal{O}(\frac{n}{E} + E)$ машинных слов памяти, работающий за $\mathcal{O}(n)$ времени суммарно и имеющий время обновления $\mathcal{O}(\frac{n}{E})$.

Замечание. Все представленные в этой главе алгоритмы эффективно работают в случае $E = \mathcal{O}(\sqrt{n})$. Для решения с большей допустимой погрешностью, имеет смысл использовать алгоритмы с параметром $E = \sqrt{n}$, в этом случае используемый объём память оптимален.

Доказательство. Зафиксируем $b = \lfloor \frac{E}{2} \rfloor$ и будем писать $\mathcal{T}_j$ для обозначения суффиксного дерева $\mathcal{T}(Q_b^j)$. Мы представим алгоритм ARR, который находит, с высокой вероятностью, наибольший удобный повтор с размером блока b в потоке S длины n и удовлетворяет ограничениям на память и время, указанным в теореме. Сравнивая соответствующие сжимаемые повторы с наибольшим палиндромом в S, найденным алгоритмом A, и выбирая более длинный из двух, мы получаем, с высокой вероятностью, решение задачи LRRS(S) с ограничениями на погрешность, память и время, заданными в теореме; см. наблюдения 3.2.1, 3.3.1, 3.3.2, и лемму 3.4.1.

Во время работы алгоритма мы поддерживаем суффиксное дерево $\mathcal{T}$, массив $pos[1..b]$ позиций в $\mathcal{T}$, массив $SI[1..b]$ недавних фреймов и последний фрейм I. Под *i-й итерацией* мы подразумеваем, как обычно, все операции, начиная с чтения $S[i]$ и до чтения $S[i{+}1]$. После $(i{-}1)$-й итерации, $\mathcal{T} = \mathcal{T}_{i-1}$, $pos[r]$ содержит последнее вычисленное значение pos_r^j (т.е. $j \geq i - b$ для каждого r), и SI содержит фреймы $I(j)$ для $j = i - b + 1, \ldots, i$ так, что $I(j)$ сохранено в $SI[(j{-}1) \overline{\bmod} b]$. Тогда i-я итерация выглядит следующим образом:

Алгоритм 3.1. Algorithm ARR (AdditiveReversedRepeat), i-я итерация $(i \geq b)$

1: $r = i \bmod b + 1$
2: read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
3: compute $FF = F^F(i - b + 1, i)$ from I and $SI[r]$
4: update $pos[r] = pos_r^{i-b}$ to pos_r^{i-1} by walking up
5: compute $pos[r] = pos_r^i$ by traversing $\mathcal{T}$ from pos_r^{i-1} by the hashletter FF
6: update *answer* by the value determined by $pos[r]$
7: **if** $r = b$ **then**
8: compute $FR = F^R(i - b + 1, i)$ from I and $SI[r]$
9: update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR by Weiner's algorithm
10: $SI[r] = I$

Обновление $pos[r]$ в строках 4-5 корректно по наблюдению 3.4.2 и предложению 3.4.1. Для обновления ответа в строке 6 мы храним в каждой вершине суффиксного дерева её глубину sd (длину соответствующей вершине строки). Тогда новое значение позиции $pos[r]$, равное $(v, raise)$, задаёт длину соответствующего удобного повтора, равную $sd(v) - raise$. Из длины и номера итерации мы получаем правое вхождение повтора. Для нахождения левого вхождения напомним, что ребро в суффиксном дереве с нижним концом v помечено последней позицией некоторого вхождения v в строку; вычитая $sd(v) - 1$ из этой позиции, мы получаем первую позицию левого вхождения повтора. Таким образом, алгоритм ARR корректно вычисляет наибольший удобный повтор. Оценим его время работы и используемую память.

Суффиксное дерево имеет размер $\mathcal{O}(\frac{n}{E})$, массив позиций, как и массив фреймов, имеет размер $\mathcal{O}(E)$, что даёт требуемое ограничение на память.

Мы храним все ребра суффиксного дерева в одной хэш-таблице, используя пары (вершина, символ) в качестве ключей; в качестве метода применяем динамическое совершенное хэширование [23]. Этот метод обеспечивает поиск за константное время; с вероятностью не меньше $1 - \frac{1}{n}$ все обновления работают также за константное время. Если обновление занимает больше чем предопределённое константное время, мы выдаём повтор $(1, 1, n)$ и прекращаем работу. Это ведёт к маловероятной ошибке той же стороны (ложно положительная), что и коллизия хэшей. Таким образом переход в суффиксном дереве выполняются за константное время на каждое пройденное ребро.

На каждой итерации потенциально тяжелые вычисления расположены в строках 5 и 9. Каждое из них, в худшем случае, требует времени, пропорционального размеру суффиксного дерева, т.е., $\mathcal{O}(\frac{n}{E})$. Заметим, что строка 4 требует константного числа операций, потому что алгоритм Вейнера добавляет к дереву не более одной внутренней вершины за итерацию.

Для оценки общего времени, посчитаем общее количество элементарных операций в каждой из строк 5 и 9 за все итерации. В строке 9 суффиксное дерево строится для строки длины $\mathcal{O}(\frac{n}{E})$ над полиномиальным алфавитом (p полиномиально от n, а значит, и от $\frac{n}{E}$). Это может быть сделано за время $\mathcal{O}(\frac{n}{E})$ с использованием хэш-таблицы. Теперь рассмотрим строку 5, сгруппировав все итерации с одним следом P_r. Рассмотрим изменения строковой глубины позиции $pos[r]$. Изначально $sd = 0$. Каждый переход по суффиксной ссылке уменьшает её на 1; она может увеличиться на 1 за итерацию, если переход по текущему символу найден. Таким образом общее число переходов по суффиксным ссылкам есть $\mathcal{O}(\frac{n}{E})$, каждый переход выполняется за константное время. Наконец, посчитаем число спусков, которые следуют за переходом по суффиксной ссылке. Рассмотрим древесную глубину td позиции $pos[r]$. Изначально $td = 0$, и td ограничено глубиной дерева, то есть $\mathcal{O}(\frac{n}{E})$. Переход по суффиксной ссылке уменьшает td не более чем на 1; каждый подъём в строке 4 уменьшает её на 1; обновления дерева не уменьшают её. Так как общее число подъёмов в строке 4 и переходов по суффиксным ссылкам в строке 5 ограничены $\mathcal{O}(\frac{n}{E})$, общее число

спусков также является $\mathcal{O}(\frac{n}{E})$. Суммируя по всем следам, получаем общее время $\mathcal{O}(n)$, как и требуется.

Таким образом, задача LRRS может быть решена с ограничениями на память и время работы, указанными в теореме. □

Далее мы модифицируем алгоритм ARR так, чтобы избежать медленных обновлений.

Теорема 3.4.2. Существует потоковый алгоритм типа Монте-Карло, решающий задачу LRRS с заданной аддитивной погрешностью $E = \mathcal{O}(n^{0.99})$, использующий $\mathcal{O}(\frac{n}{E} + E)$ машинных слов памяти, работающий за $\mathcal{O}(n + \frac{n}{E} \log n)$ времени суммарно и имеющий время обновления $\mathcal{O}(\log n)$.

Доказательство. Алгоритм ARR имеет две медленные части: обновление суффиксного дерева и поиск в нём. Для первой части мы заменим в построении алгоритм Вейнера его модификацией из работы Амира и др. [2]. Эта модификация может работать с полиномиальным целочисленным алфавитом, имеет время обновления $\mathcal{O}(\log n)$ и суммарное время работы $\mathcal{O}(n \log n)$, таким образом добавляя $\frac{n}{E} \log n$ к суммарному времени работы нашего алгоритма.

Мы должны обратить внимание, что алгоритм Амира и др. не использует суффиксные ссылки; вместо этого, позиция добавления нового суффикса к существующему дереву определяется через запрос к «сбалансированной индексной структуре» (BIS). BIS позволяет находить такую позицию за $\mathcal{O}(\log |\mathcal{T}|)$ времени; в дополнение, BIS позволяет находить наибольший общий префикс произвольной строки с деревом $\mathcal{T}$, также за $\mathcal{O}(\log |\mathcal{T}|)$ времени. Это позволяет нам строить прямые суффиксные ссылки для новых вершин с теми же ограничениями по времени.

Поиск будет использовать «отложенные» операции. Для каждого сдвига $r = 1, \ldots, b$ мы поддерживаем *конвейер* C_r, состоящий из позиции $pos = pos[r]$ в дереве $\mathcal{T}$, очереди хэш-букв, ожидающих обработки (она может рассматриваться как суффикс строки P_r) и флага $flag$ установленного в 0, если алгоритм должен пропустить обновление ответа текущим значением pos. Одна итерация алгоритма показана ниже.

Алгоритм 3.2. Алгоритм FastARR, i-я итерация ($i \geq b$)

$r = i \bmod b + 1$
read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
compute $FF = F^F(i-b+1, i)$ from I and $SI[r]$; put (C_r, FF)
$C_r = lazyProcess(C_r)$; $flag = [queue \text{ is empty}]$
if $flag = 1$ **then**
    update $answer$ by value determined by C_r
if $r = b$ **then**
    compute $FR = F^R(i-b+1, i)$ from I and $SI[r]$
    update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR and building suffix links using BIS
$SI[r] = I$

Новая хэш-буква добавляется к текущему конвейеру в строке 3; функция $lazyProcess$, вызываемая в строке 4, совершает два отложенных перехода в $\mathcal{T}$ (по суффиксным ссылкам или символам из очереди). Если доступен только один переход (по единственному символу из очереди), выполняется только он.

Пусть $C = C_r$ это текущий конвейер, $maxlen$ это максимальная строковая глубина среди всех позиций pos, которые были в C (= длина наибольшего удобного повтора с правым вхождением в P_r, найденного к текущему моменту). После каждого вызова $lazyProcess$ поддерживается следующий полу-инвариант: $balance = maxlen - sd(pos) - 2|queue| \geq 0$. Добавление символа в очередь $queue$ уменьшает $balance$ на 2; переход по суффиксной ссылке увеличивает его на 1. Переход по символу обновляет pos, а следовательно, за ним следует удаление символа из $queue$; таким образом $balance$ увеличивается на 1 или даже на 2 (если новая pos изменяет $maxlen$). Таким образом, если $lazyProcess$ совершает два перехода, то $balance$ не уменьшается, а если был совершен один переход, то $|queue| = 0$ и $balance \geq 0$ по определению.

Ввиду этого полу-инварианта, из условия $|queue| > 0$ следует $sd(pos) + |queue| < maxlen$. Это означает, что после обработки всей очереди и $maxlen$, и наибольший удобный повтор не могут быть обновлены. Следовательно, в конце итерации $flag$ выставлен в 0, если $|queue| > 0$, и в 1 в противном случае.

Наблюдение 3.4.3. Между итерацией, на которой хэш-буква была добавлена в очередь и итерацией, на которой она была обработана для получения pos, суффиксное дерево может быть обновлено, и неправильные (слишком глубокие) pos могут быть найдены. Однако, если очередь не пуста, ответ не обновится, потому что $flag = 0$. Когда очередь станет пустой, позиция pos будет корректной, так как последний символ в очереди появился после последнего обновления $\mathcal{T}$.

Из наблюдения 3.4.3 следует, что алгоритм FastARR совершает ровно те же обновления ответа, что и алгоритм ARR, и следовательно, корректен. Теперь оценим используемые время и память.

Мы не можем хранить каждую очередь $queue$ явно из-за ограничений на память. Вместо этого, мы разобьём строку $queue$ на префикс (или голову) $head$ и суффикс (или хвост) $tail$ и будем хранить их раздельно. Мы будем использовать всего четыре машинных слова памяти и поддерживать три операции, каждую за $\mathcal{O}(1)$ времени: добавить символ в конец $tail$; удалить первый символ из $head$; установить $head$ в $tail$, если $head$ пусто. Это обеспечивает полную функциональность очереди; обсудим теперь детали.

Хвост хранится как позиция $posT = (v, raise)$ строки $tail$ в дереве $\mathcal{T}$. Голова представляется двумя числами L и R, такими что отрезок $[L..R]$ основного следа Q_b равен голове; напомним что Q_b является частью структуры данных суффиксного дерева. Для удаления первой хэш-буквы из очереди, мы увеличиваем L на 1. Если $L > R$ (= очередь пуста), мы вычисляем отрезок Q_b равный $tail$: если $posT = (v, raise)$, входящее в v ребро имеет метку последней позиции x вхождения v в Q_b, значит $tail = Q_b[x - sd(v) + 1..x - raise]$. Тогда мы выставляем

$[L..R]$ равным этому интервалу и присваиваем $posT$ значение $(root, 0)$, таким образом перенося $tail$ в $head$.

Для добавления хэш-буквы c в очередь, мы пытаемся сделать переход по c из $posT$. Если попытка успешна, мы просто обновляем $posT$ полученной позицией. Если не успешна, строка $tail \cdot c$ не встречается в Q_b и мы обновляем текущий конвейер следующим образом: $pos = posT$; $posT = (root, 0)$; $L = R$ указывает на вхождение c. Если такого вхождения нет, мы выставляем $pos = posT = (root, 0); L = R = -1$. Обоснованием такого обновления является наблюдение 3.4.3: так как $maxlen$ не может быть обновлен до того как очередь станет пустой, мы прерываем текущее обновление pos и пропускаем некоторые промежуточные обновления которые не могут привести к обновлению ответа (потому что $flag = 0$); после этого мы используем хвост (без нового символа) для определения нового значения $suff_r$.

Далее мы оптимизируем вычисления в суффиксном дереве с использованием структуры данных «динамическое дерево взвешенных предков» (DWAT) [43] для $\mathcal{T}$, которая поддерживается параллельно с $\mathcal{T}$; это дерево имеет размер $\mathcal{O}(|\mathcal{T}|)$, его обновление происходит за $\mathcal{O}(\frac{\log^2 \log |\mathcal{T}|}{\log \log \log |\mathcal{T}|})$ времени на каждое обновление суффиксного дерева.

DWAT использовалось для поиска неявных суффиксных ссылок в одной из вариаций суффиксного дерева следующим образом [19]: спуск в ближайшую вершину, переход по суффиксной ссылке, подъём с помощью одного запроса к DWAT, работающего за $\mathcal{O}(\log \log |\mathcal{T}|)$ времени. Если мы применим этот метод, общее время работы алгоритма увеличится до $\Theta(n \log \log n)$. Чтобы избежать этого, мы используем следующий трюк. Сначала мы пытаемся найти неявную суффиксную ссылку обычной процедурой: подъём до ближайшей вершины, переход по суффиксной ссылке, спуск на то же расстояние. Если после $\log \log |\mathcal{T}|$ операций мы не достигли цели, мы прерываем попытку и ищем ссылку через запрос к DWAT. Этот подход позволяет объединить ограничения: $\mathcal{O}(\log \log n)$ время обновления от DWAT и $\mathcal{O}(|\mathcal{T}|) = \mathcal{O}(\frac{n}{E})$ суммарное время работы одного конвейера, что даёт $\mathcal{O}(n)$ суммарно по всем конвейерам.

С этими оптимизациями, алгоритм FastARR работает с использованием такого же объёма памяти, что и алгоритм ARR; в оценке времени обновления доминирует обновление суффиксного дерева (строка 8), работающее за $\mathcal{O}(\log n)$, включая обновление DWAT; вызов $lazyProcess$ (строка 4) требует только $\mathcal{O}(\log \log n)$ времени на операцию, как показано выше.

Общее время обновлений дерева (включая DWAT) есть $\mathcal{O}(\frac{n}{E} \log n)$; суммируя с ограничением $\mathcal{O}(n)$ для всех конвейеров, получаем результат теоремы. □

Замечание 3.4.1. Наиболее тяжёлая операция в алгоритме FastARR — это обновление суффиксного дерева. Однако, на данный момент не известно алгоритма, который даёт лучшее время обновления в худшем случае и рассчитан на полиномиальный целочисленный алфавит. Например, алгоритм Бреслауера и Итальяно [12] имеет время обновления $\mathcal{O}(\log \log n)$, но только для константных

алфавитов, тогда как алгоритм Копелевица [42] основан на у-быстром боре [58], время обновления которого в худшем случае есть $\mathcal{O}(\log n)$.

3.5 Поиск наибольшего повтора

Простое решение для задачи LRS (мы называем его алгоритм AR) очень похоже на алгоритм ARR; оно даже проще, потому что нам не нужно специальным образом учитывать перекрывающиеся повторы: символ в правом вхождении повтора всегда становится доступен позже, чем соответствующий ему в левом вхождении, вне зависимости от того, перекрывающийся повтор или нет. Достаточно внести следующие изменения в алгоритм ARR: (i) использовать P_b вместо Q_b, (ii) строить суффиксное дерево слева направо, используя алгоритм Укконена [56], усиленный динамическим совершенным хэшированием, вместо алгоритма Вейнера (строка 9; строка 8 больше не нужна), и (iii) переопределить $suff_r^i$ как наибольший суффикс P_r^i, имеющий вхождение в P_b^{i-r}.

Теорема 3.5.1. Существует потоковый алгоритм типа Монте-Карло, решающий задачу LRS с заданной аддитивной погрешностью $E = \mathcal{O}(n^{0.99})$, использующий $\mathcal{O}(\frac{n}{E} + E)$ памяти, работающий за $\mathcal{O}(n)$ времени суммарно и имеющий время обновления $\mathcal{O}(\frac{n}{E})$.

Однако есть проблема с усилением этого алгоритма для получения лучшего времени обновления. А именно, каждая итерация алгоритма Вейнера создаёт ровно одну терминальную и не более одной нетерминальной вершины, в то время как в алгоритме Укконена количество новых вершин на итерации может быть линейно от размера дерева. Следовательно, может оказаться, что некоторые обновления дерева $\mathcal{T}$ нельзя выполнить быстрее, чем за $\Theta(|\mathcal{T}|)$. Чтобы избежать этого, мы работаем со следами Q_b и Q_r, сравнивая обратные строки из «обратных» хэш-букв. Это позволяет нам продолжить использование алгоритма Вейнера для построения дерева и получить аналог теоремы 3.4.2.

Теорема 3.5.2. Существует потоковый алгоритм типа Монте-Карло, решающий задачу LRS с заданной аддитивной погрешностью $E = \mathcal{O}(n^{0.99})$, использующий $\mathcal{O}(\frac{n}{E} + E)$ машинных слов памяти, работающий за $\mathcal{O}(n + \frac{n}{E}\log n)$ времени суммарно и имеющий время обновления $\mathcal{O}(\log n)$.

Определим $pref_r^i$ как наибольший префикс Q_r^i, имеющий вхождение в Q_b^{i-r}, и переопределим pos_r^i, как позицию $pref_r^i$ в текущем дереве $\mathcal{T}(Q_b^i)$. По аналогии с наблюдением 3.4.1, получаем

Наблюдение 3.5.1. Равенство $pref_r^{i-1} = pref_r^{i-b}$ выполняется, если $i + 1 \equiv_b r$.

Как и с обратными повторами, после чтения $S[i]$ мы берём $r = (i+1) \overline{\bmod}\, b$ и вычисляем pos_r^{i-1} из $pos_r^{i-b} = (v, raise)$, поднимаясь по дереву, пока ребро с позицией на $raise$ символов над v не будет найдено. После этого мы вычисляем pos_r^i, используя прямой аналог предложения 3.4.1.

Предложение 3.5.1. *Наибольший собственный суффикс $pref_r^i$ является префиксом $pref_r^{i-1}$.*

Доказательство теоремы 3.5.2. Определим размера блока $b = \lfloor \frac{E}{2} \rfloor$ и будем писать $\mathcal{T}_j$ для суффиксного дерева $\mathcal{T}(Q_b^j)$. Алгоритм FastAR представлен ниже.

Алгоритм 3.3. Алгоритм FastAR (FastAdditiveRepeat), i-я итерация ($i \geq b$)

1: $r = i \bmod b + 1$
2: read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
3: compute $FR = F^R(i-b+1, i)$ from I and $SI[r]$
4: update $pos[r] = pos_r^{i-b}$ to pos_r^{i-1} by walking up
5: compute $pos[r] = pos_r^i$ as longest prefix of $FR{\cdot}pref_r^{i-1}$ that exists in $\mathcal{T}$
6: update *answer* by the value determined by $pos[r]$
7: **if** $r = b$ **then**
8: update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR and building suffix links using BIS
9: $SI[r] = I$

Строка 4 требует $\mathcal{O}(1)$ времени, так как только одна внутренняя вершина может появиться в $\mathcal{T}$ между соответствующими итерациями; строка 8 требует $\mathcal{O}(\log n)$ времени на одной итерации и $\mathcal{O}(\frac{n}{E} \log n)$ времени суммарно (см. алгоритм FastARR). Остаётся объяснить вычисления в строке 5. Как и в теореме 3.4.2, мы смешиваем два подхода для получения хорошего суммарного времени одновременно с хорошим временем обновления.

Один подход заключается в использовании *жёстких обратных ссылок*, которые являются обратными к суффиксным ссылкам (т.е., ведут из вершины u во все вершины вида au, где a это символ). Очевидно, они могут быть добавлены в дерево одновременно с суффиксными ссылками и сохранены в хэш-таблице, аналогичной той, что используется для перехода по символам. Для вычисления pos_r^i, нужно подняться в дереве из позиции pos_r^{i-1} до ближайшего предка, из которого есть жёсткая обратная ссылка по хэш-букве FR, затем перейти по этой ссылке и, возможно, спуститься на сколько-то букв в пределах ребра (однако, количество операций, необходимых для вычисления длины спуска равняется, в худшем случае, числу вершин, пройденных на подъёме). Заметим, что один шаг наверх уменьшает древесную глубину на 1, переход по жёсткой ссылке не может увеличить её больше, чем на 1, а спуск в пределах ребра на неё не влияет. Таким образом, суммарное число шагов вверх при обработке одного следа есть $\mathcal{O}(|\mathcal{T}|)$. Это даёт нам суммарное время работы $\mathcal{O}(n)$ для всех вычислений в строке 5.

Второй подход заключается в одном запросе к BIS, занимающим $\mathcal{O}(\log n)$ времени. Наконец, «смешанное» решение заключается в выполнении $\log n$ шагов поиска в дереве, и, если требуемая позиция не найдена, делается запрос к BIS. Это трюк позволяет получить суммарное время $\mathcal{O}(n)$ и время обновления $\mathcal{O}(\log n)$. □

3.6 Нижние оценки

В этом разделе мы применим принцип Яо (теорема 1.1.1), чтобы доказать нижние оценки для задачи LRS, с заданной длиной входа n и алфавитом Σ; мы используем обозначение $\mathsf{LRS}_{|\Sigma|}[n]$. Для задачи LRRS точно такие же результаты могут быть получены прямолинейной адаптацией теорем 2.2.1–2.2.3 (самый длинный обратный повтор может быть палиндромом), поэтому мы их здесь не приводим.

Сначала докажем что любой алгоритм типа Лас-Вегас, решающий LRS с заданной аддитивной погрешностью, требует как минимум линейной памяти, а значит, не является потоковым. Напомним, что для алгоритмов типа Лас-Вегас класс $\mathcal{A}$ состоит из всех корректных алгоритмов и $c(a, x)$ — это используемая память.

Теорема 3.6.1. Пусть A — рандомизированный онлайн-алгоритм типа Лас-Вегас, решающий задачу $\mathsf{LRS}_{|\Sigma|}[n]$ с аддитивной ошибкой $E \leq 0.49n$ и использующий $s(n)$ бит памяти. Тогда $\mathbf{E}[s(n)] = \Omega(n \log |\Sigma|)$.

Доказательство. Для упрощения вычислений мы дадим доказательство для $\Sigma = \{0, 1\}$. Оно может быть расширено на произвольный алфавит кодированием букв двоичными строками одинаковой длины. Пусть $\mathcal{P}$ — равномерное распределение на множестве всех строк длины n. Рассмотрим произвольный детерминированный алгоритм D, решающий $\mathsf{LRS}_{|\Sigma|}[n]$ с аддитивной погрешностью E. Строку z назовем «хорошей», если D использует на ней не больше $0.004n$ бит памяти, и «плохой» в противном случае. Сначала допустим, что хотя бы половина входов — хорошие. Тогда хорошие входы покрывают как минимум половину, или $2^{0.5n-1}$, возможных префиксов длины $n/2$ входного потока. После прочтения такого префикса хорошего входа, алгоритм D находится в одном из не более чем $2^{0.004n}$ состояний. Тогда мы можем выбрать класс C, состоящий из не менее чем $2^{0.496n-1}$ префиксов, дающих одинаковое состояние D. Из элементарных вычислений следует, что существует не более $n^2 \cdot 2^{0.495n}$ префиксов, которые содержат повтор длины $0.005n$. Более того, любая входная строка xx имеет общую подстроку длины $0.005n$ с не более чем $n^2 \cdot 2^{0.495n}$ строками длины $n/2$. Следовательно, для достаточно больших n класс C содержит две строки x и y, такие что ни одна из них не содержит повтор длины $0.005n$ и xx не имеет общей подстроки длины $0.005n$ с y. Тогда длина наибольшего повтора в xx это $0.5n$, в то время как для yx наибольший повтор по длине меньше $2 \cdot 0.005n = 0.01n$. Так как D находится в одном и том же состоянии после чтения x и y, он выдаёт одинаковый ответ для xx и yx; один из ответов должен быть неверным.

Поскольку алгоритм типа Лас-Вегас не дает неверных ответов, допущение о количестве хороших входов оказалось ложным, и как минимум половина входов плохие. А значит, математическое ожидание памяти, используемой алгоритмом D, не меньше чем $0.004n/2 = \Omega(n \log |\Sigma|)$ бит. Ссылка на теорему 1.1.1 завершает доказательство. □

Для алгоритмов типа Монте-Карло, $\mathcal{A}$ состоит из всех (не обязательно корректных) алгоритмов, и $c(a, x)$ является индикаторной функцией корректности (0 если алгоритм корректно работает на входе, 1 иначе). Сначала покажем, что точный ответ для LRS не может быть найден с сублинейной памятью.

Лемма 3.6.1. Существует константа γ, такая что любой рандомизированный онлайн-алгоритм типа Монте-Карло, решающий задачу $\mathsf{LRS}_\Sigma[n]$ точно с вероятностью $1 - \frac{1}{n}$, использует не меньше $\gamma n \log \min\{|\Sigma|, n\}$ бит памяти.

Доказательство. Мы используем вспомогательную задачу $\mathsf{LCP}_\Sigma[n]$, которая заключается в поиске наибольшего общего префикса левой и правой половин входа (здесь n — четно). Сначала мы докажем, что если A — это алгоритм типа Монте-Карло, решающий $\mathsf{LCP}_\Sigma[n]$ точно и использующий меньше чем $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ бит памяти, то его вероятность ошибки не менее $\frac{1}{n|\Sigma|}$.

По теореме 1.1.1, достаточно построить распределение вероятностей $\mathcal{P}$ над Σ^n, такое что для любого детерминированного алгоритма D, использующего меньше чем $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ бит памяти, математическое ожидание вероятности ошибки на строке, выбранной в соответствии с $\mathcal{P}$, будет $\geq \frac{1}{n|\Sigma|}$. Чтобы сделать это, положим $n' = \frac{n}{2}$. Для любой строки $x \in \Sigma^{n'}$ и любых $k = 1, \ldots, n'$, $c \in \Sigma$ положим $w(x, k, c) = x[1..n']x[1..k{-}1]cx[k{+}1..n']$. Пусть $\mathcal{P}$ — равномерное распределение на множестве строк $w(x, k, c)$.

Выберем произвольное максимальное по включению множество пар строк $(x, x') \in \Sigma^{n'} \times \Sigma^{n'}$, такое что различные пары не имеют общих элементов и алгоритм D находится в одинаковом состоянии после прочтения x и x'. Ввиду максимальности, для каждого состояния алгоритма D не более одной строки останется без пары, то есть всего не более $2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ строк останется без пар. Так как всего имеется $|\Sigma|^{n'} = 2^{n' \log |\Sigma|} \geq 2 \cdot 2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ возможных строк длины n', не меньше половины из них принадлежат парам.

Пусть s — наибольший общий префикс x и x', то есть $x = scv$, $x' = sc'v'$, где $c \neq c'$ — различные буквы. Тогда D возвращает один и тот же ответ на $w(x, |s|, c)$ и $w(x', |s|, c)$, хотя корректный ответ равен $|s|$ во втором случае и не меньше $|s| + 1$ в первом. Аналогично, D ошибается на $w(x, |s|, c')$ или на $w(x', |s|, c')$. Таким образом, вероятность ошибки не меньше $\frac{1}{2n'|\Sigma|} = \frac{1}{n|\Sigma|}$.

Сейчас мы докажем лемму для задачи $\mathsf{LCP}_\Sigma[n]$, используя стандартный для подобных доказательств приём усиления (amplification). Допустим, что у нас есть потоковый алгоритм типа Монте-Карло, который решает $\mathsf{LCP}_\Sigma[n]$ точно с вероятностью ошибки ε и использует $s(n)$ бит памяти. Тогда мы можем запустить k его копий одновременно и вернуть наиболее частый ответ. Новый алгоритм использует $\mathcal{O}(k \cdot s(n))$ бит памяти, и его вероятность ошибки ε_k удовлетворяет неравенству

$$\varepsilon_k \leq \sum_{2i<k} \binom{k}{i} (1-\varepsilon)^i \varepsilon^{k-i} \leq 2^k \cdot \varepsilon^{k/2} = (4\varepsilon)^{k/2}.$$

Пусть $\kappa = \frac{1}{6}\frac{\log(4/n)}{\log(1/(n|\Sigma|))}$, то есть

$$\kappa = \frac{1}{6}\frac{1-o(1)}{1+\log|\Sigma|/\log n} = \Theta\left(\frac{\log n}{\log n + \log|\Sigma|}\right) = \gamma \cdot \frac{1}{\log|\Sigma|}\log\min\{|\Sigma|, n\}$$

для некоторой константы γ. Предположим, что A использует меньше $\kappa \cdot n\log|\Sigma| = \gamma \cdot n\log\min\{|\Sigma|, n\}$ бит памяти. Тогда запуск $\left\lfloor\frac{1}{2\kappa}\right\rfloor \geq \frac{3}{4}\frac{1}{2\kappa}$ (так как $\kappa < \frac{1}{6}$) копий A параллельно требует меньше чем $\lfloor\frac{n}{2}\log|\Sigma|\rfloor$ бит памяти. Но в этом случае вероятность ошибки нового алгоритма ограничена сверху $\left(\frac{4}{n}\right)^{3/16\kappa} = \left(\frac{1}{n|\Sigma|}\right)^{18/16} \leq \frac{1}{n|\Sigma|}$, что как мы уже показали, невозможно.

Нижняя оценка для LCP может быть преобразована в нижнюю оценку для точного решения LRS модификацией входа таким образом, что наибольший повтор становится общим префиксом всей строки и её правой половины. Пусть $x = x[1..n]$ — это вход для $\mathsf{LCP}_{|\Sigma|}[n]$ с ответом k. Определим $w(x) = 0^n 1 x[1..\frac{n}{2}] 0^n 1 x[\frac{n}{2}+1..n]$, где $0, 1 \notin \Sigma$; очевидно, $w(x)$ содержит повтор длины $n+k+1$. С другой стороны, любая повторяющаяся подстрока в $w(x)$ длины $\geq n+1$ должна содержать подстроку $0^{n/2}1$, которая имеет только два вхождения в w. Таким образом, мы свели решение $\mathsf{LCP}_{|\Sigma|}[n]$ к решению $\mathsf{LRS}_{|\Sigma|}[3n+2]$. Мы уже знаем, что решение $\mathsf{LCP}[n]$ с вероятностью $1-\frac{1}{n}$ требует $\gamma \cdot n\log\min\{|\Sigma|, n\}$ бит памяти, а значит решение $LRS_{|\Sigma|}[3n+2]$ с вероятностью $1-\frac{1}{3n+2} \geq 1-\frac{1}{n}$ требует $\gamma \cdot n\log\min\{|\Sigma|, n\} \geq \gamma' \cdot (3n+2)\log\min\{|\Sigma|, 3n+2\}$ бит памяти. Сведение требует $\mathcal{O}(\log n)$ дополнительных бит памяти для подсчёта до n, но для больших n это сильно меньше нижней оценки, если мы выбираем $\gamma' < \frac{\gamma}{4}$. □

Для алгоритмов типа Монте-Карло с аддитивной погрешностью мы сначала докажем простую техническую лемму, затем вспомогательную грубую оценку и, наконец, точную оценку, используя сведение к точным алгоритмам типа Монте-Карло.

Лемма 3.6.2. Пусть x — случайная строка, равновероятно выбранная среди всех строк длины n над алфавитом $\{0, 1\}$. Тогда с вероятностью $1-\frac{1}{n}$ в x нет повторов длины $3\log n$.

Доказательство. Две случайно выбранные подстроки длины k в x совпадают с вероятностью 2^{-k}. Следовательно, матожидание числа повторов длины k в x не превосходит $n^2 \cdot 2^{-k}$. При $k = 3\log n$ это матожидание, а значит, и вероятность наличия хотя бы одного повтора, не превосходит $\frac{1}{n}$. □

Лемма 3.6.3. Любой рандомизированный онлайн-алгоритм типа Монте-Карло, решающий задачу $\mathsf{LRS}_{|\Sigma|}[n]$ с аддитивной погрешностью $E \leq 0.49n$ и вероятностью ошибки $\frac{1}{n}$, использует $\Omega(\log n)$ бит памяти.

Доказательство. Пусть $\Sigma = \{0, 1\}$. Рассмотрим равномерное распределение $\mathcal{P}$ над любым множеством P из $2^{n/2+1}$ строк чётной длины n со следующим

свойством: каждая строка $x \in \Sigma^{n/2}$ встречается дважды как левая половина строки из P, при этом одна из этих строк это xx а вторая это xy где y — случайная строка, выбранная независимо из всех строк длины $n/2$. Тогда половина строк в P имеет повтор длины $n/2$, а строки из второй половины полностью случайны, и, соответственно, с вероятностью $1 - \frac{1}{n}$ имеют самый длинный повтор длины $\Theta(\log n)$ по лемме 3.6.2. Следовательно, если алгоритм хранит любой скетч из $s(n) = o(\log n)$ бит информации о строке x, он может различить x и y во второй половине со слишком маленькой вероятностью: y имеет совпадающий с x скетч с вероятностью $2^{-s(n)} > \frac{1}{n}$. □

Теорема 3.6.2 (Аддитивное приближение алгоритмами типа Монте-Карло)**.** Любой рандомизированный онлайн-алгоритм типа Монте-Карло, решающий задачу $\mathsf{LRS}_{|\Sigma|}[n]$ с аддитивной погрешностью $E \le 0.49n$, с вероятностью $1 - \frac{1}{n}$ использует $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ бит памяти.

Доказательство. Пусть $\sigma = \min\{|\Sigma|, \frac{n}{E}\}$. Так как ограничение $\Omega(\frac{n}{E} \log \sigma)$ превращается в $\Omega(\log n)$ при $E = \Omega(\frac{n \log \sigma}{\log n})$, мы далее полагаем, что $E = o(\frac{n \log \sigma}{\log n})$.

Допустим, что существует потоковый алгоритм типа Монте-Карло A, решающий $\mathsf{LRS}_{|\Sigma|}[n]$ с аддитивной погрешностью E с вероятностью $1 - \frac{1}{n}$, используя $o(\frac{n}{E} \log \sigma)$ бит памяти. Пусть $n' = \lfloor \frac{n-E}{E+1} \rfloor$. Мы определим новый потоковый алгоритм типа Монте-Карло A', который обрабатывает строку $x[1..n']$ следующим образом: запускает алгоритм A на $x' = 0^E x[1] 0^E x[2] \cdots 0^E x[n'] 0^E$, используя $\log E \le \log n$ дополнительных бит памяти для счёта до E, получает ответ R, и возвращает число $\lfloor \frac{R}{E+1} \rfloor$. Если наибольший повтор в x имеет длину r, то наибольший повтор в x' имеет длину $(r+1)E + r$ (каждое вхождение состоит из r букв x и $r+1$ блока из нулей). Так как алгоритм A имеет аддитивную погрешность E, его ответ удовлетворяет неравенству $r(E+1) \le R \le (r+1)E + r$, и значит A' должен вернуть r. Следовательно A' решает $\mathsf{LRS}_{|\Sigma|}[n']$ точно с вероятностью $1 - \frac{1}{n} \ge 1 - \frac{1}{n'}$, используя $o(n' \log \sigma) + \log n$ бит памяти. По наложенному на E ограничению, это число является $o(n' \log \sigma)$, что противоречит лемме 3.6.1, которая показывает нижнюю оценку на использованную память $\Omega(n' \log \sigma)$ для точного решения LRS. □

Следствие 3.6.1. *Представленные алгоритмы AR, FastAR, ARR, FastARR используют оптимальный объём памяти в случае* $E = \mathcal{O}(\sqrt{n})$ *и* $|\Sigma| = \Omega(n^{0.01})$.

Глава 4

Максимальные периодические подстроки в потоках

4.1 Введение

Строка является периодической, если её экспонента, являющаяся отношением длины к минимальному периоду, не меньше 2. Максимальные периодические подстроки, или раны (см. определение в разделе 1.1.1), хорошо изучены как с алгоритмической, так и с комбинаторной точки зрения. Колпаков и Кучеров [41] показали, что строка длины n имеет $\mathcal{O}(n)$ ранов и выдвинули гипотезу, что это число меньше чем n. После нескольких промежуточных результатов, эта гипотеза была доказано Баннаи и др. [7]. На данный момент известно, что наибольшее число ранов в строке длины n асимптотически находится между $0.94n$ и $0.95n$, см. [25, 35]. Первый алгоритм для поиска ранов был предложен в [47]. В [41] представлен линейный алгоритм решения задачи Runs о поиске всех ранов в строке; этот алгоритм работает с целочисленным алфавитом полиномиального размера и основан на так называемом разложении Лемпеля–Зива (другой подход, основанный на словах Линдона, был позднее представлен в [7]). Однако над произвольным алфавитом размера σ разложение Лемпеля–Зива вычисляется за $\Theta(n \log \sigma)$ [44], в то время как все раны могут быть найдены быстрее, как было показано в серии статей [30, 45], завершившейся работой [20], в которой представлен алгоритм, работающий за время $\mathcal{O}(n \cdot \alpha(n))$, где α — обратная функции Аккермана.

Результатов о поиске ранов с ограниченной памятью почти нет; в единственной статье [29] представлены два алгоритма, находящие все раны с использованием маленького объёма памяти в дополнение к входной строке. Один из алгоритмов работает над произвольным алфавитом за $\mathcal{O}(n \log n)$ времени и использует $\mathcal{O}(1)$ дополнительной памяти, а другой работает над константным алфавитом за $\mathcal{O}(n)$ времени и использует $o(n)$ памяти. В этой главе мы

рассмотрим потоковую версию задачи.

Мы покажем, что ни один потоковый алгоритм не может решить с высокой вероятностью задачу точного подсчета всех ранов в потоке (то есть алгоритм с заметной вероятностью пропустит существующий ран или найдет несуществующий). Ввиду этого результата необходимо сформулировать приближённый вариант задачи Runs. Эту задачу мы определяем следующим образом.

Задача approxRuns: для заданной входной строки S и параметра $\varepsilon = \varepsilon(n) \in (0, \frac{1}{2}]$ вернуть множество подстрок из S, таких что

(i) для каждого рана r в S, имеющего экспоненту $\beta \geq 2 + \varepsilon$, возвращается ровно одна периодическая подстрока в r с тем же периодом, что и r, и экспонентой не меньше $\beta - \varepsilon$;

(ii) для каждого рана r в S, имеющего экспоненту $\beta < 2 + \varepsilon$, возвращается ноль или одна периодическая подстрока в r с тем же периодом, что и r.

Основная задача данной главы состоит в доказательстве следующего результата.

Теорема 4.1.1. Существует потоковый алгоритм типа Монте-Карло, решающий задачу approxRuns, использующий $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$ машинных слов памяти и имеющий время обновления[1] $\mathcal{O}(\log n)$.

В разделе 4.2 мы напомним обозначения, некоторые базовые результаты и докажем несуществование потокового алгоритма, который вычисляет количество ранов точно. В разделе 4.3 мы введём основные инструменты и конструкции; алгоритм, анонсированный в теореме 4.1.1, представлен в секции 4.4.

4.2 Определения

Строка *примитивна*, если она не является конкатенацией двух или более копий более короткой строки. Мы называем период p строки S *примитивным*, если строка $S[1..p]$ примитивна, и *коротким*, если $p \leq |S|/2$; если $p = |S|/2$, то S — *квадрат*. Следующая лемма — это классическая теорема Файна-Вильфа [24], записанная в подходящей форме (второе утверждение напрямую следует из первого).

Лемма 4.2.1. 1) Строка с примитивными периодами p и q имеет длину меньше чем $p + q - \gcd(p, q)$.
2) Все короткие периоды строки кратны минимальному периоду этой строки.

Периодическая строка с периодом p — это строка, минимальный период которой равен p и является коротким. Следовательно, S — периодическая

[1] Мы расширили множество элементарных операций операциями со словарём (вставка, удаление, поиск). Оптимальный выбор словаря зависит от ε и обсуждается в замечании 4.4.1.

тогда и только тогда, когда $\exp(S) \geq 2$; периодическая строка с экспонентой 2 является квадратом (с примитивным периодом). Периодическая подстрока $S[i..j]$ с периодом p называется *раном* (в S), если подстроки $S[i{-}1..j]$ и $S[i..j{+}1]$ не имеют периода p или не существуют. По лемме 4.2.1, это эквивалентно утверждению о том что $S[i{-}1..j]$ и $S[i..j{+}1]$ не являются периодическими. Для фиксированного S, обозначим периодическую подстроку $S[l..r]$ с периодом p тройкой (l, p, r). Из определений сразу следует

Наблюдение 4.2.1. Если (l, p, r) и (h, p, i) — две периодические подстроки и $l < h \leq h + p - 1 \leq r < i$, то подстрока (l, p, i) является периодической.

Следующая теорема демонстрирует, что не существует потокового алгоритма, который корректно идентифицирует, с высокой вероятностью, все квадраты, а значит и все раны, в строке. Пусть $\mathsf{midSquare}(\Sigma, n)$ это задача поиска длиннейшего «центрального» квадрата, имеющего вид $S[\frac{n}{2}{-}i{+}1..\frac{n}{2}{+}i]$, во входном потоке чётной длины n над алфавитом Σ.

Теорема 4.2.1. Существует константа γ, такая что любой алгоритм, точно решающий задачу $\mathsf{midSquare}(\Sigma, n)$ с вероятностью не меньше $1 - \frac{1}{n}$, использует не меньше $\gamma n \log \sigma$ бит памяти.

Доказательство. Мы используем схему как в лемме 3.6.1. Сначала мы докажем, что если потоковый алгоритм типа Монте-Карло решает midSquare точно, используя меньше чем $\lfloor \frac{n}{2} \log \sigma \rfloor$ бит памяти, то вероятность ошибки у него не меньше $\frac{1}{n\sigma}$. В соответствии с принципом Яо (теорема 1.1.1), для этого достаточно построить распределение вероятностей $\mathcal{Q}$ над Σ^n, такое что любой детерминированный алгоритм D, использующий меньше чем $\lfloor \frac{n}{2} \log \sigma \rfloor$ бит памяти, имеет вероятность ошибки не меньше $\frac{1}{n\sigma}$ на строке, выбранной в соответствии с $\mathcal{Q}$.

Пусть $a \in \Sigma$, $n' = n/2$. Для произвольного $x \in \Sigma^{n'}$, $c \in \Sigma$ и $k \in \{1, \ldots, n'\}$ обозначим $w(x, k, c) = x[1..n']cx[n'{-}k{+}2..n']a^{n'-k}$. В качестве $\mathcal{Q}$ возьмём равномерное распределение над всеми строками $w(x, k, c)$.

Выберем произвольное максимальное множество не пересекающихся пар (x, x') строк из $\Sigma^{n'}$, таких что D находится в одинаковом состоянии после прочтения x или x'. Пусть $x = vcs$, $x' = v'c's$, где $v, v', s \in \Sigma^*$, $c, c' \in \Sigma$, и $c \neq c'$. Тогда D возвращает одинаковый ответ для $w(x, |s|{+}1, c)$ и $w(x', |s|{+}1, c)$, потому что правые половины строк совпадают. Однако правильные ответы различаются: $w(x, |s|{+}1, c)$ имеет центральный квадрат с периодом $|s|$, а $w(x', |s|{+}1, c)$ не имеет такого квадрата; и если одна из строк имеет более длинный центральный квадрат, то другая его не имеет, потому что буква в правой половине не может совпасть одновременно с c из x и c' из x'. Следовательно D ошибается хотя бы на одном из рассматриваемых входов; аналогично, он ошибается на $w(x, |s|{+}1, c')$ или на $w(x', |s|{+}1, c')$.

Так как память D имеет не более $\sigma^{n'}/2$ возможных состояний, и не более одной строки на каждое состояние осталось вне пары, количество пар не меньше

$\sigma^{n'}/4$. Значит количество ошибок не меньше $\sigma^{n'}/2$, при общем количестве строк равном $\sigma^{n'} \cdot n' \cdot \sigma$. Из чего следует, что вероятность ошибки не меньше $\frac{1}{n\sigma}$.

Теперь допустим, что некоторый потоковый алгоритм A типа Монте-Карло решает midSquare точно, с вероятностью ошибки $\varepsilon \le \frac{1}{n}$, и использует $s(n)$ бит памяти. Тогда мы можем запустить k его копий одновременно и вернуть наиболее частый ответ. Новый алгоритм A_k использует $\mathcal{O}(k \cdot s(n))$ бит памяти и имеет вероятность ошибки ε_k, удовлетворяющую неравенству

$$\varepsilon_k \le \sum_{2i<k} \binom{k}{i} (1-\varepsilon)^i \varepsilon^{k-i} \le 2^k \cdot \varepsilon^{k/2} = (4\varepsilon)^{k/2}.$$

Вспомним, что $\sigma = \mathcal{O}(n^p)$ для некоторой константы p. Обозначим $k = 2p+3$ и возьмём любое положительное число $\gamma \le \frac{1}{2k}$. Если $s(n) < \gamma n \log \sigma$, то алгоритм A_k использует меньше чем $\lfloor \frac{n}{2} \log \sigma \rfloor$ бит памяти и для достаточно большого n имеет вероятность ошибки меньше чем $\frac{1}{n\sigma}$; как показано выше, это невозможно, следовательно теорема выполняется для выбранного значения γ. □

4.3 Инструменты

4.3.1 Хэши, фреймы, чекпойнты

Наш алгоритм для задачи approxRuns использует хэши Карпа–Рабина (см. раздел 1.1.4). Пусть p — фиксированное простое число из отрезка $[n^4, n^5]$, и r — фиксированное число, выбранное случайно и равновероятно из $\{1, \ldots, p-1\}$. Хэш строки S определяется как $\phi(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^i\right) \bmod p$. Вероятность совпадения хэшей для двух различных строк длины m не превосходит m/p. Наш алгоритм сравнивает хэши подстрок, имеющих одинаковую длину вида 2^j. Вероятность того, что у пары таких строк совпадут хэши, меньше чем n^3/p и следовательно, меньше допустимой вероятности ошибки для алгоритма типа Монте-Карло. Все дальнейшие рассуждения подразумевают, что совпадений хэшей не происходит. Напомним, что *фрейм* строки A — это кортеж $(|A|, \phi(A), r^{|A|} \bmod p, r^{-|A|} \bmod p)$.

Все определения ниже относятся к входному потоку S. Напомним, что i-я итерация потокового алгоритма, обрабатывающего S, начинается с чтения $S[i]$ и заканчивается перед чтением $S[i{+}1]$. Мы пишем $I(i)$ для обозначения фрейма подстроки $S[1..i{-}1]$. Из леммы 1.1.1 следует, что $I(i{+}1)$ может быть вычислен за время $\mathcal{O}(1)$ по $I(i)$ и $S[i]$.

Вся информация, хранимая алгоритмом, привязана к *чекпойнтам*, которые образуют подмножество всех позиций. Каждая позиция k становится чекпойнтом на k-й итерации и «живёт» в течение $\mathsf{ttl}(k)$ итераций, где функция времени жизни (см. раздел 2.3.2) определена равенством $\mathsf{ttl}(k) = 2^{t_\varepsilon + 2 + \beta(k)}$, где $t_\varepsilon = \lceil \log \frac{2}{\varepsilon} \rceil$ и $\beta(k)$ — это максимальная степень двойки, делящая k. Если $k + \mathsf{ttl}(k) = i$, то

в начале i-й итерации k «умирает» (теряет статус чекпойнта) и вся связанная информация удаляется.

Количество чекпойнтов на i-й итерации есть $\mathcal{O}(\frac{\log i}{\varepsilon})$ (лемма 2.3.5), на каждой итерации умирает не более одного чекпойнта (лемма 2.3.8).

4.3.2 Видимые периодические строки

Наш алгоритм работает с подстроками длины 2^j, $j = 0, \ldots, \lfloor \log n \rfloor$. Такие подстроки, имеющие вхождение в чекпойнте, назовём *j-блоками*. Дадим теперь ключевое определение. Пусть (h, p, i) — периодическая строка, $j = \lfloor \log p \rfloor$, $f = i - p - 2^j + 1$. Периодическая строка (h, p, i) называется *видимой*, если $\mathsf{ttl}(h), \mathsf{ttl}(f) \geq 2^{j+2}$ и $f - h \leq 2^j$ (см. рис. 4.1). Такая периодическая строка покрыта парами вхождений двух перекрывающихся (или касающихся) j-блоков; мы покажем, что видимые периодические строки могут быть обнаружены на i-й итерации благодаря тому, что h и f в этот момент являются чекпойнтами.

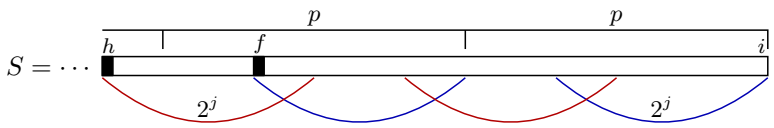

Рис. 4.1: Видимая периодическая строка покрыта перекрывающимися вхождениями двух j-блоков.

Лемма 4.3.1. Каждая периодическая строка (l, p, r) с экспонентой $\geq 2 + \varepsilon$ содержит видимую периодическую подстроку с периодом p.

Доказательство. Пусть $j = \lfloor \log p \rfloor$, $z = 2^{\max\{0, j - t_\varepsilon\}}$. Число k делится на z тогда и только тогда, когда $\beta(k) \geq j - t_\varepsilon$ или, что то же самое, $\mathsf{ttl}(k) \geq 2^{j+2}$. Пусть $i = r - (r - 2^j - p + 1) \bmod z$, $h = i - 2p + 1 - (i - 2p + 1) \bmod z$. Рассмотрим подстроку $S[h..i]$. Её длина $\geq 2p$, и $r - z + 1 \leq i \leq r$. Более того,

$$\begin{aligned} h &\geq (r - z + 1) - 2p + 1 - (z - 1) = r - 2p - 2z + 3 \\ &\geq r - 2p - 2^{j+1-t_\varepsilon} + 3 \geq r - 2p - \lceil \varepsilon p \rceil + 1 = l. \end{aligned}$$

Таким образом, $S[h..i]$ является подстрокой $S[l..r]$, имеет период p и длину $\geq 2p$; так как p — минимальный период $S[l..r]$, он является примитивным. Следовательно, (h, p, i) — периодическая подстрока по лемме 4.2.1(2). Далее, из определения i и h следует, что z делит $f = i - p - 2^j + 1$ и h, а значит, $\mathsf{ttl}(f), \mathsf{ttl}(h) \geq 2^{j+2}$. Наконец, $f - h = p - 2^j + (i - 2p + 1) \bmod z$ — это минимальное число, кратное z и превосходящее $p - 2^j < 2^j$, откуда $f - h \leq 2^j$. Таким образом, (h, p, i) — видимая периодическая подстрока по определению. □

Лемма 4.3.2. Пусть (h, p, i) — видимая периодическая подстрока и $(h+z, p, i+z)$ — периодическая подстрока, $j = \lfloor \log p \rfloor$, $z = 2^{\max(0, j - t_\varepsilon)}$. Тогда

(i) подстрока $(h{+}z, p, i{+}z)$ — видимая, (ii) подстрока $(h, p, i{+}z)$ — периодическая и (iii) утверждения (i), (ii) остаются верными при замене $(h{+}z, p, i{+}z)$ на $(h{-}z, p, i{-}z)$.

Доказательство. Заметим, что h и $f = i{-}p{-}2^j{+}1$ делятся на z по определению видимой периодической подстроки; то же верно для $h + z$, $f + z$, из чего следует что $\mathsf{ttl}(h{+}z), \mathsf{ttl}(f{+}z) \geq 2^{j+2}$. Так как $(f + z) - (h + z) = f - h \leq 2^j$, получаем (i) по определению. Из $z < p$ получаем $i - (h{+}z) \geq p$; тогда (ii) следует из замечания 4.2.1. Аналогичные рассуждения работают для (iii). □

Под *списком отслеживания* мы понимаем структуру данных W, содержащую описанный далее список периодических подстрок. Изначально W пуст. На i-й итерации он обновляется следующим образом. Для всех видимых периодических подстрок (h, p, i) в потоке, если W содержит периодическую подстроку (l, p, r) с таким же периодом, эта периодическая строка обновляется до (l, p, i); иначе, (h, p, i) добавляется в W. После этого удаляются все периодические подстроки (l, p, r), такие что $r + z = i$, где z определяется как в лемме 4.3.2.

Лемма 4.3.3. Любой потоковый алгоритм, который
- находит все видимые периодические подстроки вида (h, p, i) в течение i-й итерации;
- поддерживает список отслеживания;
- выдаёт периодические подстроки сразу после удаления из списка отслеживания,

решает задачу approxRuns.

Доказательство. Пусть (l, p, r) — ран в S с экспонентой $\alpha \geq 2 + \varepsilon$, и пусть (h, p, i) — самая левая видимая периодическая подстрока с периодом p внутри $S[l..r]$ (такая подстрока существует по лемме 4.3.1). Определим z как ранее; тогда $h - z < l$: иначе периодическая подстрока $(h{-}z, p, i{-}z)$ будет видимой по лемме 4.3.2, что противоречит выбору (h, p, i). Пусть k таково, что $i + kz \leq r < i + (k{+}1)z$. Тогда $(h + z, p, i + z), \ldots, (h + kz, p, i + kz)$ — видимые периодические подстроки по лемме 4.3.2; более того, внутри $S[l..r]$ нет других видимых периодических подстрок с периодом p, потому что необходимо прибавить хотя бы z, чтобы получить позицию с необходимым ttl. Рассмотрим i-ю итерацию. Алгоритм обнаруживает (h, p, i) и смотрит в W. Если W содержит периодическую подстроку с периодом p, эта периодическая подстрока была добавлена или обновлена не более z итераций назад; следовательно, S содержит видимую периодическую подстроку (h', p, i'), где $i - z \leq i' < i$. Эта периодическая подстрока не является подстрокой $S[l..r]$ но пересекается с ней на не менее чем $2p - z > p$ символов. Тогда подстрока (h', p, r) — периодическая по замечанию 4.2.1; эта периодическая подстрока полностью содержит ран с тем же периодом, что противоречит определению. Значит, список отслеживания не содержит периодической подстроки с периодом p и алгоритм добавляет (h, p, i) в него.

Теперь заметим, что периодическая подстрока с периодом p в W будет обновляться на итерациях $i+z, \ldots, i+kz$, и будет удалена на итерации $i+(k+1)z$. Таким образом, алгоритм выдаст периодическую подстроку $(h, p, i+kz)$, которая короче подстроки (l, p, r) на не более чем $2z-2 \leq \varepsilon p$ символов и, следовательно, имеет экспоненту не меньше $\alpha - \varepsilon$.

Ран экспоненты меньшей чем $2+\varepsilon$ может не содержать видимых периодических подстрок внутри; в остальном, вышеприведенное рассуждение работает и для таких ранов. □

Лемма 4.3.3 сводит теорему 4.1.1 к построению алгоритма, который находит все видимые периодические подстроки сразу после их вхождения, поддерживает список отслеживания, и удовлетворяет требуемым ограничениям по времени работы и используемой памяти.

Замечание 4.3.1. Алгоритм удовлетворяющий условиям леммы 4.3.3, находит все раны с периодами $p \leq 4\lceil \log 1/\varepsilon \rceil$ точно (если квадрат такого периода p является суффиксом строки $S[1..i]$, то все позиции в нём — чекпойнты, а значит, начало и конец периодичности будут найдены точно).

4.3.3 Свежие и чёрствые вхождения

Нас интересуют недавние вхождения блоков, и мы определяем два типа таких вхождений. На i-й итерации, вхождение j-блока T в позиции h строки $S[1..i]$ является *свежим*, если $h > i - 2^{j+1} + 1$ и *чёрствым*, если $i - 3 \cdot 2^j + 1 < h \leq i - 2^{j+1} + 1$. Это определение можно переформулировать так: вхождение T свежее тогда и только тогда, когда оно было прочитано менее $|T|$ итераций назад, и чёрствое тогда и только тогда, когда оно было свежим $|T|$ итераций назад. *Обычное* вхождение — это вхождение, не являющееся свежим (чёрствые вхождения являются обычными). Следующее наблюдение, очевидное из рис. 4.1, поясняет связь свежих вхождений и видимых периодических подстрок.

Наблюдение 4.3.1. Если (h, p, i) — это видимая периодическая подстрока, $j = \lfloor \log p \rfloor$, тогда на i-й итерации (i) суффикс T длины 2^j строки $S[1..i]$ является j-блоком, имеющим вхождение в чекпойнте $f = i - p - 2^j + 1$ и (ii) подстрока U длины 2^j в позиции $h+p$ является свежим вхождением j-блока, имеющего вхождение в чекпойнте h.

Лемма 4.3.4. 1) Любые два свежих вхождения T пересекаются.
2) Если T имеет хотя бы три свежих вхождения, все позиции свежих вхождений образуют арифметическую прогрессию с разностью, равной минимальному периоду T.
3) Аналоги 1 и 2 выполняются для чёрствых вхождений.

Доказательство. Так как отрезок $[i-2^{j+1}+2..i]$ содержит менее $2|T|$ позиций, 1) очевидно. Далее заметим, что два из любых трёх свежих вхождений T

пересекаются не меньше чем на $|T|/2$ символов, а следовательно, T имеет короткий период. Тогда 2) следует из леммы 4.2.1, и разность прогрессии равна минимальному периоду T. Из выполнения свойств 1) и 2) на i-й итерации следует свойство 3) на $(i+|T|)$-й итерации. □

Мы говорим, что каждый j-блок имеет *серию свежих вхождений* и *серию чёрствых вхождений* (каждая из них может быть пустой). По лемме 4.3.4, такие серии можно хранить, используя $\mathcal{O}(1)$ памяти.

Если вхождение на текущей итерации перестаёт быть свежим (или чёрствым), мы говорим о нём как об *истёкшем* свежем (соотв., чёрством) вхождении.

4.3.4 Структуры данных

Опишем структуры данных, которые мы используем. Для каждого j-блока T бы поддерживаем базовую структуру B_T, называемую *группой* и состоящую из
- фрейма строки T (мы храним только хэш, остальные элементы одинаковы для всех j-блоков);
- двусвязных списков $flist$ и $rlist$ всех чекпойнтов, в возрастающем порядке, являющихся позициями, соответственно, свежих и обычных вхождений T;
- серий $fseries$ и $sseries$ свежих и чёрствых вхождений T.

В каждом узле списков $flist$ и $rlist$ мы храним, кроме чекпойнта и указателей на следующий и предыдущий узел, два вспомогательных числа: период ($period$) и расширение ($extension$). Эти числа для чекпойнта k в группе B_T вычисляются на $(k+|T|)$-й итерации после добавления свежих вхождений T в B_T. Мы определяем $period$ как расстояние между соседними вхождениями в серии $B_T.fseries$; если серия — одноэлементная, положим $(period, extension) = (0, 0)$. Если $period = p > 0$, то у строки $S[1..k+|T|]$ некоторый суффикс имеет период p. Если список отслеживания содержит повтор с таким периодом, мы определяем $extension$ как его позицию; иначе, $extension$ определяется как позиция первого свежего вхождения T.

Замечание 4.3.2. Группа может рассматриваться как структура константного размера (фрейм, две серии, указатели на начала и концы списков) и множество узлов константного размера (позиция, указатели на следующий и предыдущий элементы списка, период, расширение). Мы можем хранить все группы в массиве с ячейками константного размера, в конце которого находится стек пустых ячеек. Это позволяет создавать новую группу/узел и удалять существующую группу/узел за $\mathcal{O}(1)$ операций. Размер массива пропорционален сумме количества групп и количества вхождений j-блоков; каждое из слагаемых есть $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$.

Теперь обсудим, как хранить и обновлять $fseries$ и $sseries$. Для каждой серии мы храним кортеж $(first, period, last, frame1, frame2)$, состоящий из позиций первого и последнего вхождений, расстояния между двумя последовательными вхождениями («период») и фреймов для позиций первого и второго

вхождения. Пустая и одноэлементная серия хранятся как (0) и $(first, 0, frame1)$ соответственно. Серии j-блока T должны обновляться на i-й итерации, если появляется новое свежее вхождение T, либо если свежее/чёрствое вхождение становится истёкшим.

Лемма 4.3.5. Каждое обновление $fseries$ и $sseries$ требует $\mathcal{O}(1)$ времени.

Доказательство. Сначала рассмотрим удаление. Вхождение, которое нужно удалить, всегда первое в серии. Для одноэлементной серии удаление тривиально. Для двухэлементной серии достаточно установить $first$ в $(first + period)$ и $frame1$ в $frame2$. Если серия длиннее, нам также нужно вычислить значение $frame2$ ($period$ и $last$ не изменяются). По лемме 1.1.1, мы вычисляем, за $\mathcal{O}(1)$ времени, фрейм строки $A = S[first..first{+}period - 1] = S[first{+}period..first{+}2{\cdot}period - 1]$ из $frame1$ и $frame2$; затем, опять используя лемму 1.1.1, вычисляем новое значение $frame2$ как $I(first{+}2{\cdot}period)$ из $frame2$ и фрейма строки A.

Теперь рассмотрим добавление. Новое вхождение всегда является последним, поэтому если серия содержит ≥ 2 элементов, мы просто присваиваем новое значение $last$. Для более коротких серий нам также нужно знать фрейм позиции добавляемого вхождения. Если мы добавляем в $sseries$, нужный нам фрейм — это $frame1$, только что удалённый из $fseries$ (новое чёрствое вхождение — это истёкшее свежее). Если мы добавляем в $fseries$, то новое вхождение T является суффиксом $S[1..i]$, а значит, нужный фрейм может быть вычислен по лемме 1.1.1 из фреймов строк $S[1..i]$ и T. Таким образом, общее количество операций ограничено константой. □

Для навигации мы используем шесть словарей, описанных в следующей таблице. Значения в первых пяти словарях хранятся как указатели.

Id	Ключ	Значение
H_1	j, хэш F	группа j-блока с хэшем F
H_2	j, чекпойнт k	группа j-блока, имеющего вхождение в k
H_3	j, позиция k	группа j-блока, имеющего первое свежее вхождение в k
H_4	j, позиция k	группа j-блока, имеющего первое чёрствое вхождение в k
HH	j, чекпойнт k	узел для k в группе j-блока, имеющего вхождение в k
HC	чекпойнт k	фрейм $I(k)$

Мы также храним список отслеживания W в виде двусвязного списка, упорядоченного по периоду периодических подстрок.

4.4 Алгоритм

Каждая итерации начинается с чтения нового символа $S[i]$ из потока и вычисления фрейма $I(i+1)$; новый фрейм сохраняется в переменной I (старое значение I кладётся в HC). Последующие операции группируются в четыре стадии:

- *Удаление чекпойнта.* Здесь мы удаляем все данные про чекпойнт, который умирает на этой итерации. (По лемме 2.3.8, удаляется не более одного чекпойнта.)

- *Обновление групп.* Здесь мы добавляем j-блоки вида $S[h..i]$ в существующие группы или создаём новые группы для них. Также мы переносим чекпойнты истёкших свежих вхождений из $flist$ в $rlist$ и удаляем истёкшие чёрствые вхождения из $sseries$.

- *Обнаружение периодических подстрок.* Здесь мы находим видимые периодические подстроки вида (h, p, i). Параллельно с этим, завершается обновление существующих групп: добавление свежих вхождений в $fseries$ и перенос истёкших свежих вхождений в $sseries$.

- *Обновление списка отслеживания.* Здесь мы добавляем найденные видимые периодические подстроки в список отслеживания, а также удаляем из списка и выводим периодические подстроки, которые гарантированно заканчиваются раньше позиции i.

Полностью i-я итерация алгоритма имеет следующий вид:

Алгоритм 4.1. Алгоритм R, i-я итерация

read $S[i]$; $HC(i) \leftarrow I$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
Удаление чекпойнтов ▷ алгоритм 4.2
Обновление групп ▷ алгоритм 4.3
Обнаружение периодических подстрок ▷ алгоритмы 4.4, 4.5
Обновление списка отслеживания ▷ алгоритм 4.6

4.4.1 Удаление чекпойнтов

Мы обрабатываем все j-блоки в позиции чекпойнта, удаляя чекпойнт из групп и словарей; группы, оставшиеся без чекпойнтов, также удаляются.

Алгоритм 4.2. Алгоритм R, i-я итерация: удаление чекпойнтов

compute $\mathsf{ttl}(i)$; $h \leftarrow i - \mathsf{ttl}(i)$
if $h \geq 1$ **then**
    delete $HC(h)$
    for $\{j \leftarrow 0;\ h + 2^j - 1 < i\}$ **do**
        $B \leftarrow H_2(j, h)$; delete $H_2(j, h)$
        $node \leftarrow HH(j, h)$; delete $HH(j, h)$
        delete $node$ from $B.flist$ or $B.rlist$
        if $B.flist$ and $B.rlist$ are empty **then**
            delete B from H_1, H_3, H_4 ▷ необходимые для удаления ключи хранятся в B
            delete group B

Лемма 4.4.1. Алгоритм 4.2 работает за $\mathcal{O}(\log i)$.

Доказательство. Цикл в строке 4 выполняется $\mathcal{O}(\log i)$ раз, $\mathsf{ttl}(i)$ вычисляется за $\mathcal{O}(\log i)$, группа может быть удалена за $\mathcal{O}(1)$ (см. замечание 4.3.2). □

4.4.2 Обновление групп

Эта стадия состоит из трёх последовательных циклов. Первый цикл обрабатывает j-блоки вида $S[h..i]$, где h — чекпойнт. Для каждого блока мы вычисляем его хэш и берём его группу B из таблицы H_1; если группы B не существует, она создаётся. Вхождение в позиции h свежее, поэтому узел для h добавляется в $B.flist$. Период и расширение вычисляются для этого узла по определению.

Второй цикл перемещает чекпойнты из списка свежих в список обычных. Чекпойнт, который нужно переместить, зависит только от j, и соответствующая группа может быть найдена с использованием таблицы H_2.

Третий цикл удаляет первые элементы чёрствых серий, если эти элементы стали истёкшими. Позиции таких элементов определяются по j, и соответствующая группа может быть найдена с использованием таблицы H_4.

Алгоритм 4.3. Алгоритм R, i-я итерация: обновление групп

1: **for** $\{j \leftarrow 0;\ h \leftarrow i - 2^j + 1$ && $h \geq 1$ && $h + \mathsf{ttl}(h) > i\}$ **do**
2: $I_h \leftarrow HC(h)$; $F \leftarrow \phi(S[h..i])$; $B \leftarrow H_1(j, F)$ ▷ $\phi(S[h..i])$ вычисляется по I_h и I
3: **if** $B = null$ **then**
4: $B \leftarrow$ new group; $H_1(j, F) \leftarrow B$; $B.frame \leftarrow F$
5: $N \leftarrow$ new node; add N to $B.flist$; $N.checkpoint \leftarrow h$
6: $HH(j, h) \leftarrow N$; add h to $B.fseries$; $N.period \leftarrow B.fseries.period$
7: $N.extension \leftarrow \{t$, if $(t, N.period, i') \in W$ for some i'; $B.fseries.first$ otherwise$\}$
8: **for** $\{j \leftarrow 0;\ h \leftarrow i - 2^{j+1} + 1$ && $h \geq 1$ && $h + \mathsf{ttl}(h) > i\}$ **do**
9: $B \leftarrow H_2(j, h)$
10: move $B.flist.first$ to $B.rlist$
11: **for** $\{j \leftarrow 0;\ h \leftarrow i - 3 \cdot 2^j + 1$ && $h \geq 1\}$ **do**
12: $B \leftarrow H_4(j, h)$
13: **if** $B \neq null$ **then**
14: $H_4(j, h{+}B.sseries.period) \leftarrow B$; delete $H_4(j, h)$
15: delete $B.sseries.first$ ▷ см. лемму 4.3.5

Лемма 4.4.2. Алгоритм 4.3 работает за $\mathcal{O}(\log i)$ операций.

Доказательство. Каждый цикл выполняется $\mathcal{O}(\log i)$ раз и содержит константное число обращений к словарям. Группа создаётся за $\mathcal{O}(1)$ времени (замечание 4.3.2). Вычисление $\mathsf{ttl}(h)$ в строках 1,8 требует константного времени, если значение $\beta(i{+}1)$ вычислено один раз (за логарифмическое время) до циклов. В строке 7 мы обращаемся к списку отслеживания W для вычисления расширения $extension$. Так как периоды в расширениях увеличиваются с увеличением j, всего нам нужен один проход по W. По лемме 4.4.4 представленной ниже,

$|W| = \mathcal{O}(\log i)$. Наконец, серии в строке 15 модифицируются за $\mathcal{O}(1)$ времени по лемме 4.3.5. □

4.4.3 Обнаружение периодических подстрок

На этой стадии мы решаем первую из основных задач: найти видимые периодические подстроки, являющиеся суффиксами $S[1..i]$ (см. лемму 4.3.3). Для каждого j мы ищем подстроки с периодами из отрезка $[2^j..2^{j+1}-1]$. Чтобы такая периодическая подстрока существовала, суффикс $T = S[i-2^j+1..i]$ входного потока должен иметь вхождение в некоторой позиции f (см. рис. 4.1). В частности, для T должна существовать группа, и нужно найти её, не зная хэш $\phi(T)$. Заметим, что если T не имеет вхождений в чекпойнтах, то у нас нет возможности вычислить $\phi(T)$.

Пусть T имеет вхождение в чекпойнте k. Рассмотрим строку $T' = S[k..k+2^{j-1}-1] = S[i-2^j+1..i-2^{j-1}]$, которая является префиксом T длины 2^{j-1}. Так как строка T' имеет вхождение в чекпойнте k, ей соответствует группа, скажем B', и у T' есть свежее вхождение в позиции $i-2^j+1$. Это вхождение — первое в $B'.fseries$, потому что оно истекает на текущей итерации. Тогда можно взять $I(i-2^j+1) = B'.fseries.frame1$ и вычислить $F = \phi(T)$ по лемме 1.1.1; см. ниже алгоритм 4.4, строки 5-–8. Заметим что в строках 9–10 мы перемещаем упомянутое вхождение T' из свежих в чёрствые, а в строке 14 добавляем свежее вхождение T.

Алгоритм 4.4. Алгоритм R, i-я итерация: периодические подстроки I

```
1:  for {j ← 0; k = i − 2^j + 1 && k ≥ 1} do
2:      if j = 0 then
3:          F ← φ(S[i])
4:      else
5:          B' ← H_3(j − 1, k)
6:          if B' = null then
7:              continue
8:          F ← φ(S[k..i])                          ▷ из I(i) и B'.fseries.first
9:          add k to B'.sseries; H_4(j, B'.sseries.first) ← B'
10:         H_3(j, k+B'.fseries.period) ← B'; delete H_3(j, k); delete B'.fseries.first
11:     B ← H_1(j, F)
12:     if B = null then
13:         continue
14:     add k to B.fseries; H_3(j, B.fseries.first) ← B
15:     search a visible repetition using j, B      ▷ Алгоритм 4.5
```

Теперь мы знаем группу B строки T. Для нахождения видимых периодических подстрок, мы определяем все варианты для позиции чекпойнта f (см. рис. 4.1; заметим, что T имеет чёрствое вхождение в f), вычисляем p и h для каждого f, и проверяем, задаёт ли тройка (h, p, i) периодическую подстроку. Ниже мы покажем, как сделать все вычисления за $\mathcal{O}(1)$ времени. Два основных шага:

(i) Выбираем не более двух кандидатов для f, гарантируя что другие вхождения T не ведут к периодическим подстрокам вида (h, p, i) с $p \in [2^j..2^{j+1}-1]$;

(ii) Для фиксированного f вычисляем p, h, и проверяем, определяет ли (h, p, i) периодическую подстроку.

Начнём с (ii). Из i, j, f получаем период $p = i - f + 2^j + 1$, $z = 2^{\max\{0, j-t_\varepsilon\}}$ и чекпойнт $h = \lfloor \frac{i-2p+1}{z} \rfloor \cdot z$. В группе $B_h = H_2(j, h)$ проверяем за константное число арифметических операций наличие свежего вхождения в позиции $h+p$. Если его нет, то нет и периодической подстроки с периодом p. Если есть, мы знаем, что $S[h..i]$ имеет короткий период p. Осталось проверить что p — примитивный. Если он не примитивный, мы можем написать $S[i-2p+1..i] = U^{2t}$, где U — примитивное, $|U| = q$, $t > 1$, $p = qt$. Тогда U — суффикс T, а T — суффикс U^t. Тогда T имеет вхождения, заканчивающиеся в позициях $i, i-q, \ldots, i-tq$ и не имеет вхождений между указанными позициями, потому что U, будучи примитивным, входит в каждую подстроку U^2 только два раза. Поскольку $q \le \frac{p}{2} < 2^j$, вхождение T, заканчивающиеся в $i-q$, является свежим по определению, а значит, $B.fseries.period = q$. Вхождение T в позиции f, которое заканчивается в $i-tq$, чёрствое, то есть последнее чёрствое вхождение T находится на расстоянии q от первого свежего вхождения: $B.fseries.first - B.sseries.last = q$. С другой стороны, если $q = B.fseries.period$ делит p и равно $B.fseries.first - B.sseries.last$, то строка $S[i-p+1..i]$ имеет период q, который делит её длину, а значит, период p не примитивный. Таким образом, мы проверяем эти два условия и выдаём периодическую подстроку (h, p, i) тогда и только тогда, когда хотя бы одно из них ложно. Общее число совершенных операций есть $\mathcal{O}(1)$.

Теперь рассмотрим (i). Пусть f_1 — последний чекпойнт в $B.rlist$. Если вхождение T в позиции f_1 — не чёрствое, то T не имеет чёрствых вхождений, то есть кандидатов для f нет. Допустим теперь, что это вхождение — чёрствое и определим f_1 как кандидата. Если $|B.sseries| \le 2$, то только два последних узла в $B.rlist$ могут быть кандидатами, и мы добились требуемого. Поэтому далее рассмотрим случай $|B.sseries| \ge 3$ и воспользуемся периодами: теперь строка T периодична с минимальным периодом $q = B.sseries.period$. Пусть $p = i - f_1 - 2^j + 1$. Тогда возможные периоды периодических подстрок, соответствующих чёрствым вхождениям T, имеют вид $p + mq$, где $m \ge 0$, $p + mq < 2^{j+1}$ (мы должны рассмотреть худший случай, когда все чёрствые вхождения являются чекпойнтами).

Если $S[1..i]$ заканчивается на периодическую подстроку с периодом p, то строки $S[i-2p+1..i-p]$ и $S[i-p+1..i]$ равны, а значит, имеют одинаковый наибольший суффикс с периодом q. Пусть V (соотв., V') это наибольший суффикс $S[1..i]$ (соотв., $S[1..i-p]$) с периодом q. Заметим, что суффикс $S[1..i]$ длины $p+q$ имеет период p за счёт двух вхождений T на таком расстоянии; если он также имеет период q, являющийся примитивным, то по лемме 4.2.1 q делит p. Тогда ни один из периодов $p+mq$ не является примитивным, а значит, в рассматриваемом

диапазоне периодов нет ни одной видимой периодической подстроки. Поэтому допустим что $|V| < p+q$.

С другой стороны, T — периодическая подстрока с периодом q, и суффикс $S[i-p+1..i]$, так что подстрока V тоже периодическая; пусть $V = (x, q, i)$. Пусть l такое, что в списке отслеживания есть периодическая подстрока вида (l, q, r); если W не содержит периодической подстроки с периодом q, положим $l = i - 2^j + 1$ равным позицией суффикса T в $S[1..i]$. Из леммы 4.3.1 мы получаем $x \in [l-\lfloor q\varepsilon \rfloor+1..l]$. Аналогично, строка $V' = (x', q, i-p)$ — периодическая и имеет суффикс T. Аналог значения l, используемого выше, хранится как $l' = f_1.extension$, и мы опять по лемме 4.3.1 получаем $x' \in [l'-\lfloor q\varepsilon \rfloor+1..l']$.

Если p — период периодической подстроки, то или $|V|, |V'| \geq p$ или $V = V'$. За $\mathcal{O}(1)$ арифметических операций можно проверить, выполняется ли каждое из двух условий. В зависимости от результата да/нет, f_1 является/ не является кандидатом для f. Далее рассмотрим отличный от f_1 чекпойнт f_2, который соответствует периоду $p+mq$ для некоторого $m \geq 1$, и пусть V'' — длиннейший суффикс строки $S[1..i-p-mq+1]$, имеющий период q. Так как $|V| < p + mq$, равенство $V'' = V$ — необходимое условие для наличия периодической подстроки с периодом q. Заметим, что V'' — префикс V', то есть входит в S в той же самой позиции x'; найденные выше диапазоны для значений x и x' имеют длину $\lfloor q\varepsilon \rfloor$, и при этом $2\cdot\lfloor q\varepsilon \rfloor - 1 \leq q$. Следовательно, существует не более одного периода вида $p + mq$, для которого равенство $|V''| = |V|$ возможно для некоторых значений x и x'; все остальные чёрствые вхождения T не соответствуют периодическим подстрокам. Мы вычисляем значение m с помощью арифметических операций; позиция $f_2 = f_1 - mq$ это единственный возможный второй кандидат для f. Таким образом (i) доказано. Приведенное рассуждение обосновывает корректность описанного ниже алгоритма 4.5; пример с двумя кандидатами для f приведен на рис. 4.2.

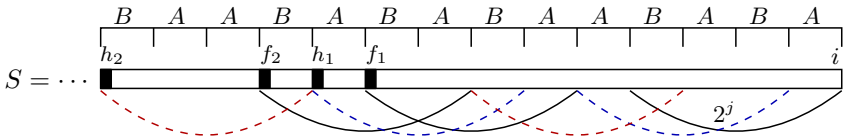

Рис. 4.2: Пример обнаружения периодических подстрок. A и B — строки длины 2^{j-2}. Суффикс $T = BABA$ имеет два чёрствых вхождения в чекпойнтах f_1 и f_2 (возможные периоды $p_1 = 5 \cdot 2^{j-2}$, $p_2 = 7 \cdot 2^{j-2}$, соответственно). Периодические подстроки (h_1, p_1, i) и (h_2, p_2, i) обнаружены. Полусвежая серия имеет короткий период $q = 2^{j-1}$. Получаем $V = V'' = uABABA, V' = uABABABA$, где u наибольший общий суффикс A и B. Чекпойнт f_1 является кандидатом, так как удовлетворяет $|V|, |V'| \geq p_1$.

В совокупности, мы получаем следующую лемму:

Лемма 4.4.3. Все видимые периодические подстроки, являющиеся суффиксами $S[1..i]$, могут быть вычислены за $\mathcal{O}(\log i)$ операций.

Доказательство. С увеличением j, периоды q (строка 10 алгоритма 4.5) не уменьшаются. Следовательно, все запросы к списку отслеживания W (строка

Алгоритм 4.5. Алгоритм R, i-я итерация: периодические подстроки II

$f_1 \leftarrow B.rlist.top$ ▷ позиция самого правого обычного вхождения T
if $f_1 \leq i - 3 \cdot 2^j$ **then**
    continue ▷ нет чёрствых вхождений в чекпойнтах; переход к следующему j в алг. 4.4
else
    $C \leftarrow \{f_1\}$ ▷ всегда включаем f_1 в список C кандидатов
    if $|B.sseries| \leq 2$ **then**
        if $f_1.previous > i - 3 \cdot 2^j$ **then** ▷ чёрствое вхождение в предыдущем чекпойнте
            add $f_1.previous$ to C
    else ▷ найдём границы q-периодичности
        $p \leftarrow i - f_1 - 2^j + 1$; $q \leftarrow B.sseries.period$; $l' \leftarrow f_1.extension$
        $l \leftarrow \{t$, if $(t, q, i') \in W$ for some i'; $B.fseries.first$ otherwise$\}$
        if $\lceil \frac{l-l'+1-\lfloor q\varepsilon \rfloor - p}{q} \rceil = \lfloor \frac{l-e-1+\lfloor q\varepsilon \rfloor - p}{q} \rfloor$ **then**
            $f_2 \leftarrow f_1 - \lfloor \frac{l-l'-1+\lfloor q\varepsilon \rfloor - p}{q} \rfloor \cdot q$ ▷ вхождение, для которого возможно $V'' = V$
            if $H_2(j, f_2)$ exists **then** ▷ f_2 является чекпойнтом
                add f_2 to C
for $\{f \in C\}$ **do**
    $p \leftarrow i - f - 2^j + 1$; $z \leftarrow 2^{\max\{0, j - t_\varepsilon\}}$; $h \leftarrow \lfloor \frac{i-2p}{z} \rfloor \cdot z$; $B_h \leftarrow H_2(j, h)$
    if $h + p \in B_h.fseries$ **then** ▷ найден период p
        $q \leftarrow B.fseries.period$ ▷ проверка примитивности p
        if $\{p \neq 0 \bmod q \;||\; B.fseries.first - B.sseries.last \neq q\}$ **then**
            add (h, p, i) to New ▷ добавление в список новых периодических подстрок

11 алгоритма 4.5) могут быть выполнены за $\mathcal{O}(|W|)$ времени с использованием указателя. По лемме 4.4.4, $|W| = \mathcal{O}(\log i)$. Помимо обращений к W, для каждого j алгоритмы 4.4 и 4.5 совершают константное число операций. □

4.4.4 Обновление списка отслеживания

После предыдущей стадии, у нас есть список New видимых периодических подстрок, обнаруженных на текущей итерации. Теперь нам нужно слить New со списком отслеживания W, и после этого удалить из W «старые» периодические подстроки. Нашей целью является доказательство следующей леммы.

Лемма 4.4.4. На i-й итерации, список отслеживания (i) имеет длину $\mathcal{O}(\log i)$ и (ii) может быть обновлен за $\mathcal{O}(\log i)$ операций.

Доказательство (i) требует удаления некоторых периодических подстрок из W раньше, чем описано в определении, если такие периодические подстроки не могут впоследствии обновиться. Алгоритм для (ii) — это алгоритм 4.6, описанный ниже. Нам потребуется две леммы.

Лемма 4.4.5 (Лемма о трёх квадратах, [21])**.** Если три квадрата с примитивными периодами $p_1 < p_2 < p_3$ заканчиваются в одной позиции строки, то $p_3 \geq p_1 + p_2$.

Лемма 4.4.6. Пусть даны периодические подстроки $(h, p, i) \in New$ и $(l, q, r) \in W$, такие что $r < i$ и $\frac{2}{3}p < q < \frac{3}{2}p$. За $\mathcal{O}(1)$ операций можно проверить, является ли (l, q, i) периодической подстрокой.

Доказательство. Пусть $j = \lfloor \log p \rfloor$, $T = S[i - 2^j + 1..i]$. Так как периодическая подстрока (h, p, i) добавлена на текущей итерации, то T — это j-блок в чекпойнте $f = i - p - 2^j + 1$ (непрерывные дуги на рис. 4.3).

Сначала рассмотрим случай $q > p$. Так как $q < \frac{3}{2}p \le p + 2^j$, то взаимное расположение периодических подстрок (h, p, i) и (l, q, r) совпадает с изображённым на рис 4.3,a. Периодическая подстрока (l, q, r) расширяется до (l, q, i) тогда и только тогда, когда суффикс T имеет вхождение в позиции $i - q - 2^j + 1$ (пунктирная дуга на рис. 4.3,a). Так как $q - p < 2^j$, то это вхождение пересекается со вхождением в позиции f. Следовательно, в группе j-блока T расширение $f.extension$ нетривиально. А именно, $f.period$ делит $q - p$, и $f.extension \le i - q - 2^j + 1$. Так что мы находим узел позиции f как $HH(j, f)$ и проверяем значения $f.period$ и $f.extension$; всего выполняется $\mathcal{O}(1)$ операций.

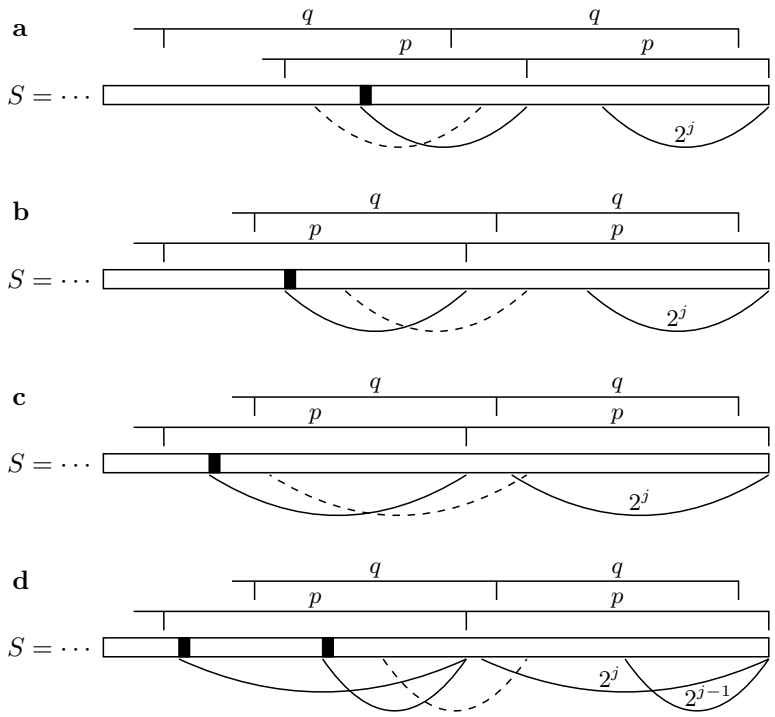

Рис. 4.3: Взаимное расположение новой периодической подстроки с периодом p и живой периодической подстроки с периодом q, где q близко к p. Чёрные прямоугольники обозначают живые чекпойнты.

Теперь пусть $q < p$. Если $r - q \le i - p$, то (l, q, i) не задаёт периодическую подстроку: иначе суффикс длины $p + q$ строки $S[1..i]$ будет имеет примитивные периоды p и q, что противоречит лемме 4.2.1(2). Следовательно, периодические подстроки (h, p, i), (l, q, r) и вхождения T расположены как на рис. 4.3,b–d. Если $2^j < q$ (рис. 4.3,b), то (l, q, i) задает периодическую подстроку тогда и только тогда, когда T имеет чёрствое вхождение в позиции $i - q - 2^j + 1$ (пунктирная дуга). Если $q \le 2^j \le q + i - r$ (рис. 4.3,c), то существование

периодической подстроки (l, q, i) эквивалентно наличию свежего вхождения T в позиции $i - q - 2^j + 1$ (пунктирная дуга). Наконец, если $2^j > q + i - r$ (рис. 4.3,d), мы не можем больше полагаться только на вхождения T. Например, может получиться, что $l > i - q - 2^j + 1$; и если мы не видим вхождения T в позиции $i - q - 2^j + 1$, которая находится вне периодической подстроки с периодом q, мы не можем извлечь полезных следствий. Однако в этом случае $2^j > q > \frac{2}{3}p$, и следовательно позиция $f' = i - p - 2^{j-1} + 1$ — это чекпойнт (или $\beta(f') = j-1$ и $\mathsf{ttl}(f') \geq 2^{j+1} > p + 2^{j-1}$, или $\beta(f') = \beta(f)$ и эти две позиции имеют одинаковый ttl). Пусть $T' = S[i - 2^{j-1} + 1..i]$; это $(j-1)$-блок со вхождением в чекпойнте f'. Тогда (l, q, i) — периодическая подстрока тогда и только тогда, когда T' имеет чёрствое вхождение в позиции $i - q - 2^{j-1} + 1$ (пунктирная дуга на рис. 4.3,d). Следовательно, во всех трёх случаях достаточно проверить имеет ли данный блок (свежее или чёрствое) вхождение в заданной позиции, что требует $\mathcal{O}(1)$ операций. Лемма доказана. □

Алгоритм 4.6. Алгоритм R, i-я итерация: обновление списка отслеживания

for $(h, p, i) \in New$ **do** ▷ добавление новых периодических подстрок
  if $(l, p, r) \in W$ for some l, r **then**
    update (l, p, r) to (l, p, i)
  else
    add (h, p, i) to W
for $(l, p, r) \in W$ **do**
  $j \leftarrow \lfloor \log p \rfloor$; $z \leftarrow 2^{\max\{0, j - t_\varepsilon\}}$
  if $r + z = i$ **then** ▷ периодическая подстрока пропустила обновление
    delete (l, p, r) from W; output (l, p, r)
for $(h, p, i) \in W$ **do** ▷ удаления/обновления по лемме 4.4.6
  $j \leftarrow \lfloor \log p \rfloor$; $f \leftarrow i - p - 2^j + 1$; $B \leftarrow H_2(j, f)$; $B' \leftarrow H_2(j-1, i-p-2^{j-1}+1)$
  for $\{(l, q, r) \in W$ such that $2p/3 < q < p\}$ **do**
    if $(r - q \leq l - p)$ || $(q > 2^j$ && $i - q - 2^j + 1 \notin B.sseries)$
      || $(q \leq 2^j \leq q + i - r$ && $i - q - 2^j + 1 \notin B.fseries)$
      || $(q + i - r \leq 2^j$ && $i - q - 2^{j-1} + 1 \notin B'.sseries)$ **then**
      delete (l, q, r) from W; output (l, q, r)
    else
      update (l, q, r) to (l, q, i)
  for $\{(l, q, r) \in W$ such that $p < q < 3p/2\}$ **do**
    if $(f.period = 0)$ || $(q - p \neq 0 \bmod f.period)$
      || $(f.extension > i - q - 2^j + 1)$ **then**
      delete (l, q, r) from W; output (l, q, r)
    else
      update (l, q, r) to (l, q, i)

Доказательство леммы 4.4.4. Докажем индукцией по i следующий факт:

(∗) *для любого положительного целого r, список отслеживания после i-й итерации содержит не более двух периодических подстрок с периодами из отрезка $[r..\frac{3}{2}r]$.*

База индукции очевидна. Для каждого шага рассмотрим отрезок $[r..\frac{3}{2}r]$ содержащий период хотя бы одной новой периодической подстроки (иначе, доказывать нечего). После применения третьего внешнего цикла алгоритма 4.6, все оставшиеся периодические подстроки с периодами из отрезка $[r..\frac{3}{2}r]$ заканчиваются в позиции i, и их не более двух по лемме 4.4.5. Следовательно, $(*)$ доказано. Очевидно, что $(*)$ влечёт утверждение (i).

Каждый из трёх внешних циклов алгоритма 4.6 может быть выполнен за время, пропорциональное суммарной длине задействованных списков с использованием одного или двух указателей. Факт $(*)$ показывает, что указатель для q в третьем внешнем цикле никогда не идёт назад более чем на два элемента. Так как $|New| = \mathcal{O}(\log i)$ по лемме 4.4.5, (i) влечёт (ii). □

Доказательство теоремы 4.1.1. Алгоритм, описанный в разделе 4.4 удовлетворяет лемме 4.3.3 и, следовательно, решает задачу approxRuns. Ограничение на используемую память основывается на лемме 2.3.5: количество ключей в каждом словаре, суммарный размер групп и суммарная длина списков чекпойнтов ограничены количеством чекпойнтов, умноженным на $\log n$. Размер списка отслеживания пренебрежимо мал. Ограничение на время работы следует из лемм 4.3.5–4.4.4. Для вставки/удаления групп и узлов см. замечание 4.3.2. □

В заключение скажем несколько слов про выбор словарей.

Замечание 4.4.1. Если параметр $\varepsilon = \varepsilon(n)$ мал (обратно полиномиален), имеет смысл использовать в качестве словарей динамические совершенные хэш-таблицы [6, 23]. Обе процитированные версии гарантируют, что с вероятностью $1 - m^{-c}$, где m — размер словаря, а c — произвольная константа, все операции со словарём будут работать за $\mathcal{O}(1)$ времени. Таким образом, общая вероятность неудачного запуска алгоритма будет оставаться меньше $1/n$, при этом будет $\mathcal{O}(\log n)$ элементарных операций между чтениями символов. Однако, это не так в случае больших (например константа или обратный полилогарифм) значений ε. В этом случае мы рекомендуем использовать детерминированные словари Андерсона и Торапа [3], которые дадут оценку в $\mathcal{O}\big(\sqrt{\frac{\log\log n}{\log\log\log n}} \cdot \log n\big)$ элементарных операций между чтениями.

Заключение

Итоги выполненного исследования

В диссертации рассмотрены задачи поиска основных регулярных структур в строках в модели потока данных. Представим краткое резюме по каждой из трёх основных глав и сформулируем связанные открытые проблемы для дальнейшего исследования.

Отметим, что все разработанные нами алгоритмы являются рандомизированными алгоритмами типа Монте-Карло, т.е. решают задачи с высокой вероятностью, используя детерминированный объём памяти за детерминированное время. При этом доказано, что другие типы алгоритмов для решения поставленных задач требуют как минимум линейных затрат памяти, а значит, непригодны для потоковой модели.

Палиндромы в потоках

Для задачи LPS о наибольшем палиндроме в модели потока данных мы представили алгоритмы реального времени для приближенного решения с аддитивной и мультипликативной погрешностью. При этом объём памяти, используемый представленными алгоритмами, асимптотически точно совпадает с имеющимися нижними оценками при большинстве значений параметров задачи. Таким образом, вопрос о вычислительной сложности задачи LPS в модели потока данных разрешён.

Отдельно можно отметить введённую функцию времени жизни (ttl), которая показала себя удобным инструментом и для решения задачи о поиске максимальных периодических подстрок (approxRuns).

Повторы и обратные повторы в потоках

Мы показали, что задача поиска наибольшего повтора(LRS) и задача поиска наибольшего обратного повтора(LRRS) в модели потока данных могут быть решены только с использованием приближенных алгоритмов типа Монте-Карло. И доказали нижнюю оценку в $\Omega(\frac{n}{E}\log\min\{|\Sigma|, \frac{n}{E}\})$ бит памяти для приближения ответа с аддитивной погрешностью E.

Для обеих задач представлены алгоритмы типа Монте-Карло, близкие к реальному времени, использующие $\mathcal{O}(E + \frac{n}{E})$ слов памяти, где E — аддитивная

ошибка приближения. При этом используемая память точно совпадает с нижними оценками при условиях $E = \mathcal{O}(\sqrt{n})$ и $|\Sigma| = \Omega(n^{0.01})$. Алгоритмы основаны на суффиксных деревьях, что весьма необычно для потоковой модели.

Максимальные периодические подстроки в потоках

Мы доказали, что в потоковой модели не существует алгоритма, точно находящего все максимальные периодические подстроки и сформулировали приближенную версию задачи (approxRuns).

Для приближенной задачи мы представили алгоритм типа Монте-Карло, близкий к реальному времени, и использующий $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$ слов памяти.

Перспективы дальнейшей разработки темы

В задаче поиска наибольшего повтора(LRS) и задаче поиска наибольшего обратного повтора(LRRS) в модели потока данных открытыми остаются вопросы о нижней оценке на необходимую память для приближения с мультипликативной ошибкой, а также с аддитивной ошибкой $E = \Omega(\sqrt{n})$. Кроме того, открыт вопрос о потоковом алгоритме с мультипликативной ошибкой.

В задаче приближенного поиска всех максимальных периодических подстрок в потоках открытым остаётся вопрос о нижней оценке на необходимую память.

Список литературы

[1] Alon N., Matias Y., Szegedy M. The space complexity of approximating the frequency moments // Journal of Computer and system sciences. — 1999. — Vol. 58, no. 1. — P. 137–147.

[2] Towards real-time suffix tree construction / A. Amir, T. Kopelowitz, M. Lewenstein, N. Lewenstein // International Symposium on String Processing and Information Retrieval / Springer. — 2005. — P. 67–78.

[3] Andersson A., Thorup M. Dynamic ordered sets with exponential search trees // J. ACM. — 2007. — Vol. 54, no. 3. — P. 13.

[4] Apostolico A., Breslauer D., Galil Z. Parallel detection of all palindromes in a string // Theoret. Comput. Sci. — 1995. — Vol. 141. — P. 163–173.

[5] Parallel construction of a suffix tree with applications / A. Apostolico, C. Iliopoulos, G. M. Landau et al. // Algorithmica. — 1988. — Vol. 3, no. 1-4. — P. 347–365.

[6] Arbitman Yuriy, Naor Moni, Segev Gil. De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results // Automata, Languages and Programming, 36th International Colloquium, ICALP 2009. — Vol. 5555 of Lecture Notes in Computer Science. — Springer, 2009. — P. 107–118.

[7] The "Runs" Theorem / H. Bannai, T. I, S. Inenaga et al. // SIAM J. Comput. — 2017. — Vol. 46, no. 5. — P. 1501–1514.

[8] Counting distinct elements in a data stream / Z. Bar-Yossef, TS Jayram, R. Kumar et al. // International Workshop on Randomization and Approximation Techniques in Computer Science / Springer. — 2002. — P. 1–10.

[9] Palindrome recognition in the streaming model / P. Berenbrink, F. Ergün, F. Mallmann-Trenn, E. Sadeqi Azer // STACS 2014. — Vol. 25 of LIPIcs. — Germany : Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl Publishing, 2014. — P. 149–161.

[10] Palindromic length in linear time / K. Borozdin, D. Kosolobov, M. Rubinchik, A. M. Shur // 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017) / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. — 2017.

[11] Breslauer D., Galil Z. Real-time streaming string-matching // Combinatorial Pattern Matching. — Vol. 6661 of LNCS. — Berlin : Springer, 2011. — P. 162–172.

[12] Breslauer D., Italiano G. F. Near real-time suffix tree construction via the fringe marked ancestor problem // Journal of Discrete Algorithms. — 2013. — Vol. 18. — P. 32–48.

[13] Computing a longest common palindromic subsequence / S. R. Chowdhury, Md Hasan, S. Iqbal et al. // Fundamenta Informaticae. — 2014. — Vol. 129, no. 4. — P. 329–340.

[14] Dictionary Matching in a Stream / R. Clifford, A. Fontaine, E. Porat et al. // ESA 2015. — Vol. 9294 of LNCS. — Springer, 2015. — P. 361–372.

[15] The k-mismatch problem revisited / R. Clifford, A. Fontaine, E. Porat et al. // SODA 2016. — SIAM, 2016. — P. 2039–2052.

[16] Clifford R., Kociumaka T., Porat E. The streaming k-mismatch problem // Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms / SIAM. — 2019. — P. 1106–1125.

[17] Introduction to Algorithms / T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. — second edition. — The MIT Press, 2001.

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

[19] Order-preserving indexing / M. Crochemore, C. S. Iliopoulos, T. Kociumaka et al. // Theoretical Computer Science. — 2016. — Vol. 638. — P. 122–135.

[20] Near-Optimal Computation of Runs over General Alphabet via Non-Crossing LCE Queries / M. Crochemore, C. S. Iliopoulos, T. Kociumaka et al. // String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016. — Vol. 9954 of Lecture Notes in Computer Science. — 2016. — P. 22–34.

[21] Crochemore M., Rytter W. Squares, Cubes, and Time-Space Efficient String Searching // Algorithmica. — 1995. — Vol. 13. — P. 405–425.

[22] Crochemore M., Rytter W. Jewels of stringology: text algorithms. — World Scientific, 2002.

[23] Dietzfelbinger M., auf der Heide F. M. Dynamic hashing in real time // Informatik. — Springer, 1992. — P. 95–119.

[24] Fine N. J., Wilf H. S. Uniqueness theorems for periodic functions // Proc. Amer. Math. Soc. — 1965. — Vol. 16. — P. 109–114.

[25] Beyond the Runs Theorem / J. Fischer, S. Holub, T. I, M. Lewenstein // String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015. — Vol. 9309 of Lecture Notes in Computer Science. — Springer, 2015. — P. 277–286.

[26] Galil Z. Real-time algorithms for string-matching and palindrome recognition // Proc. 8th annual ACM symposium on Theory of computing (STOC'76). — New York, USA : ACM, 1976. — P. 161–173.

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

[28] Galil Z., Seiferas J. A Linear-Time On-Line Recognition Algorithm for "Palstar" // J. ACM. — 1978. — Vol. 25. — P. 102–111.

[29] Gasieniec L., Kolpakov R. M., Potapov I. Space efficient search for maximal repetitions // Theor. Comput. Sci. — 2005. — Vol. 339, no. 1. — P. 35–48.

[30] Faster Longest Common Extension Queries in Strings over General Alphabets / P. Gawrychowski, T. Kociumaka, W. Rytter, T. Walen // 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016. — Vol. 54 of LIPIcs. — Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. — P. 5:1–5:13.

[31] Gawrychowski P., Manea F., Nowotka D. Testing Generalised Freeness of Words // STACS 2014. — Vol. 25 of LIPIcs. — Dagstuhl Publishing, 2014. — P. 337–349.

[32] Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams / P. Gawrychowski, O. Merkurev, A. M. Shur, P. Uznanski // Proceedings CPM 2016. — Vol. 54 of LIPIcs. — Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. — P. 18:1–18:13.

[33] Tight tradeoffs for real-time approximation of longest palindromes in streams / P. Gawrychowski, O. Merkurev, A. M. Shur, P. Uznanski // Algorithmica. — 2019. — Vol. 81, no. 9. — P. 3630–3654.

[34] Gusfield D. Algorithms on Strings, Trees and Sequences. Computer Science and Computational Biology. — Cambridge University Press, 1997.

[35] Holub S. Prefix frequency of lost positions // Theor. Comput. Sci. — 2017. — Vol. 684. — P. 43–52.

[36] Jalsenius M., Porat B., Sach B. Parameterized Matching in the Streaming Model // STACS 2013. — Vol. 20 of LIPIcs. — Dagstuhl Publishing, 2013. — P. 400–411.

[37] Counting arbitrary subgraphs in data streams / D. M Kane, K. Mehlhorn, T. Sauerwald, H. Sun // International Colloquium on Automata, Languages, and Programming / Springer. — 2012. — P. 598–609.

[38] Kari L., Mahalingam K. Watson–Crick palindromes in DNA computing // Natural Computing. — 2010. — Vol. 9, no. 2. — P. 297–316.

[39] Karp R., Rabin M. Efficient randomized pattern matching algorithms // IBM Journal of Research and Development. — 1987. — Vol. 31. — P. 249–260.

[40] Knuth D. E., Morris J., Pratt V. Fast pattern matching in strings // SIAM J. Comput. — 1977. — Vol. 6. — P. 323–350.

[41] Kolpakov R., Kucherov G. Finding maximal repetitions in a word in linear time // Proc. 40th Ann. Symp. Found. Comput. Sci. — IEEE Press, 1999. — P. 596–604.

[42] Kopelowitz T. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list // Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on / IEEE. — 2012. — P. 283–292.

[43] Kopelowitz T., Lewenstein M. Dynamic weighted ancestors // Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms / Society for Industrial and Applied Mathematics. — 2007. — P. 565–574.

[44] Kosolobov D. Lempel-Ziv Factorization May Be Harder Than Computing All Runs // 32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany. — Vol. 30 of LIPIcs. — Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. — P. 582–593.

[45] Kosolobov D. Computing runs on a general alphabet // Inf. Process. Lett. — 2016. — Vol. 116, no. 3. — P. 241–244.

[46] Kosolobov D., Rubinchik M., Shur A. M. Pal^k is linear recognizable online // Proc. 41th Int. Conf. on Theory and Practice of Computer Science (SOFSEM 2015). — Vol. 8939 of LNCS. — Springer, 2015. — P. 289–301.

[47] Main Michael G. Detecting leftmost maximal periodicities // Discrete Applied Mathematics. — 1989. — Vol. 25, no. 1-2. — P. 145–153.

[48] 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.

[49] McGregor A. Graph stream algorithms: a survey // ACM SIGMOD Record. — 2014. — Vol. 43, no. 1. — P. 9–20.

[50] Merkurev O., Shur A. M. Searching Long Repeats in Streams // Proceedings CPM 2019. — Vol. 128 of LIPIcs. — Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. — P. 31:1–31:14.

[51] Merkurev O., Shur A. M. Searching Runs in Streams // International Symposium on String Processing and Information Retrieval / Springer. — Vol. 11811 of LNCS. — 2019. — P. 203–220.

[52] Munro J. I., Paterson M. S. Selection and sorting with limited storage // Theoretical computer science. — 1980. — Vol. 12, no. 3. — P. 315–323.

[53] Porat B., Porat E. Exact and approximate pattern matching in the streaming model // FOCS 2009. — IEEE Computer Society, 2009. — P. 315–323.

[54] Rubinchik M., Shur A. M. EERTREE: An Efficient Data Structure for Processing Palindromes in Strings // Combinatorial Algorithms - 26th International Workshop, IWOCA 2015, Revised Selected Papers. — Vol. 9538 of LNCS. — Springer, 2016. — P. 321–333.

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

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

[57] Weiner P. Linear pattern matching algorithms // Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on / IEEE. — 1973. — P. 1–11.

[58] Willard Dan E. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$ // Information Processing Letters. — 1983. — Vol. 17. — P. 81–84.

[59] Yao A. C.-C. Probabilistic Computations: Toward a Unified Measure of Complexity (Extended Abstract) // FOCS 1977. — IEEE Computer Society, 1977. — P. 222–227.

Список иллюстраций

2.1 Поиск палиндрома, который может обновить ответ. Квадраты обозначают позиции j, такие что фрейм $I(j)$ сохранён; скобки показывают подстроки, которые могут быть проверены на палиндромность. По лемме 2.3.1, только подстроки-«кандидаты» могут быть палиндромами длины $> answer.len$. . . 22

2.2 Состояние списка SP после итерации $i = 53$ (подразумевается $q_\varepsilon = 1$). Чёрные квадраты обозначают числа j, такие что фреймы $I(j)$ сохранены в текущий момент. К примеру, из (2.1) следует что $\mathsf{ttl}(28) = 2^{1+2+2} = 32$, так что $I(28)$ будет сохранено в SP до итерации $28 + 32 = 60$. 23

3.1 Суффиксное дерево $\mathcal{T}(ABABAC)$. Слева метки на рёбрах записаны в виде строк. Справа, в том же суффиксном дереве, метки записаны в виде (sd, последняя позиция вхождения). На обеих картинках цветом изображены суффиксные ссылки. На правой картинке красная точка соответствует позиции $pos = (v, 1)$. 34

3.2 Следы и сжатые повторы. Сверху: сжимаемый повтор $(4, 19, 8)$ и его сжатый повтор $(3, 1, 5, 2)$ (выделено цветом). Снизу: хэш-буквы, доступные во всех прямых следах после прочтения $S[17]$ (выделено цветом). 36

3.3 Суффиксное дерево построенное по основному следу Q_4, равное $\mathcal{T}(ABABA)$. В следах $P_1, \ldots, P_4$ цветом выделены $suf f_r^i$; стрелками показаны pos_r^i. . . . 38

4.1 Видимая периодическая строка покрыта перекрывающимися вхождениями двух j-блоков. 54

4.2 Пример обнаружения периодических подстрок. A и B — строки длины 2^{j-2}. Суффикс $T = BABA$ имеет два чёрствых вхождения в чекпойнтах f_1 и f_2 (возможные периоды $p_1 = 5 \cdot 2^{j-2}$, $p_2 = 7 \cdot 2^{j-2}$, соответственно). Периодические подстроки (h_1, p_1, i) и (h_2, p_2, i) обнаружены. Полусвежая серия имеет короткий период $q = 2^{j-1}$. Получаем $V = V'' = uABABA, V' = uABABABA$, где u наибольший общий суффикс A и B. Чекпойнт f_1 является кандидатом, так как удовлетворяет $|V|, |V'| \geq p_1$. 63

4.3 Взаимное расположение новой периодической подстроки с периодом p и живой периодической подстроки с периодом q, где q близко к p. Чёрные прямоугольники обозначают живые чекпойнты. 65