Содержание

1 Введение

Можно ли уменьшить объем хранимой информации без потери скорости доступа к ней? Т.е. можно ли решать классические задачи для строк на сжатых строках без полной распаковки? Известны различные представления сжатых строк, например, прямолинейные программы (ПП) [29, 5, 16], коллаж-системы [22], представления с помощью антисловарей [24]. На данный момент сжатие текста с помощью контекстно-свободных грамматик или, что эквивалентно, с помощью ПП, привлекает много внимания, поскольку грамматики обеспечивают более структурированное сжатие. При этом текст T порождается контекстно-свободной грамматикой $\mathcal{T}$. Для простоты предполагается, что мы имеем дело с грамматиками, порождающими только одно слово. Но даже такое хорошо структурированное представление сжатого текста не является универсальным. Так, встречаются задачи, которые для сжатых строк не имеют полиномиального алгоритма. Например, задача **Сжатого расстояния Левенштейна** или задача **Наибольшей общей подпоследовательности** [26, 25]. Более того, возникают удивительные ситуации, когда элементарная задача на строках оказывается неразрешимой в случае сжатых строк. Так, задача **Сжатого расстояния Хэмминга** [26] является $\#P$-полной, если обе строки представлены в виде ПП. Однако, известен широкий спектр задач, для которых существуют полиномиальные алгоритмы, например задача **Равенства двух сжатых текстов** [28] или задача **Поиска сжатого образца в сжатом тексте** [26].

Мы будем рассматривать строки, представленные прямолинейными программами. Формально, прямолинейная программа это контекстно-свободная грамматика в нормальной форме Хомского, которая порождает в точности одну строку. Длина строки, выводимой из прямолинейной программы, может быть экспоненциально больше, чем размер программы. Поэтому преобразование строки в ПП может рассматриваться как способ сжатия текста.

В первой части работы мы обсудим известную логарифмическую связь между Лемпель-Зив-факторизацией (LZ-факторизацией) и минимальной грамматикой для произвольного текста T. Также мы представим реализацию простейших операций над ПП. Во второй части работы мы сформулируем ряд классических строковых задач, для которых существуют полиномиальные алгоритмы на сжатых строках, и приведем лучшие на данный момент алгоритмы, которые позволяют их решить. В

заключительной части работы мы рассмотрим несколько операций над сжатыми строками, для который не существует полиномиального алгоритма, и дадим оценку их сложности.

2 Обозначения

Зафиксируем произвольный алфавит Σ. *Прямолинейная программа* – это последовательность правил вывода вида:

$$X_1 = expr_1, X_2 = expr_2, \ldots, X_n = expr_n,$$

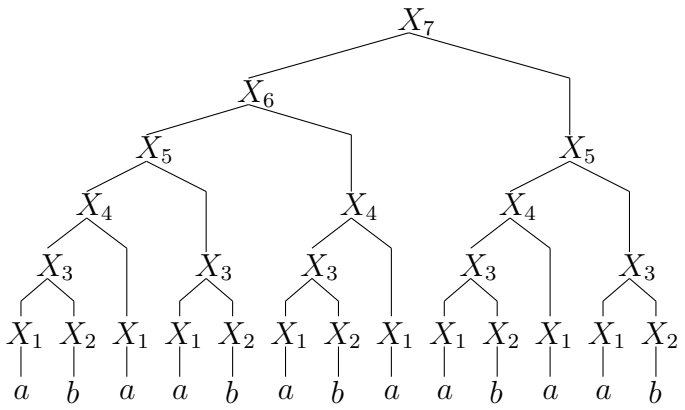
где X_i – это переменные, а $expr_i$ – выражения вида:

- $expr_i$ – символ из алфавита Σ (такие правила будем называть терминальными), или
- $expr_i = X_l \cdot X_r (l, r < i)$ (такие правила будем называть нетерминальными), где «$\cdot$» обозначает конкатенацию правил X_l и X_r.

Пример: Рассмотрим ПП $\mathcal{X}$, которая порождает текст «*abaababaabaab*»:

$$X_1 = a, X_2 = b, X_3 = X_1 \cdot X_2, X_4 = X_3 \cdot X_1,$$
$$X_5 = X_4 \cdot X_3, X_6 = X_5 \cdot X_4, X_7 = X_6 \cdot X_5$$

Тогда дерево вывода грамматики имеет вид:



В дальнейшем дерево вывода грамматики $\mathcal{X}$ будем обозначать через $Tree(\mathcal{X})$.

В работе принято следующее соглашение: все прямолинейные программы обозначаются курсивными буквами, например, $\mathcal{X}$, а строки, которые выводятся из них, соответствующими обычными буквами, например, X. *Размером* ПП $\mathcal{X}$ будем называть размер грамматики и обозначать через $|\mathcal{X}|$. Аналогично, *длину* выводимой строки обозначим через

$|X|$. Через $X[i \dots j]$ мы будем обозначать подстроку строки X, которая начинается в позиции i и заканчивается в позиции j.

Позициями в тексте называются точки между последовательными буквами. Подстрока *касается* данной позиции, если позиция находится внутри или на границе подстроки. Для нетерминальных правил $X_i = X_l \cdot X_r$ определим *позицию разреза* – так называется точка, в которой заканчивается X_l и начинается X_r. А для терминальных правил будем считать, что позиция разреза равна 0.

3 ПП как сжатое представление строк

В качестве алгоритма сжатия мы будем рассматривать разновидность алгоритма Лемпеля-Зива ($LZ77$) без самопересечений (который иногда называется $LZ1$). Интуитивно, LZ алгоритм сжимает исходный текст за счет того, что находит повторяющиеся подстроки. Сжатие LZ алгоритмом определяет естественное разложение сжимаемого текста на предложения (факторы). Пусть $T \in \Sigma^*$, тогда *LZ-факторизация* T задается с помощью разбиения: $T = f_1 \cdot f_2 \cdot \ldots f_k$, где $f_1 = w[1]$ и для каждого $1 \leq i \leq k$ фактор f_i – наибольший префикс $f_i \ldots f_k$, который входит в $f_1 \ldots f_{i-1}$. Обозначим LZ-факторизацию строки T через $LZ(T)$, а ее *размер* положим равным числу факторов и обозначим через $|LZ(T)|$.

Пусть $\mathcal{T}$ – грамматика, порождающая текст T, тогда определим дерево $PTree(\mathcal{T})$ через дерево вывода $Tree(\mathcal{T})$ следующим образом: $PTree(\mathcal{T})$ – максимальное поддерево дерева $Tree(\mathcal{T})$ такое, что каждый его внутренний узел не может быть найден левее в дереве $Tree(\mathcal{T})$. Введем понятие грамматической факторизации T (*G-факторизации*): так мы будем называть последовательность подстрок, порожденную листьями дерева $PTree(\mathcal{T})$. Формально, мы можем определить грамматическую факторизацию так: просматриваем T слева направо, каждый раз выбираем в качестве нового G-фактора наибольший префикс, который порождается нетерминалом, который уже существует левее или букву, если не найдено такого нетерминала.

Пример: Рассмотрим LZ и грамматическую факторизации 7-го слова Фибоначчи.

$$LZ(\text{«}abaababaabaab\text{»}) = f_1\ f_2\ f_3\ f_4\ f_5\ f_6 = a\ b\ a\ aba\ baaba\ ab;$$

$$G(\text{«}abaababaabaab\text{»}) = g_1\ g_2\ g_3\ g_4\ g_5\ g_6\ g_7\ g_8\ = a\ b\ a\ aba\ b\ a\ aba\ ab.$$

На рисунке 1 изображено дерево $PTree(\mathcal{T})$ для некоторой грамматики $\mathcal{T}$ и показано как по этому дереву построить грамматическую факторизацию.

Лемма 1 (о размере ПП)**.**
Пусть $\mathcal{T}$ контекстно-свободная грамматика в нормальной форме Хомского, порождающая единственную строку T, и g – число G-факторов строки T, тогда $|\mathcal{T}| \geq g$.

Доказательство: По грамматике $\mathcal{T}$ мы строим дерево $PTree(\mathcal{T})$, которое является представлением G-факторизации текста T. Факторы, соответствующие различным буквам из Σ, не нуждаются в рассмотрении, так как им соответствуют уникальные терминальные правила из $\mathcal{T}$. По построению дерево $PTree(\mathcal{T})$ таково, что все его внутренние узлы, отличные от терминальных правил, имеют различные метки. Поэтому в $\mathcal{T}$ найдется по крайней мере $g-1$ различных нетерминальных правил (это можно понять, построив G-факторизацию для слова a^n). Ясно, что G-факторизация любой строки содержит хотя бы один фактор единичной длины. В итоге мы получаем, что $|\mathcal{T}| \geq g$. $\square$

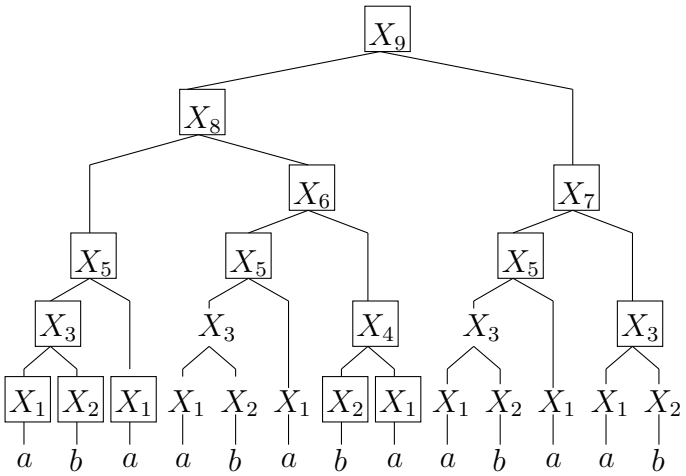


Рисунок 1. Грамматика $\mathcal{T}$, которая выводит «*abaababaabbab*».
Узлы, заключенные в прямоугольники, образуют дерево $PTree(\mathcal{T})$, а их листы задают грамматическую факторизацию строки.

Лемма 2 (о размере ПП)**.**
Для любой строки T и ее грамматического сжатия $\mathcal{T}$ верно, что $|\mathcal{T}| \geq |LZ(T)|$.

Доказательство: Необходимо показать, что число LZ-факторов не больше числа G-факторов. Это следует из интуитивно понятного факта, что LZ-факторизация более жадная, чем G-факторизация. Формально, пусть $f_1 f_2 \dots f_k$ – LZ-факторизация и $g_1 g_2 \dots g_r$ – G-факторизация строки T. Тогда индукцией по i мы можем доказать, что для каждого $i \leq min(k, r)$ верно неравенство: $|g_1 g_2 \dots g_i| \leq |f_1 f_2 \dots f_i|$. Поэтому,

если $r \leq k$, тогда $|g_1 g_2 \dots g_r| \leq |f_1 f_2 \dots f_r|$ и $f_1 f_2 \dots f_r = T$, так как $g_1 g_2 \dots g_r = T$. Аналогично, если $k \leq r$. $\square$

Определение. Введем понятие AVL-*грамматики*, но для начала дадим определение AVL-*дерева*: для каждого узла T_i произвольного дерева $Tree(\mathcal{T})$ определена функция $bal(T_i)$, которая равна разнице между высотами его левого и правого поддеревьев. Дерево $Tree(\mathcal{T})$ называется *AVL-сбалансированным* тогда и только тогда, когда $bal(T_i) \leq 1$ для всех узлов T_i. Будем говорить, что грамматика $\mathcal{T}$ является AVL-*сбалансированной*, если $Tree(\mathcal{T})$ является AVL-сбалансированным. Обозначим через $h(T)$ высоту $Tree(\mathcal{T})$, а через $h(T_i)$ – высоту поддерева $Tree(\mathcal{T}_i)$ с корнем в узле T_i.

Лемма 3 (о высоте сбалансированной ПП)**.**
Если $\mathcal{T}$ – AVL-*сбалансированная грамматика и $|T| = a^n$, где a – некоторая константа, тогда $h(\mathcal{T}) = O(n)$.*

Эта лемма является свойством AVL-сбалансированных деревьев, ее доказательство, например, можно найти в книге Кнута [9] (стр. 469).

Лемма 4 (о конкатенации)**.**
Пусть A, B – два нетерминальных правила AVL-*сбалансированной грамматики. Тогда мы можем сконструировать за время $O(|h(A) - h(B)|)$* AVL-*сбалансированную грамматику $\mathcal{G} = \mathcal{A} \cdot \mathcal{B}$, где $G = A \cdot B$ с помощью добавления не более $O(|h(A) - h(B)|)$ новых нетерминалов.*

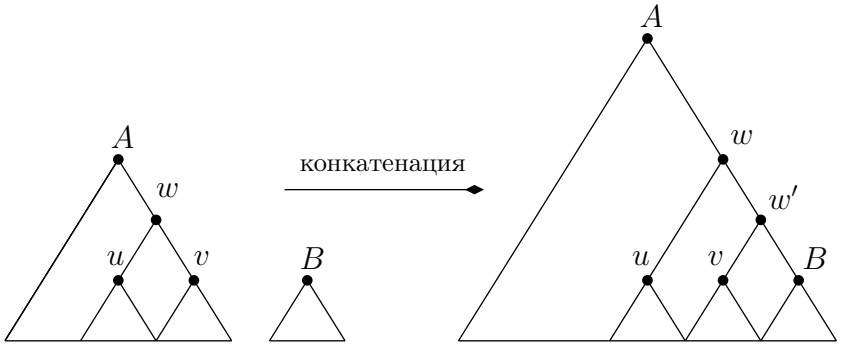
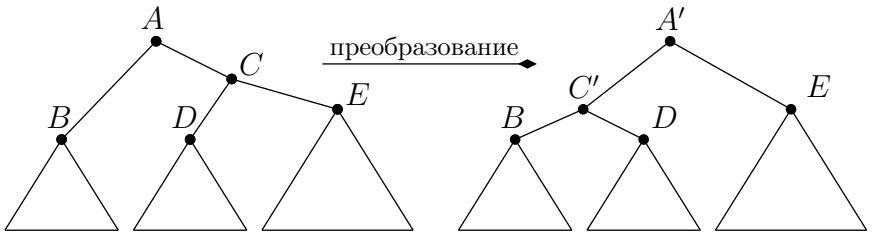


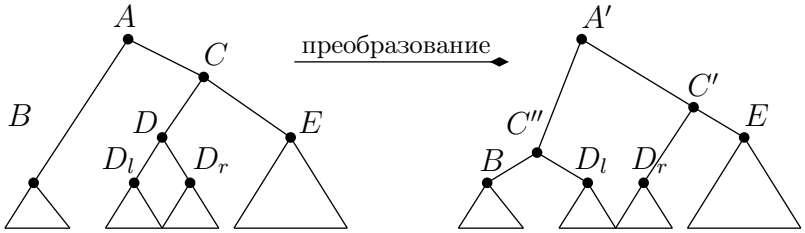
Рисунок 2. Первый шаг алгоритма конкатенации.
Добавляем новую вершину v' в $Tree(\mathcal{A})$, которая может нарушить баланс в дереве.

Доказательство: Мы не будем заострять внимание на деталях алгоритма конкатенации AVL-деревьев. Его описание также можно найти

в книге Кнута [9] (стр. 474). Для нас важнее оценить число новых правил, которые необходимо добавить, чтобы сохранить свойство сбалансированности. Предположим, что $h(\mathcal{A}) \geq h(\mathcal{B})$, другой случай аналогичен. Начиная с корня дерева $Tree(\mathcal{A})$ мы будем спускаться по самой правой ветке. Из свойства сбалансированности следует, что высота дерева сына будет отличаться от высоты дерева отца не более чем на 2. Поэтому мы сможем найти такой узел v, что $h(\mathcal{B}) - h(v) \in \{0, 1\}$. Пусть w – отец v, тогда добавим в дерево $\mathcal{A}$ новый узел v' такой, что v и B – сыновья v', а w – отец v' (см. рисунок 2). Но построенное дерево может оказаться несбалансированным на правой ветке. С помощью локальных преобразований, показанных на рисунке 3, мы можем вернуть баланс в дерево вывода.



$h(C) = h(B) + 2, h(D) = h(E) - 1$, тогда преобразование вернет сбалансированность.



$h(C) = h(B) + 2, h(D) = h(E)$, тогда несбалансированность может перейти на C'.

Рисунок 3. Локальные преобразования

Такие перестановки поддеревьев являются причиной возникновения новых правил. Поэтому нам придется делать копии некоторых старых правил, так как если при перестановки деревьев мы всего лишь изменим соответствующие правила вывода, то это может отразиться на всей структуре грамматики, поскольку мы ничего не знаем о вхождениях этих

правил в других частях грамматики. Нетрудно видеть, что новых правил будет порядка $O(h(\mathcal{A}) - h(\mathcal{B}))$, так как преобразования носят локальных характер. $\square$

Теперь мы покажем как построить грамматику по LZ-факторизации. Пусть $f_1 f_2 \dots f_k$ – LZ-факторизация строки T. Мы преобразуем ее в грамматику, чей размер будет больше размера факторизации на линейный множитель. Предположим, что мы уже построили AVL-сбалансированную грамматику $\mathcal{T}$ размера $O(i \cdot n)$ для префикса $f_1 f_2 \dots f_{i-1}$. Если f_i – терминальный символ, то мы полагаем $\mathcal{G} = \mathcal{G} \cdot \mathcal{A}$, где A – терминальное правило. Иначе мы определяем интервал, которому соответствует положение f_i в префиксе $f_1 f_2 \dots f_{i-1}$. Основываясь на факте, что $\mathcal{T}$ сбалансирована, мы можем найти линейное число нетерминалов $S_1, S_2, \dots, S_{t(i)} \in \mathcal{T}$ таких, что $f_i = S_1 \cdot S_2 \cdot \dots S_{t(i)}$. Последовательность $S_1, S_2, \dots S_{t(i)}$ будем называть грамматическим разбиением фактора f_i. В итоге применяем конкатенацию к этому разбиению и добавляем новый фактор в $\mathcal{T}$.

Изначально определим $\mathcal{T}$ как грамматику порождающую, первый символ T и содержащую все возможные терминальные правила (эти правила не обязательно связаны со строкой T). Алгоритм начинается с вычисления LZ-факторизации, это может быть реализовано с помощью суффиксных деревьев за $O(a^n \log |\Sigma|)$. Если LZ-факторизация достаточно большая (число факторов превышает $\frac{a^n}{n}$), то мы отбрасываем ее и записываем тривиальную грамматику размера a^n, порождающую эту строку. Иначе мы получили только $k \leq \frac{a^n}{n}$ факторов, которые обрабатываются слева направо. Для каждого f_i начиная с $i = 2$ мы строим его грамматическое разбиение $S_1, S_2, \dots S_{t(i)}$, полагаем $\mathcal{P} = \mathcal{S}_1 \cdot \mathcal{S}_2 \cdot \dots \mathcal{S}_{t(i)}$ и переопределяем $\mathcal{T} = \mathcal{T} \cdot \mathcal{P}$.

Грамматическое разбиение и конкатенация $\mathcal{T}$ и $\mathcal{P}$ выполняется за время $O(n)$, так как высота грамматики линейная. По лемме о высоте сбалансированной ПП (см. стр. 9) мы имеем $t(i) = O(n)$, и нам необходимо выполнить $t(i)$ конкатенаций, каждая из которых добавляет $O(n)$ правил. Следовательно, алгоритм дает приближение пропорциональное $O(n^2)$.

Мы можем несколько усложнить выполнение $O(n)$ конкатенаций и за счет этого добиться уменьшения коэффициента с n^2 до n. Ключевое наблюдение: последовательность высот поддеревьев S_i, соответствующих следующему LZ-фактору, является сначала возростающей, а потом убывающей. Поэтому мы можем разбить последовательность на две подпо-

следовательности: неубывающая по высоте $R_1, R_2, \dots R_k$, будем называть ее правосторонней и невозрастающая по высоте $L_1, L_2, \dots L_r$, будем называть ее левосторонней.

Лемма 5 (о конкатенации)**.**
Предположим $R_1, R_2, \dots R_k$ правосторонняя последовательность и $\mathcal{G}_i$ – AVL-грамматика, которая возвращает конкатенацию $R_1, R_2, \dots R_i$ слева направо. Тогда $|h(R_i) - h(G_{i-1})| \leq max\{h(R_i) - h(R_{i-1}), 1\}$.

Доказательство: Мы используем следующий простой факт, верный для любой пары нетерминалов A, B. Обозначим $h = max\{h(A), h(B)\}$, тогда справедливы неравенства:

$$h \leq h(\mathcal{A} \cdot \mathcal{B}) \leq h + 1 \tag{1}$$

Пусть u_i – отец узла, соответствующего R_i. Покажем индукцией по i, что $h(G_i) \leq h(u_i)$. Для $i = 1$ мы имеем $G_i = R_i$, тогда

$$h(u_1) = h(R_1) + 1 \geq g(G_1).$$

Предположим, что неравенство выполнено для $i - 1 : h(G_{i-1}) \leq h(u_{i-1})$. Возможно два случая:

- $h(G_{i-1}) = h(R_i)$. Тогда в соответствии с (1): $h(G_i) = h(G_{i-1} \cdot R_i) \leq h(G_{i-1}) + 1$ и согласно предположению индукции получаем $h(G_i) \leq h(u_{i-1}) + 1 \leq h(u_i)$.
- $h(G_{i-1}) < h(R_i)$. Тогда в соответствии с (1): $h(G_i) = h(G_{i-1} \cdot R_i) \leq h(R_i) + 1 \leq h(u_i)$.

Вернемся к доказательству основного утверждения. Если $h(G_{i-1}) \geq h(R_i)$, тогда

$$|h(R_i) - h(G_{i-1})| = h(G_{i-1}) - h(R_i) \leq h(u_{i-1}) - h(R_i) \leq 1.$$

Последнее неравенство следует из свойства AVL-сбалансированности. Если $h(G_{i-1}) \leq h(R_i)$, тогда $|h(R_i) - h(G_{i-1})| = h(R_i) - h(G_{i-1}) \leq h(R_i) - h(R_{i-1})$, так как $h(G_{i-1}) \geq h(R_{i-1})$. $\square$

Теорема 1 (о приближенном грамматическом сжатии)**.**
За время $O(a^n \log |\Sigma|)$ мы можем построить грамматику, чей размер

будет отличаться от размера минимальной грамматики на линейный множитель. По заданной LZ-факторизации длины k мы можем построить соответствующую ей грамматику размера $O(k \cdot n)$ за время $O(k \cdot n)$.

Доказательство: Новый фактор f_i разбивается на интервалы $S_1, S_2, \dots S_{t(i)}$. Достаточно показать, что мы можем за время $O(n)$ построить AVL-грамматику для конкатенации $S_1, S_2, \dots S_{t(i)}$ с добавлением $O(n)$ новых правил в $\mathcal{G}$, предполагая что грамматики для $S_1, S_2, \dots S_{t(i)}$ уже построены.

Последовательность $S_1, S_2, \dots S_{t(i)}$ содержит левостронную и правостороннюю подпоследовательности. Грамматики $\mathcal{H}', \mathcal{H}''$ соответствующие этим подпоследовательностям (с добавлением линейного числа правил) вычисляются по лемме о конкатенации (см. стр. 12) и вычисляем конкатенацию $\mathcal{H}'$ и $\mathcal{H}''$. Пусть $R_1, R_2, \dots R_k$ – правосторонние поддеревья. Тогда время и число дополнительных правил, необходимых для конкатенации $R_1, R_2, \dots R_k$, могут быть оценены следующим образом:

$$\sum_{i=2}^{k} |h(R_i) - h(G_{i-1})| \leq \sum_{i=2}^{k} max\{h(R_i) - h(R_{i-1}), 1\}$$

$$\leq \sum_{i=2}^{k} (h(R_i) - h(R_{i-1})) + \sum_{i=2}^{k} 1 \leq h(R_k) + k = O(n)$$

Аналогичные рассуждения применимы для левосторонней подпоследовательности. В итоге, обрабатывая каждый фактор, f_i мы увеличиваем грамматику на $O(n)$ правил и тратим $O(n)$ времени. Если наша цель добиться линейной пропорциональности, то мы рассматриваем случай, когда число факторов k порядка $O(a^n/n)$. Если LZ-факторизация вычислена, то общее время и размер построенной грамматики равны $O(k \cdot n)$. □

В качестве следствия мы получаем такое утверждение:

Следствие. Пусть $\mathcal{G}$ – грамматика (ПП) размера k, порождающая единственную строку длины a^n. тогда мы можем построить за время $O(k \cdot n)$ эквивалентную грамматику высоты $O(n)$.

Замечание. Все утверждения в этом разделе взяты из работы В. Риттера [29].

4 Простейшие операции над ПП

В этом разделе мы представим несколько простейших операций, которые часто используются при работе со сжатыми строками. Далее мы будет неявно использовать эти операции при решении более сложных задач. Пусть нам дана ПП $\mathcal{T}$ размера n.

- **Вычисление массива длин правил.** Пробегаем по всем правилам грамматики, начиная с правил меньшей длины. Длину терминальных правил полагаем равной 1. Длину нетерминального правила вычисляем как сумму длин левого и правого сыновей. За $O(n)$ мы построим массив всех длин правил из $\mathcal{T}$.

- **Вычисление массива позиций разреза.** Алгоритм аналогичен вычислению массива длин. Для каждого терминального правила полагаем позицию разреза равной 0. Для нетерминального правила позиция разреза равна длине левого правила. На заполнение массива длин нам потребуется $O(n)$ времени.

- **Вычисление массива первых/последних букв.** Покажем как вычислить массив первых букв. Пробегаем по всем правилам грамматики, начиная с правил меньшей длины. Для терминального правила храним символ, который выводит это правило. Для нетерминального правила храним первый символ левого правила (он уже вычислен). Поэтому на заполнение всего массива нам потребуется $O(n)$ времени.

- **Взятие грамматики для подстроки.** Нам заданы позиции k_1, k_2 – начала и конца подстроки в T. Необходимо построить подграмматику $\mathcal{T}$, которая бы выводила в точности строку $T[k_1 \dots k_2]$. Найдем наименьшее по длине правило $T_i \in \mathcal{T}$, которое полностью содержит строку $T[k_1 \dots k_2]$. Спускаемся из корня $\mathcal{T}$ пока не найдем правило T_i такое, что $T[k_1 \dots k_2]$ касается позиции разреза и оно является наименьшим по длине. Возьмем подграмматику, которая порождает это правило. Будем просматривать $\mathcal{T}$, начиная с правил меньшей длины, пока не встретим T_i, и каждому правилу ставим в соответствие массив правил, которые выводятся из него. Терминальному правилу T_0 ставим в соответствие T_0. Нетерминальному правилу

$T_j = T_l \cdot T_r$ ставим в соответствие объединение (без повторов) массивов правил, порождающих T_l и T_r, и правила T_j. За $O(n^2)$ мы построим множество правил, которые порождают T_i. Нам необходимо преобразовать $\mathcal{T}_i$ так, чтобы она выводила в точности строку $T[k_1 \dots k_2]$. Покажем как осуществить коррекцию левого края дерева вывода $\mathcal{T}_i$ (аналогично осуществляется коррекция правого края). Идем сверху вниз по ветке $\mathcal{T}_i$, которая ведет в позицию k_1, и помечаем этот путь. Если нам встретилось правило, которое содержит позицию k_1 в левой части, тогда ставим метку 0. А если правило содержит k_1 в правой части, то ставим метку 1. Теперь будем возвращаться по этому пути и добавлять новые правила. Ищем первое правило $T_{i_1} = T_{i_L} \cdot T_{i_R}$, которое было помечено 0, и берем последнее правило в пути T_{j_1} (терминальное). Добавляем новое правило $T'_{i_1} = T_{j_1} \cdot T_{i_R}$. Теперь возьмем произвольное правило $T_{i_k} = T_{i_L} \cdot T_{i_R}$, которое помечено 0, и найдем ближайшее новое правило $T'_{i_{k-1}}$. Тогда добавляем новое правило: $T'_{i_k} = T'_{i_{k-1}} \cdot T_{i_R}$. В итоге мы добавим не более $O(n)$ новых правил и нам требуется $O(n^2)$ времени.

- **Обращение грамматики.** Пусть нам задана ПП $\mathcal{T}$, которая выводит текст T. Мы хотим построить ПП $\mathcal{T}^R$, которая бы выводила строку T^R. Просматриваем $\mathcal{T}$, начиная с правил меньшей длины. Все терминальные правила из $\mathcal{T}$ мы переносим без изменений в $\mathcal{T}^R$. Предположим, что вы взяли очередное нетерминальное правило $T_i = T_l \cdot T_r$ и правила T_l^R, T_r^R уже вычислены. Тогда мы добавляем новое правило $T_i^R = T_r^R \cdot T_l^R$ в $\mathcal{T}^R$. Ясно, что полученная грамматика $\mathcal{T}^R$ выводит текст T^R. И нам требуется $O(n)$ времени.

Замечание. Все приведенные выше алгоритмы считаются общеизвестными и не принадлежат какому-любо автору.

5 Полиномиально разрешимые задачи

В этом этой главе мы рассмотрим классические строковые задачи, которые являются полиномиально разрешимыми в терминах сжатых строк.

5.1 Задача Поиск сжатого образца в сжатом тексте

Задача: **Поиск сжатого образца в сжатом тексте**

Вход: ПП $\mathcal{P}$ и $\mathcal{T}$, которые выводят строки P и T соответсвенно, при этом $|\mathcal{P}| = m, |\mathcal{T}| = n$ и $|P| < |T|$.

Выход: Множество всех позиций текста T, начиная с которых образец P входит в текст T, представленное в сжатом виде.

5.1.1 Идея алгоритма

- **Локализация** Зафиксируем правило $T_i \in \mathcal{T}$ и его позицию разреза γ. Будем искать все вхождения образца в строке T_i, которые касаются γ. Поскольку позиции разреза покрывают множество всех позиций текста, то ни одного вхождения образца в тексте не будет пропущено. Другими словами, если мы зафиксируем некоторое вхождение образца P в тексте T, то мы сможем найти правило $T_i \in \mathcal{T}$, которое полностью содержит образец и P касается γ. Алгоритм поиска такого правила очень прост: пусть T_n – корень дерева $Tree(\mathcal{T})$ и $T_n = T_l \cdot T_r$. Если P касается позиции разреза T_n, то мы останавливаемся. Иначе P полностью содержится левее или правее позиции разреза и мы переходим либо к T_l, либо к T_r. Далее проделываются аналогичные шаги. Поскольку высота дерева вывода порядка $O(n)$, то за линейное число шагов алгоритм остановится.

- **Эффективная структура данных** Следующая лемма представляет идею, которую мы будем использовать для эффективного хранения вхождений образца вокруг позиции разреза некоторого правила:

  Лемма 6 (основная)**.**
  Все вхождения P в T, касающиеся некоторой фиксированной позиции в тексте, образуют арифметическую прогрессию.

Введем понятие *таблицы арифметических прогрессий*(кратко AP-таблица): Для любой пары $1 \leq i \leq m, 1 \leq j \leq n$ определим значение $AP[i,j]$ как сжатое представление арифметической прогрессии тех вхождений P_i в T_j, которые касаются позиции разреза T_j. Арифметическую прогрессию мы храним в виде тройки: $\langle a, d, t \rangle$, где a – начало, d – шаг, t – число элементов прогрессии. Если $|T_j| < |P_i|$, тогда положим $AP[i,j] = 0$. Если $|T_j| \geq |P_i|$, но не найдено ни одного вхождения P_i вокруг позиции разреза T_j, то положим $AP[i,j] = \varnothing$.

Ясно, что если мы сможем корректно заполнить AP-таблицу, то в строке таблицы, соответствующей правилу P_m, будет содержаться информация о всех вхождениях P в T. С помощью AP-таблицы мы можем ответить на такие запросы: определить входит ли P в T как подстрока, найти первое вхождение образца в тексте, найти количество всех вхождений, проверить есть ли вхождение начиная с данной позиции. Например, покажем как по AP-таблице вычислить количество всех вхождений P в тексте. Индукция по всем правилам из $\mathcal{T}$. Для однобуквенных текстов мы берем только мощность соответствующей арифметической прогрессии. Для произвольного правила $T_i = T_l \cdot T_r$ число вхождений P в T_i равно сумме числа всех вхождений P в T_l, числа всех вхождений P в T_r и числа всех вхождений, касающихся позиции разреза T_i, без учета только касающихся вхождений P.

5.1.2 Техническая реализация

Все шаги алгоритма делаются индуктивно, начиная с правил, соответствующих текстам меньшей длины.

- **Предварительные вычисления** За время $O(m+n)$ мы можем вычислить массивы длин, позиций разрезов, первых/последних букв для текстов $P_1, \ldots, P_m, T_1, \ldots, T_n$ (детали в разделе Простейшие операции над ПП).

- **Заполнение строк/столбцов, соответствующих терминальным правилам** Зафиксируем пару P_i, T_j. Пусть $|P_i| = 1$, тогда, если $|T_j| = 1$, мы распаковываем два символа и сравниваем их. Результат сравнения записываем в виде арифметической прогрессии в $AP[i,j]$. Иначе $|T_j| > 1$ и мы распаковываем два символа $T_j[\gamma]$ и

$T_j[\gamma+1]$, где γ – позиция разреза T_j и сравниваем их с символом $P_i[1]$. Результат сравнения представляем в виде арифметической прогрессии и сохраняем его в $AP[i,j]$.

Пусть теперь $|T_j| = 1$. Тогда для всех правил $|P_i| > 1$ мы полагаем $AP[i,j] = 0$, иначе сравниваем пару и символов и результат сохраняем в $AP[i,j]$.

- **Вычисляем произвольный элемент AP-таблицы.** Зафиксируем произвольные правила $P_i = P_l \cdot P_r, T_j$ такие, что $|P_i|, |T_j| > 1$. Без ограничения общности будем считать, что $|P_l| \geq |P_r|$. Предполагается, что значения $AP[i,j]$ для $i' < i$ и $j' < j$ уже вычислены. Зафиксируем позицию разреза γ правила T_j.

  Нетрудно понять, что любое вхождение P_i, касающееся позиции γ, состоит из вхождения P_l и вхождения P_r внутри $|P_i|$-окрестности позиции разреза. При этом P_l заканчивается в точке, откуда начинается P_r. Поэтому правило поиска вхождения состоит из двух шагов:

  1. найти все вхождения P_l "вокруг" γ;
  2. найти все вхождения P_r, которые начинаются с окончаний вхождений P_l.

  При вычислении нового элемента AP-таблицы мы будем использовать функцию $LSearch(i,j,\alpha,\beta)$, которая возвращает вхождения P_i в T_j, которые целиком лежат внутри интервала (α,β). Реализация функции $LSearch$ представлена ниже, здесь мы лишь отметим ее важные свойства:

  1. $LSearch$ использует значения $AP[i,k]$, где $1 \leq k \leq j$;
  2. Работает корректно при условии: $\beta - \alpha \leq 3|P_i|$;
  3. Работает за время $O(j)$;
  4. На выходе локальный поиск выдает пару арифметических прогрессий, причем внутри каждой прогрессии все вхождения имеют общую точку и все вхождения из первой прогрессии находятся левее всех вхождений из второй;

Теперь покажем, как вычислить значение $AP[i, j]$ за 5 запусков локального поиска.

Ищем большую часть P_i. За один вызов $LSearch$ мы найдем все вхождения P_l в интервале $(\gamma - |P_i|, \gamma + |P_l|)$, так как $|P_i| + |P_l| \leq 3|P_l|$. В результате мы получим две прогрессии, представляющие все потенциальные стартовые позиции для вхождений P_i, касающихся разреза. Однако, мы не можем проделать аналогичные шаги с P_r, так как длина соответствующего интервала поиска может не быть константой относительно $|P_r|$. Поэтому обе прогрессии сдвигаем на $|P_l|$ позиций вправо и получаем множество окончаний вхождений P_l. И будем искать только те вхождения P_r, которые начинаются с позиций из сдвинутых прогрессий.

Ищем меньшую часть P_i. Будем обрабатывать каждую прогрессию по отдельности. Назовем точку окончания вхождения P_l *континентальной*, если она находится на расстоянии не менее $|P_l|$ от последнего окончания в прогрессии. Иначе окончание будем называть *приморским*. По свойству 4 функции $LSearch$ все вхождения P_l из одной прогрессии имеют общую точку. Следовательно, все подстроки длины $|P_r|$, стартующие с континентальных окончаний, полностью совпадают. Поэтому нам достаточно проверить все приморские точки и одну континентальную. Для проверки континентального окончания мы применим $LSearch$ к $|P_r|$-строке, начинающейся с этого окончания. Еще за один вызов $LSearch$ мы обработаем все приморские окончания (запускаем $LSearch$ для строки P_r в $|P_r|$-окрестности последнего окончания) и пересечем ответ с прогрессией приморских окончаний вхождений P_l. Пересечение двух прогрессий занимает $O(n)$ шагов.

Приводим ответ к одной прогрессии. По основной лемме (стр. 16) мы знаем, что все вхождения P_i, касающиеся разреза T_j образуют одну арифметическую прогрессию, поэтому мы можем взять все результаты с предыдущего шага и привести их к одной прогрессии.

Сложность вычисления элемента таблицы: мы применили $LSearch$ для строки P_l, четыре раза применили $LSearch$ для P_r и дважды вычисляли пересечение арифметических прогрессий. Таким образом, мы имеем семь участков по $O(n)$ шагов.

5.1.3 Функция *LSearch*

Напомним задачу, которую решает локальный поиск:

Вход: заданы номера правил i, j и область поиска (α, β)

Выход: все вхождения P_i в T_j, которые целиком лежат внутри интервала (α, β), представленные в виде пары арифметических прогрессий.

Алгоритм: В логике работы алгоритма можно выделить два этапа: **Обход** – рекурсивная процедура, которая на вход получает четверку параметров (i, j, α, β). На каждой итерации мы выполняем следующие шаги:

1) берем из AP-таблицы вхождения P_i в T_j, которые касаются разреза;

2) обрезаем прогрессию этих вхождений, оставляя лишь полностью содержащиеся внутри (α, β) и записываем ее в список ответов;

3) проверяем, имеет ли пересечение (α, β) с левой/правой частью T_j длину хотя бы $|P_i|$. Если имеет, тогда делаем рекурсивный вызов процедуры обхода с параметрами i, индексом левой/правой части и интервалом пересечения;

Нам надо понять глубину рекурсии вызовов процедуры обхода. Рассмотрим положение всех интервалов, с которыми мы работали во время процедуры обхода, в тексте T_j. Любая пара из них или вложена, или не пересекается. При этом для каждого k интервалы, на которых процедура обхода запускалась с параметром T_k, не могут быть вложенными. Поскольку мы требуем, чтобы исходный интервал был не больше $3 \cdot |P_i|$, то для каждого k во время обхода было не больше трех интервалов, ассоциированных с T_k. Таким образом, при обходе мы сделали не более $3j$ рекурсивных вызовов и общее время работы не превосходит $O(j)$.

Мы предполагаем, что процедура обхода поддерживает указатель на соответствующее место в списке ответов. Это означает, что в конце работы мы получим отсортированный список из не более чем $3j$ прогрессий. Здесь "отсортированный означает", что последний элемент k-й прогрессии не больше первого элемента $k+1$ прогрессии.

Второй этап – **упрощение**. Разобьем интервал (α, β) на три равные части и пусть $\delta_1 = \frac{2\alpha+\beta}{3}$, $\delta_2 = \frac{\alpha+2\beta}{3}$ – позиции разбиения. По Лемме(основной) мы знаем, что все вхождения P_i в (α, β) образуют две арифметические прогрессии. А именно, вхождения, касающиеся δ_1, и вхожде-

ния, касающиеся δ_2, но не касающиеся δ_1. Здесь мы используем неравенство $\beta - \alpha \leq 3|P_i|$, из которого следует, что каждое вхождение P_i должно задеть δ_1 или δ_2. Поэтому в процедуре упрощения мы просто проходим по всему списку прогрессий и будем сравнивать расстояние между последним элементом текущей прогрессии и первым элементом последующей с шагом каждой из этих прогрессий. Если все три числа равны, тогда мы объединяем прогрессии. Иначе мы объявляем новую прогрессию. Из аргументов, приведенных выше, мы получаем, что новая прогрессия не может быть объявлена более одного раза.

5.1.4 Сложность алгоритма поиска сжатого образца

На этапе предварительных вычислений нам требуется $O(n+m)$ времени и столько же места. Далее для каждой пары индексов $1 \leq i \leq m, 1 \leq j \leq n$ мы вычисляем элемент AP-таблицы. Т.е. на заполнение всей AP-таблицы нам требуется $O(n^2m)$ времени и $O(nm)$ пространства. В итоге, $O(n^2m)$ – время работы алгоритма, $O(nm)$ – пространство необходимое алгоритму.

Замечание Приведенный алгоритм взят из работы Ю. Лифшица [26].

5.2 Задача Вычисление таблицы перекрытий

Определение. Для строк P и T определим *множество перекрытий* между P и T: $OV(P,T) = \{k > 0 : P[|P| - k + 1 \dots |P|] = T[1 \dots k]\}$. Тогда под таблицей перекрытий для ПП $\mathcal{T}$ мы будем понимать таблицу размера $n \times n$, которая для каждой пары $1 \leq i, j \leq n$ хранит множество $OV(T_i, T_j)$, представленное в сжатом виде (непосредственно представление мы опишем позднее). Теперь мы можем сформулировать задачу **Вычисление таблицы перекрытий**:

Вход: ПП $\mathcal{T}$, представляющая текст T;

Выход: таблица перекрытий;

5.2.1 Ключевые идеи

- **Хранение таблицы перекрытий.**

  Определение. Неотрицательное p будем называть *периодом* строки T, если $T[i] = T[i-p]$ для всех определенных пар $i, i-p$. Мно-

жество всех периодов строки будем обозначать через $Periods(T)$.

Определение. Для двух числовых множеств A и B введем операцию: $A \oplus B = \{a + b : a \in A, b \in B\}$. В случае, когда $A = \{a\}$, будем писать $a \oplus B$ вместо $\{a\} \oplus B$.

Лемма 7 (теорема Файна-Вилфа)**.**
Пусть p, q – периоды строки T и $p + q \leq |T|$, тогда НОД(p, q) является периодом T.

Определение. Будем говорить, что множество $\{0, 1, \ldots, N\}$ *линейно сжимаемо* относительно N, если его можно разбить на не более $\lfloor \log_2 N \rfloor + 1$ арифметических прогрессий.

Лемма 8 (о сжимаемости)**.**
Множество $Periods(T)$ линейно сжимаемо относительно $|T|$.

Доказательство: Индукция по параметру $j = \lfloor \log_2 |T| \rfloor$.

База: $j = 0$, буква имеет периоды 0 и 1 (образующие единственную арифметическую прогрессию), следовательно мы имеем в точности $\lfloor \log_2 |T| \rfloor + 1$ прогрессий.

Шаг: Пусть $k = \lceil \frac{|T|}{2} \rceil$ (т.е. мы зафиксировали некоторое значение j и по нему вычислили значение k). Рассмотрим множество $U = Periods(T) \cap \{0, 1, \ldots k\}$. Если p_1, p_2 – произвольные элементы U, тогда $p_1 + p_2 \leq |T|$ и по теорема Файна-Вилфа, мы получаем, что НОД(p_1, p_2) – период T. Поэтому U является арифметической прогрессией с шагом равным наибольшему общему делителю всех значений из множества U. Пусть q – наименьший период больший k. Не трудно видеть, что $Periods(T) = U \cup (q \oplus Periods(T[q + 1 \ldots |T|]))$. Действительно, пусть $p \in Periods(T[q + 1 \ldots |T|])$. Покажем, что $T[i] = T[i + p + q]$ для всех $i \in \{1 \ldots |T| - p - q\}$. $T[i] = T[i + q]$, так как q – период T. А $T[i + p] = T[i + q + p]$, так как p – период $T[q + 1 \ldots |T|]$. Поскольку $\lfloor \log_2(|T| - q) \rfloor < j$, то мы можем применить предположение индукции, а множество U является арифметической прогрессией, следовательно утверждение верно. □

Ясно, что множество $OV(T_i, T_j)$ может быть представлено парой $max = max\{i : i \in OV(T_i, T_j)\}$ и $Periods(T_j[1 \ldots max])$, так как $OV(T_i, T_j) = \{max - i : i \in Periods(T_j[1 \ldots max])\} \setminus \{0\}$. На са-

мом деле мы будем хранить множество периодов в упорядоченном виде: Пусть a_1 – наименьший период, который всегда 0, а d_1 – разница между a_1 и следующим периодом по возрастанию. Для $i > 1$ положим a_i первый период, который не содержится в множестве $\bigcup_{k=0}^{i-1}\{a_k + j \cdot d_k : j \geq 0\}$, а d_i равно разнице между a_i и предыдущим периодом. Каждая пара $< a_i, d_i >$ представляет отрезок $\{a_i + j \cdot d_i : 0 \leq j$ и $a_i + j \cdot d_i < a_{i+1}\}$ множества периодов. Пары $\langle a_k, d_k \rangle$ мы храним отсортированными относительно a_k.

Лемма 9 (о запросе перекрытия)**.**
Если множество перекрытий между T_i и T_j вычислено и представлено в сжатом виде, то на извлечение информации о перекрытии между T_i и T_j требуется $O(\log n)$ времени.

Доказательство: Сначала мы ищем отрезок, которому принадлежит наше значение. Это реализуется с помощью бинарного поиска по значениям a_i и на это требуется $O(\log n)$ времени, так как у нас не более n отрезков. Далее за константное время мы можем определить, принадлежит ли наше значение прогрессии. □

- **Логика вычислений**

  Таблицу перекрытий будем заполнять, начиная с правил меньшей длины к большим. Для пары терминальных правил мы будем вычислять величину их перекрытия по определению. Введем операцию *расширения префикса* $PrefixExtension(U, T_i, T_j)$ $(PExt(U, i, j))$, где $U \subseteq \{1, \ldots, n\}$: $PExt(U, i, j) = \{k + |T_j| : k \in U, T_i[1 \ldots k] \cdot T_j$ – префикс $T_i\}$. Тогда, если $T_i = T_l \cdot T_r$ и положить $U = OV[T_l, T_j]$, а $W = OV[T_j, T_r]$, нетрудно видеть, что

  $$OV[T_i, T_j] = PExt(U, j, r) \cap W.$$

  Поэтому нам остается научиться эффективно вычислять функцию $PExt$.

5.2.2 Эффективное вычисление функции $PExt$

Определение. Пусть $|T_j| > k$ и $0 \leq k \leq |T_i|$, тогда

$$FirstMismatch(T_i, T_j, k)(FM) = min\{i > 0 : T_i[|T_i| - k + i] \neq T_j[i]\}$$

Если такого i не существует полагаем, что $FM(T_i, T_j, k) = 0$.

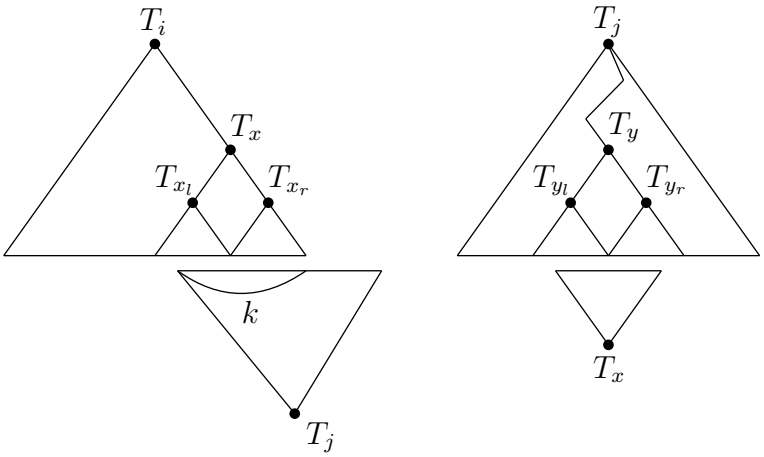


Рисунок 4. Иллюстрация одного шага поиска первого несовпадения

Лемма 10 (о связи с перекрытием)**.**
Пусть $T_i, T_j \in \mathcal{T}$, тогда значение $FM(i, j, k)$ может быть вычислено с помощью $O(n)$ запросов на перекрытие между предшественниками T_i и T_j.

Доказательство: Рассмотрим $Tree(\mathcal{T}_i)$ и $Tree(\mathcal{T}_j)$. Мы будем использовать идею бинарного поиска для того, чтобы спускаться в обоих деревьях. Итак, нам надо вычислить первое несовпадение в перекрытии между T_i и T_j. Используя информацию о длинах правил, мы можем найти первое правило T_x в дереве $Tree(\mathcal{T}_i)$ такое, что позиция $|T_x| - k$ находится левее позиции разреза T_x (т.е. $|T_x| - k < |T_{x_l}|$). Проверим, принадлежит ли $k - |T_{x_l}|$ множеству $OV(T_{x_l}, T_j)$. Если не принадлежит, то первое несовпадение находится в правиле T_{x_l}, и мы запускаем рекурсивно $FM(x_l, j, k - |T_{x_r}|)$. Если принадлежит, то несовпадение находится где-то внутри T_{x_r}. Ищем в дереве $Tree(\mathcal{T}_j)$ первое правило T_y такое, что T_{y_l}, T_{y_r} образуют перекрытия с T_{x_r} (т.е. правило T_y полностью охватывает правило T_{x_r}). Мы проверяем существование перекрытия между T_{y_l} и

T_{x_r}. Если его не существует, то первое несовпадение находится в этом перекрытии, и мы запускаем поиск первого несовпадения между T_{y_l} и T_{x_r}. Иначе первое несовпадение находится в перекрытии между T_{x_r} и T_{y_r}, и мы запускаем поиск первого несовпадения в этом перекрытии.

Ясно, что на каждой итерации мы спускаемся хотя бы на один уровень в одном из деревьев вывода, а поскольку высота грамматики $\mathcal{T}$ равна $O(n)$, то наш алгоритм остановится и при этом мы запустим не более $O(n)$ запросов перекрытия. □

Лемма 11 (о продолжении периодичности)**.**
Предположим, что таблица перекрытий вычислена для всех предшественников правила T_i и $T_i = T_l \cdot T_r$, где строка T_l периодическая с периодом p. Тогда продолжение периода p в T_r можно вычислить за время $O(n \log n)$.

Доказательство:

Сначала мы ищем несовпадение в перекрытии между T_l и T_r с помощью запуска $FM(T_l, T_r, p)$. Если $FM(T_l, T_r, p) = 0$, то ищем продолжение периодичности в перекрытии между T_r и T_r с помощью запуска $FM(T_r, T_r, |T_r| - p)$. □

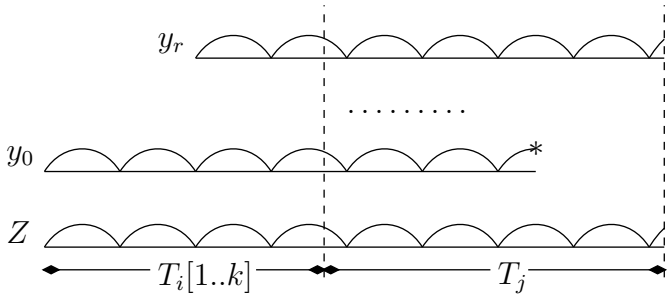
Теорема 2 (о таблице перекрытий)**.**
Пусть $T_i, T_j \in \mathcal{T}$ и таблица перекрытий вычислена для всех предшественников этих правил. Пусть $S = \{t_0, t_1, \ldots, t_s\} \subseteq \{1, \ldots, k\}$ – арифметическая прогрессия. Тогда значение $PExt(S, i, j)$ может быть вычислено за $O(n \log n)$.

Доказательство:

Пусть $t_0, t_1, \ldots, t_s$ – убывающая последовательность. Нам необходимо вычислить все возможные продолжения $x_l = T_i[1 \ldots t_l]$ в T_i, которые совпадают с T_j. Обозначим $y_l = T_i[1 \ldots |x_l| + |T_j|]$ и $Z = T_i[1 \ldots k] \cdot T_j$. Тогда нам необходимо найти все индексы i такие, что y_i – суффикс Z ($0 \le i \le s$). Мы будем называть такие i подходящими индексами. Пусть p – шаг S, тогда p – период $T_i[1 \ldots k]$. Вычислим продолжение p-периодичности в Z и в y_0 по лемме о продолжении периодичности (см. стр. 25). Возможны следующие случаи:

1. **Нет обрыва периодичности в Z, но есть обрыв в y_0**

   Тогда все индексы $i \geq r$ будут подходящими, где r – первый индекс такой, что y_r является периодической строкой.

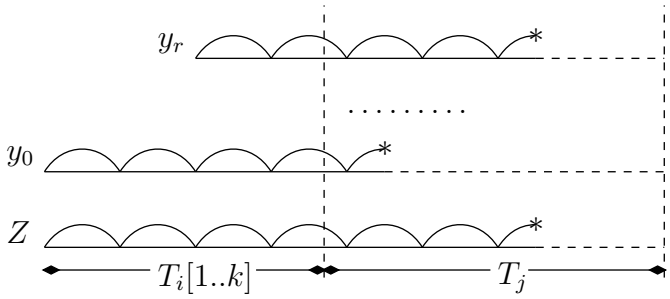


2. **Нет обрыва периодичности ни в Z, ни в y_0**, тогда все индексы подходящие.

3. **Есть обрыв периодичности в Z, но нет в y_0**, тогда ни один из индексов не является подходящим.

4. **Есть обрыв периодичности и в Z и в y_0**

   Тогда может существовать единственный хороший индекс i такой, что первое несовпадение в y_i находится в точности под первым несовпадением в Z. Мы можем легко вычислить такой i или показать, что его не существует. Если мы нашли такой индекс, тогда нам необходимо проверить равенство "хвостов" Z и y_i. Это можно осуществить с помощью функции FM.



Заметим, что множество хороших индексов является подпрогрессией S. □

5.2.3 Оценка сложности

В каждой клетке таблицы перекрытий мы храним $O(n)$ арифметических прогрессий. Поэтому при вычислении нового элемента таблицы нам необходимо применить $O(n)$ раз операцию $PExt$. Значит сложность вычисления одного элемента таблицы равна $O(n^2 \log n)$. Следовательно, время необходимое нам для вычисления всей таблицы равно $O(n^4 \log n)$ и требуется $O(n^3)$ пространства для ее хранения.

Таблица перекрытий является вспомогательной конструкцией, которую можно использовать для поиска различных объектов. Например, ее можно использовать для поиска образца в тексте, основываясь на таком наблюдении: P входит в текст T тогда и только тогда, когда $\mathcal{T}$ содержит правило $T_i = T_l \cdot T_r$ такое, что для некоторого t $P[1 \dots t]$ является суффиксом T_l и $P[t+1 \dots |P|]$ является префиксом T_r, т.е. $|P| \in OV(T_l, P) \oplus OV(P, T_r)$. Также таблица перекрытий используется для поиска наибольшей подстроки между двумя сжатыми строками. Эту задачу мы рассмотрим подробно ниже.

Замечание. Приведенный алгоритм взят из работы М. Карпински, В. Риттера и А. Шинохары [13].

5.3 Задача Наибольшая общая сжатая подстрока

Задача: **Наибольшая общая сжатая подстрока**

Вход: ПП $\mathcal{P}$ и $\mathcal{T}$, которые выводят тексты P и T соответственно

Выход: ПП, которая выводит наибольшую общую подстроку текстов P и T

5.3.1 Логика поиска подстроки

Мы можем осуществить итерацию по правилам грамматик $\mathcal{P}$ и $\mathcal{T}$, основываясь на следующей идее: для любой подстроки Z строки T всегда существует правило $T_i \in \mathcal{T}$ такое, что Z является подстрокой T_i и касается позиции разреза этого правила. Из этого следует, что наибольшая общая подстрока строк P и T полностью содержится в правилах P_i и T_j для некоторых $1 \le i \le m, 1 \le j \le n$ и касается позиций разреза этих правил. Поэтому нам достаточно уметь находить наибольшую общую подстроку пары правил.

Зафиксируем произвольные правила $P_i = P_{l_i} \cdot P_{r_i}$ и $T_l = T_{l_j} \cdot T_{r_j}$ и пусть

$k \in OV(P_i, T_j)$, тогда определим операцию $Ext(P_i, T_j, k) = k + h_1 + h_2$, где $h_1 = LCSuf(P_{l_i}[1 \ldots |P_{l_i}| - k], T_{l_j})$ и $h_2 = LCPref(P_{r_i}, T_{r_j}[k+1 \ldots T_{r_j}])$, где $LCSuf$ – *операция взятия наибольшего общего суффикса* между двумя ПП, а $LCPref$ – *операция взятия наибольшего общего префикса*. Для любого $k \in OV(T_j, P_i)$ можно определить значение $Ext(T_j, P_i, k)$ аналогично. Тогда длина наибольшей общей подстроки между P и T равняется максимальному элементу в объединении:

$$\bigcup_{1 \le i \le m, 1 \le j \le n} (Ext(P_i, T_j, OV(P_{i_l}, T_{j_r})) \cup Ext(T_j, P_i, OV(T_{l_j}, P_{r_i})) \cup LCStr(P_i, T_j))$$

$Ext(P_i, T_j, OV(P_{i_l}, T_{j_r})) = \{Ext(P_i, T_j, k) : k \in OV(P_{i_l}, T_{j_r}))\}$,
$Ext(T_j, P_i, OV(T_{l_j}, P_{r_i})) = \{Ext(T_j, P_i, k) : k \in OV(T_{j_l}, P_{i_r}))\}$,
$LCStr(P_i, T_j) = LCSuf(P_{i_l}, T_{j_l}) + LCPref(P_{i_r}, T_{j_r})$

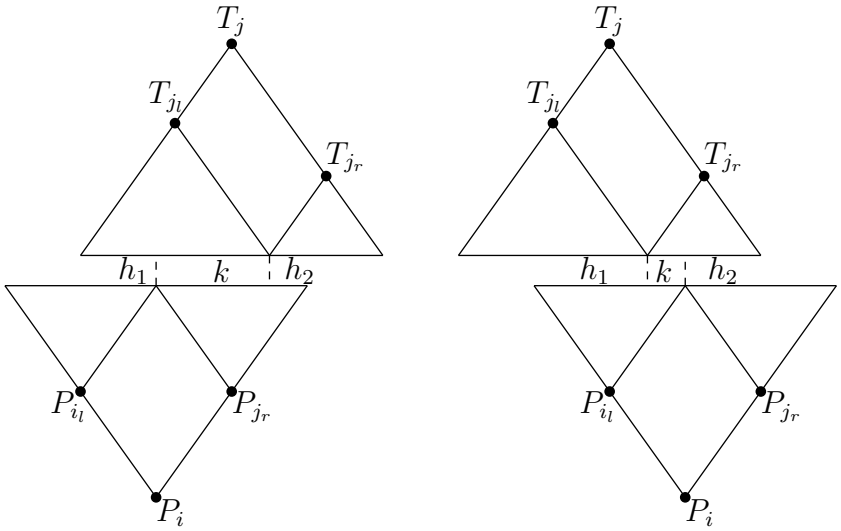


Рисунок 4. Возможные выравнивания пары правил

Ясно, что эти множества содержат длины всех возможных расширений пары строк P_i, T_j. Поэтому, нам достаточно научиться вычислять $max(Ext(P_i, T_j, OV(P_{i_l}, T_{j_r})))$ и $max(Ext(T_j, P_i, OV(T_{l_j}, P_{r_i})))$. Следующая лемма показывает как вычислить максимальный элемент этих двух множеств с помощью функции FM.

Лемма 12 (о максимизации расширения)**.**
Для любых правил $P_i = P_{i_l} \cdot P_{i_r}, T_j = T_{j_l} \cdot T_{j_r}$ мы можем вычислить значения $max(Ext(P_i, T_j, OV(P_{i_l}, T_{j_r})))$ и $max(Ext(T_j, P_i, OV(T_{l_j}, P_{r_i})))$ за $O(n^2 \log n)$.

Доказательство: Мы приведем алгоритм, который вычисляет значение $max(Ext(P_i, T_j, OV(P_{i_l}, T_{j_r})))$, а второе значение можно вычислить аналогично. Зафиксируем произвольную прогрессию $\langle a, d, t\rangle \in OL(P_{i_l}, T_{r_j})$. Будем предполагать, что $t > 1$ и $a < d$, поскольку случаи $t = 1$ или $a = d$ тривиальны.

Положим $u = T_{j_r}[1 \dots a]$ и $v = T_{j_r}[a+1 \dots d]$ и вычислим следующие значения:

$$e_1 = LCPref(P_{i_r}, (vu)^*) = \begin{cases} FM(T_{i_r}, P_{i_r}, a+1), \text{ если } FM(T_{j_r}, P_{i_r}, a+1) < d \\ FM(P_{i_r}, P_{i_r}, d+1) + d, \text{ иначе} \end{cases}$$

$$e_2 = LCSuf(P_{i_l}, (uv)^*) = FM(P_{i_l}^R, P_{i_l}^R, d+1) + d,$$

$$e_3 = LCPref(T_{j_r}, (vu)^*) = FM(T_{j_r}, T_{j_r}, d+1) + d,$$

$$e_4 = LCSuf(T_{j_l}, (uv)^*) = \begin{cases} FM(P_{i_l}^R, T_{j_l}^R, a+1), \text{ если } FM(P_{j_l}^R, T_{j_l}^R, a+1) < d \\ FM(T_{j_l}^R, T_{j_l}^R, d+1) + d, \text{ иначе} \end{cases}$$

Мы можем вычислить эти значения по крайней мере за шесть вызовов функции FM. Заметим, что $P_i[|P_{i_l}| - e_2 + 1 \dots |P_{i_l}| + e_1]$ наибольшая подстрока P_i, которая содержит $P_i[|P_{i_l}| - d + 1 \dots |P_{i_l}|]$ и имеет период d, а $T_j[|T_{j_l}| - e_4 + 1 \dots |T_{j_l}| + e_3]$ наибольшая подстрока T_j, которая содержит $T_j[|T_{j_l}| + 1 \dots |T_{j_l}| + d]$ и имеет период d.

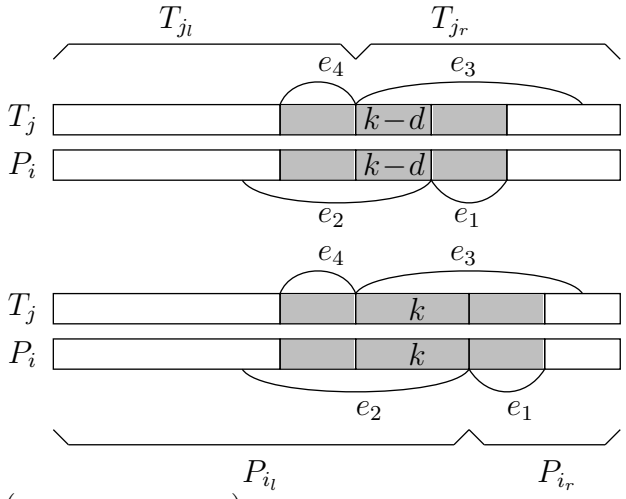
Пусть $k \in \langle a, d, t\rangle$, покажем, как меняется значение $Ext(P_i, T_j, k)$ в зависимости от k:

1. $\mathbf{k < min(e_3 - e_1, e_2 - e_4)}$.

   Если $k - d \in \langle a, d, t\rangle$, то нетрудно видеть, что $Ext(P_i, T_j, k) = Ext(P_i, T_j, k-d) + d$, поэтому мы можем вычислить значение:

$$A = max\{Ext(P_i, T_j, k) : k < min(e_3 - e_1, e_2 - e_4)\} = Ext(P_i, T_j, k'),$$

$$\text{где } k' = max\{k \ : k < min(e_3 - e_1, e_2 - e_4)\}$$

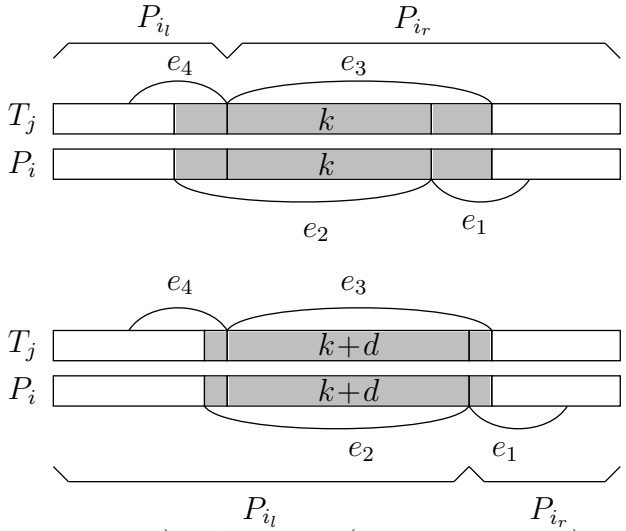


2. $\mathbf{k > max(e_3 - e_1, e_2 - e_4)}$.

   Если $k + d \in \langle a, d, t \rangle$, то нетрудно видеть, что $Ext(P_i, T_j, k) = Ext(P_i, T_j, k + d) + d$. Поэтому мы можем вычислить значение:

   $$B = max\{Ext(P_i, T_j, k) : k > max(e_3 - e_1, e_2 - e_4)\} = Ext(P_i, T_j, k''),$$

   где $k'' = min\{k : k > max(e_3 - e_1, e_2 - e_4)\}$



3. $\mathbf{min(e_3 - e_1, e_2 - e_4) < k < max(e_3 - e_1, e_2 - e_4)}$.

   В этом случае мы получаем, что $Ext(P_i, T_j, k) = min(e_1 + e_2, e_3 + e_4)$ для любого k из области значений этого случая. Поэтому мы можем вычислить значение:

   $$C = max\{Ext(P_i, T_j, k) : min(e_3 - e_1, e_2 - e_4) < k < max(e_3 - e_1, e_2 - e_4)\} =$$

$$= min(e_1 + e_2, e_3 + e_4)$$

4. $\mathbf{k = e_3 - e_1}$.

   Мы можем вычислить значение:

$$E = Ext(P_i, T_j, k) = k + min(e_2 - k, e_4) + LCPref(T_{j_r}[k+1 \ldots |T_{j_r}|], P_{i_r}) =$$
$$= k + min(e_2 - k, e_4) + FM(T_{j_r}, P_{i_r}, k+1)$$

5. $\mathbf{k = e_2 - e_4}$.

   Мы можем вычислить значение:

$$E = Ext(P_i, T_j, k) = k + LCSuf(P_{i_l}[1 \ldots |P_{i_l}| - k], T_{j_l}) + min(e_1, e_3 - k) =$$
$$= k + FM(P^R_{i_l}, T^R_{j_l}, k+1) + min(e_1, e_3 - k)$$

6. $\mathbf{k = e_3 - e_1 = e_2 - e_4}$.

   Мы можем вычислить значение:

$$F = Ext(P_i, T_j, k) =$$
$$= k + LCSuf(P_{i_l}[1 \ldots |P_{i_l}| - k], T_{j_l}) + LCPref(T_{j_r}[k+1 \ldots |T_{j_r}|], P_{i_r}) =$$
$$= k + FM(P^R_{i_l}, T^R_{j_l}, k+1) + FM(T_{j_r}, P_{i_r}, k+1).$$

Ясно, что верны следующие неравенства: $F \geq max\{D, E\} \geq C \geq max\{A, B\}$. Основываясь на этом, мы можем вычислить $Ext(P_i, T_j, \langle a, d, t \rangle)$ с помощью не более двух вызовов FM, при условии что значения e_1, e_2, e_3, e_4 уже вычислены. □

Поскольку каждая клетка таблицы перекрытий содержит $O(n)$ прогрессий и каждый вызов FM требует $O(n \log n)$ времени, то нам необходимо $O(n^2 \log n)$ времени на вычисление $max(Ext(P_i, T_j, OV(P_{i_l}, T_{j_r})))$.

5.3.2 Оценка сложности

Во время работы алгоритма мы будем хранить три глобальных параметра: правило, в котором содержится текущая наибольшая общая подстрока, ее позиция начала и длина этой подстроки. Для каждой пары правил P_i, T_j мы вычисляем три значения: $max(Ext(P_i, T_j, OV(P_{i_l}, T_{j_r})))$, $max(Ext(T_j, P_i, OV(T_{l_j}, P_{r_i})))$ и $LCStr(P_i, T_j)$ за $O(n^2 \log n)$. Если на некотором шаге мы получили длину большую, чем глобальная, то мы обновляем все три значения. В итоге нам потребуется $O(n^4 \log n)$ времени.

Поскольку во время работы алгоритма мы строим таблицу перекрытий, то нам потребуется $O(n^3)$ пространства.

Замечание. Представленный алгоритм взят из работы В. Матсубары, Ш. Иненаги, А. Ишино, А. Шинохары, Т. Накамуры и К. Хашимото [27].

5.4 Задача Поиск всех палиндромов

Определение. Строка T называется *палиндромом*, если $T = T^R$. Если мы зафиксируем некоторую позицию в T, то можно определить *максимальный по длине палиндром* в этой позиции. Определим множество $Pals(T) = \{(p, q) : T[p \dots q]$ – максимальный палиндром с центром в $\lfloor \frac{p+q}{2} \rfloor\}$.

Задача: **Поиск всех падиндромов**

Вход: ПП $\mathcal{T}$, которая выводит текст T

Выход: Множество $Pals(T)$, представленное в сжатом виде

5.4.1 Ключевые идеи

Ясно, что множество $Pals(T)$ может содержать экспоненциально много элементов, например, мы можем взять строку $T =$ «$aa \dots aa$» такую, что $|T| = 2^n$. Поэтому нам необходимо эффективно хранить это множество. Для любого правила $T_i \in \mathcal{T}$ определим $Pals^*(T_i)$ – множество максимальных палиндромов T_i, которые касаются позиции разреза T_i. Формально $Pals^*(T_i) = \{(p, q) \in Pals(T_i) : 1 \le p \le |T_l| + 1, |T_l| \le q \le |T_i|, p \le q\}$. Определим множества $PPals(T) = \{(1, q) \in Pals(T) : 1 \le q \le |T|\}$ – множество префиксных палиндромов и $SPals(T_i) = \{(p, |T|) \in Pals(T) : 1 \le p \le |T|\}$ – множество суффиксных палиндромов. Возникает **идея накопления палидромов** с увеличением длины правил: для любого правила $T_i = T_l \cdot T_r$

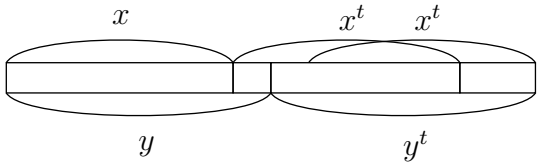
$$Pals(T_i) = (Pals(T_l) \backslash SPals(T_l)) \cup Pals^*(T_i) \cup ((Pals(T_r) \backslash PPals(T_r)) \oplus |T_l|).$$

Поэтому нам достаточно научиться вычислять множества $PPals(T_i)$, $SPals(T_i), Pals^*(T_i)$ для каждого правила.

Лемма 13 (о краевых палиндромах)**.**
Для любой строки T множества $PPals$ и $SPals$ линейно-сжимаемы относительно $|T|$.

Доказательство: Доказательство проведем для множества $PPals$, случай с $SPals$ аналогичен. Индукция по параметру $j = \log_2 |T|$.

База. Если $j = 0$, для однобуквенного текста палиндромов не существует, поэтому утверждение верно.



Шаг. Пусть для всех значений меньших j утверждение верно. Покажем его для j. Ясно, что центры всех палиндромов строки T лежат не правее позиции $\frac{\lceil |T| \rceil}{2}$. Для палиндромов с центрами в позициях $0, \ldots, \frac{\lceil |T| \rceil}{4}$ справедливо предположение индукции, так как $\log_2(\frac{\lceil |T| \rceil}{2}) = j - 1$. Нам остается понять как расположены центры палиндромов в $(\frac{\lceil |T| \rceil}{4}, \frac{\lceil |T| \rceil}{2}]$. Пусть $q_1, q_2 \in (\frac{\lceil |T| \rceil}{4}, \frac{\lceil |T| \rceil}{2}]$ – произвольные центры палиндромов xx^t и yy^t соответственно. Так как x – префикс y, то x^t – суффикс y^t. Ясно, что $q_2/q_1 < 2$ (из-за взаимного расположения), тогда строка x^t, начинающаяся в q_1 пересекается со строкой x^t, являющейся префиксом y^t. Значит, строка x^t является p-периодической для некоторого значения p. Поэтому строки x, y тоже p-периодические. Следовательно, центры палиндромов, расположенные в $(\frac{\lceil |T| \rceil}{4}, \frac{\lceil |T| \rceil}{2}]$, образуют арифметическую прогрессию с периодом, равным наибольшему общему делителю всех значений p для всех возможных пар q_1, q_2. $\square$

Лемма 14 (о усеченных палиндромах)**.**
Для любого правила $T_i = T_l \cdot T_r$ и любой пары $(p, q) \in Pals^*(T_i)$ существует целое $h \geq 0$ такое, что $(p + h, q - h) \in SPals(T_l) \cup (PPals(T_r) \oplus |T_l|) \cup \{(|T_l|, |T_l| + 1)\}$.

Доказательство: Так как $T_i[p \ldots q]$ палиндром, то $T_i[p + h \ldots p - h]$ также является палиндромом для любого $0 \leq h \leq \lfloor \frac{t-s}{2} \rfloor$. Рассмотрим следующие случаи:

1. Если $\lfloor \frac{p+q}{2} \rfloor < |T_l|$, то для $h = p - |T_l|$ мы получаем, что $(p + l, p - l) \in SPals(T_l)$.

2. Если $\lfloor \frac{p+q}{2} \rfloor > |T_l|$, то для $h = |T_l| - p + 1$ мы получаем, что $(p + h, p - h) \in PPals(T_r)$.

3. Пусть $\lfloor \frac{p+q}{2} \rfloor = |T_l|$. Если $q-p+1$ – нечетно, тогда можно применить случай 1, так как $T_l[|T_l|] = T_l[|T_l|]^R$ и $(|T_l|, |T_l|) \in SPals(T_l)$. Если $q-p+1$ четно, положим $h = |T_l| - p$. В этом случае, мы получаем $p+q = 2|T_l|+1$, поэтому $p+h = |T_l|$ и $q-h = |T_l|+1$.

По лемме о усеченных палиндромах множество $Pals^*(T_i)$ может быть вычислено с помощью "расширения" всех палиндромов в $SPals(T_l)$ и $PPals(T_r)$ до максимальных внутри T_i и поиска максимального четного палиндрома, центрированного в позиции $|T_l|$ строки T_i. □

Рекурсивное вычисление $Pals^*(T_i), PPals(T_i), SPals(T_i)$**:**
Для любого правила $T_i = T_l \cdot T_r$ верно:

- $Pals^*(T_i) = Ext(T_i, SPals(T_l)) \cup Ext(T_i, PPals(T_r)) \cup Pals^*(T_i)$, где
  $Ext(T_i, SPals(T_l)) = \{Ext(T_i, (p, |T_l|)) : (p, |T_l|) \in SPals(T_l)\}$,
  $Ext(T_i, PPals(T_r)) = \{Ext(T_i.(|T_l|+1, |T_l|+q)) : (1, q) \in PPals(T_r)\}$,
  $CPals(T_i) = \{(|T_l| - l + 1, |T_l| + l) \in Pals(T_i) : l \geq 1\}$.
- $Pals(T_i) = PPals(T_l) \cap \{(1, q) \in Pals^*(T_i)\}$ и
  $SPals(T_i) = (SPals(T_r) \oplus |T_l|) \cap \{(p, |T_i|) \in Pals^*(T_i)\}$.

Эффективное вычисление $Pals^*(T_i)$**:**
Для произвольных правил $T_i = T_l \cdot T_r, T_j$ определим *множество вхождений* T_j в T_i, касающихся позиций разреза T_i следующим образом:

$$Occ(T_i, T_j) = \{s > 0 : T_i[s \ldots s + |T_j| - 1] = T_j, |T_l| - |T_j| + 1 \leq s \leq |T_l|\}$$

Во время работы алгоритма мы будем заполнять AP-таблицу, т.е. на каждом шаге алгоритма мы тратим $O(n)$ операций на заполнение AP-таблицы. А каждая ячейка AP-таблицы как раз хранит множество $Occ(T_i, T_j)$.

Лемма 15 (о вычислении FM)**.**
Для любых правил T_i, T_j и целого k, $FM(T_i, T_j, k)$ может быть вычислено за время $O(n^2)$ при условии, что $Occ(T_i, T_j)$ уже вычислено для всех предшественников.

Доказательство: Пусть $T_j = T_l \cdot T_r$, тогда нетрудно заметить, что

$$FM(T_i, T_j, k) = \begin{cases} |Y_l| + FM(T_i, T_r, k), \text{ если } k \in Occ(T_i, T_l) \\ FM(T_i, T_l, k), \text{ иначе } \square \end{cases}$$

Это лемма дает рекурсивный алгоритм вычисления $FM(T_i, T_j, k)$.

Для произвольного правила $T_i = T_l \cdot T_r$ и любой арифметической прогрессии $(1, \langle a, d, t\rangle) \subseteq PPals(T_r)$ и $(\langle a', d', t'\rangle, |T_l|) \subseteq SPals(T_l)$ положим $Ext(T_i, (1, \langle a, d, t\rangle)) = \{Ext(T_i, (|T_l| + 1, |T_l| + q)) : q \in \langle a, d, t\rangle\}$ и $Ext(T_i, (\langle a', d', t'\rangle, |T_l|)) = \{Ext(T_i, (p, |T_l|)) : p \in \langle a', d', t'\rangle\}$.

Лемма 16 (о расширении префикса/суффикса)**.**
$Ext(T_i, (1, \langle a, d, t\rangle))$ и $Ext(T_i, (\langle a', d', t'\rangle, |T_l|))$ могут быть представлены не более двумя арифметическими прогрессиями и парой начала/конца максимального палиндрома. Эти значения могут быть вычислены не более чем за 4 вызова FM.

Доказательство: По лемме 3.4 из [3].

Лемма 17 (ключевая)**.**
Для любого правила $T_i = T_l \cdot T_r$ множество $Pals^*(T_i)$ может быть вычислено за время $O(n^3)$ и требует $O(n)$ пространства.

Доказательство: По определению множество $CPals(T_i)$ или пусто или содержит единственный элемент. Если оно не пусто, то содержит максимальный четный палиндром центрированный в позиции $|T_l|$. Пусть $l = FM(T_r, T_l^R, 1)$, тогда мы получаем

$$CPals(T_i) = \begin{cases} \varnothing, \text{ если } l = 0 \\ (|T_l - l + 1, |T_l| + l|), \text{ иначе.} \end{cases}$$

Поэтому мы можем вычислить $CPals(T_i)$ за $O(n^2)$.

Теперь рассмотрим $Ext(T_i, SPals(T_l))$. По лемме о вычислении FM (см. стр. 34) и лемме о расширении префикса/суффикса (см. стр. 35) каждое подмножество $Ext(T_i, \langle a, d, t\rangle) \subseteq Ext(T_i, SPals(T_l))$ требует $O(1)$ пространства и может быть вычислено за $O(n^2)$. Из леммы о краевых палиндромах (см. стр. 32) следует, что $Ext(T_i, SPals(T_l))$ содержит не более $O(n)$ арифметических прогрессий, поэтому мы можем вычислить $Ext(T_i, SPals(T_l))$ за $O(n^3)$. □

5.4.2 Оценка сложности

Для каждого правила $T_i \in \mathcal{T}$ мы умеет строить множество $Pals(T_i)$ за время $O(n^3)$, поэтому общее время работы алгоритма равно $O(n^4)$. Поскольку за время работы алгоритма будет построена AP-таблица, то нам требуется $O(n^2)$ пространства.

Замечание. Представленный алгоритм взят из работы В. Матсубары, Ш. Иненаги, А. Ишино, А. Шинохары, Т. Накамуры и К. Хашимото [27].

5.5 Задача Поиск квадратов в строке

Задачи **Свобода строки от квадратов** и **Поиск всех квадратов в строке** являются очень близкими. Поэтому для удобства понимания мы сначала представил алгоритм для задачи **Свобода строки от квадратов** (с его помощью мы поймем почему мы не пропустим ни один квадрат в строке), а потом – **Поиск всех квадратов в строке** (нам останется только решить задачу хранения экспоненциально большого количества квадратов).

5.5.1 Свобода строки от квадратов

В этом параграфе под *квадратом* будем понимать строку вида xx, где $x \in \Sigma^*$.

Задача: **Свобода строки от квадратов**
Вход: ПП $\mathcal{T}$, которая представляет текст T.
Вопрос: Свободна ли строка от квадратов?
Идеи алгоритма:

- **Локализация** Мы будем искать только те квадраты, которые полностью содержатся внутри некоторого $T_i \in \mathcal{T}$ и касаются позиции разреза этого правила. Почему при этом мы не пропустим ни одного вхождения квадрата? Зафиксируем произвольный квадрат xx в строке T и, начиная с корня дерева $Tree(\mathcal{T})$, осуществим следующую операцию: Пусть нам встретилось правило $T_i = T_l \cdot T_r$. Если xx касается позиции разреза T_i, то мы останавливаемся и выдаем i. Иначе xx полностью содержится в T_l/T_r, тогда мы переходим к правилу T_l/T_r. Поскольку высота $\mathcal{T}$ порядка $O(n)$, то ясно, что алгоритм остановится на некотором шаге и вернет нам номер искомого правила.

  Поэтому мы можем зафиксировать некоторое правило $T_i \in \mathcal{T}$ и нам достаточно научиться проверять его на свободу от квадратов.

- **Представление строки** Для простоты изложения мы будем считать, что $|T_i| = 2^k$ для некоторого фиксированного k. Как это понимать? Ясно, что существует такое значение $k-1$, что $max(|T_l|, |T_r|) \leq 2^{k-1}$ и при этом $k = O(n)$ (если предположить, что $|T_i| = a^n$, тогда $|T_i| = 2^{n \log_2 a}$, т.е. $k = n \log_2 a$). Неформально можно считать, что мы дополнили строку T_i слева и справа специальными символами до длины 2^k так, что $|T_l| = 2^{k-1} = |T_r|$.

- **Итерация по длинам квадратов** Поскольку $|T_i| = 2^k$, то в этой строке могут существовать квадраты с $|x| \in \{1 \dots 2^{k-1}\}$, при этом не забываем, что они касаются позиции разреза T_i. Разобьем отрезок длин на $O(k)$ частей и будем последовательно искать квадраты с длиной $|x|$, соответствующей некоторому отрезку разбиения, т.е. квадраты с $|x| = 1$ мы ищем по определению и далее ищем квадраты с $|x| = \{2^{j-1} + 1 \dots 2^j\}$, где $j \in \{1 \dots k-1\}$. Ясно, что при таком поиске ни одного квадрата не будет пропущено. Поэтому мы можем зафиксировать отрезок $\{2^{j-1} + 1 \dots 2^j\}$.

- **Идея локальной области поиска** Разобьем 2^j-окрестность позиции разреза T_i на 8 одинаковых блоков длины 2^{j-2}. Зафиксируем один из таких блоков – B. В зависимости от расположения блока B мы можем зафиксировать 4-е последовательных блока слева или справа от B. Найдем все вхождения B в области из 4-х блоков. Из рисунка видно, что вхождения блока B, начинающиеся в блоках 1 и 4, не будут удовлетворять заявленным длинам квадрата. Поэтому нас интересуют только вхождения B, которые начинаются во 2 и 3 блоках. Блоки 2 и 3 будем называть локальной областью поиска.

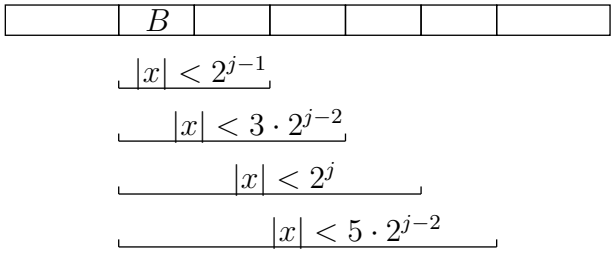


Рисунок 5. Выделение локальной области поиска.

Лемма 18 (о плотности вхождений)**.**
Если локальная область поиска блока B содержит более одного вхождения блока, тогда она содержит квадрат.

Доказательство: Если локальная область поиска содержит более одного вхождения B, то либо мы имеем квадрат BB (в случае двух вхождений), либо хотя бы пара из них пересекается. Если два блока пересекаются, тогда строка B является периодической и, следовательно, содержит хотя бы один квадрат. □

Благодаря этой Лемме нам необходимо обрабатывать лишь конечное число вхождений блока B в локальной области поиска.

Техническая реализация

Пусть нам задана ПП $\mathcal{T}$ такая, что $|\mathcal{T}| = O(n)$ и $|T| = 2^n$. Предполагается, что все правила $X_i \in \mathcal{T}$ имеют вид $X_i = X_l \cdot X_r$, где $|X_i| = 2^k, |X_l| = |X_r| = 2^{k-1}$. Проследим первые три шага алгоритма и затем опишем произвольный шаг алгоритма.

- На первом шаге алгоритма мы ищем квадраты xx такие, что $|x| \in \{2^{k-2}+1, \ldots, 2^{k-1}\}$, которые касаются позиции разреза, то есть позиции 2^{k-1}. Разбиваем всю строку на 8 блоков длины 2^{k-3}.

  1. Фиксируем блок $B = T_i[0 \ldots 2^{n-3}]$ и локальную область поиска $LA = T_i[2^{k-2} \ldots 2^{k-1}]$. Строим грамматики, которые соответствуют строкам B и LA. Запускаем алгоритм поиска образца в тексте на $\mathcal{B}$ и $\mathcal{LA}$. Если найдено более одного вхождения B, тогда останавливаем алгоритм и отвечаем "нет". Иначе надо понять, образует ли пара блоков квадрат.

     Зафиксируем произвольный найденный блок $B' = [j \ldots j + 2^{k-2}]$. С помощью алгоритма поиска образца в тексте мы хотим найти наибольшее значение l такое, что $T[2^{n-3}+1 \ldots 2^{n-3}+l] = T[j + 2^{k-2} + 1 \ldots j + 2^{k-2} + l]$. Формально, вначале полагаем $l = 1$ и распаковываем пару символов. Если они равны, то удваиваем значение l и переходим на следующий шаг. Иначе останавливаемся и полагаем $l = 0$. На произвольном шаге: проверяем равенство сжатых строки $T_i[2^{n-3} + 1 \ldots 2^{n-3} + l]$ и $T_i[j + 2^{k-2} + 1 \ldots j + 2^{k-2} + l]$ с помощью алгоритма поиска образца в тексте. Если равны, тогда увеличиваем значение l и

переходим на следующий шаг. Иначе уменьшаем l и переходим на следующий шаг. Если в итоге $l \geq i - 2^{n-3}$, тогда пара блоков образует квадрат.

2. Фиксируем блок $B = T_i[2^{k-3} \dots 2^{k-2}]$ и локальную область поиска $LA = [3 \cdot 2^{k-3} \dots 5 \cdot 2^{k-3}]$. Запускаем алгоритм поиска образца в тексте на $\mathcal{B}$ и $\mathcal{LA}$). Если найдено более одного вхождения, тогда выдаем ответ "нет". Иначе, для пары блоков B и $B' = T_i[j \dots j + 2^{n-3}]$ вычисляем длину максимального суффикса и префикса. Пусть мы добавили суффикс длины s и префикс длины p. Если $s + p \geq j - 2^{n-2}$, тогда в строке существует квадрат.
3. Блоки $T_i[2^{n-2} \dots 3 \cdot 2^{n-3}], T_i[3 \cdot 2^{n-3} \dots 2^{n-1}]$ обрабатываются аналогично случаю 2.
4. Блок $T_i[2^{n-1} \dots 5 \cdot 2^{n-3}]$ обрабатываются аналогично случаю 2. Меняется только то, что локальная область поиска находится слева от блока, а не справа.

- На втором шаге мы ищем квадраты xx такие, что $|x| \in \{2^{n-3} + 1 \dots 2^{n-2}\}$ и касаются позиции 2^{n-1}. На данном шаге нам не удастся сузить область поиска, поскольку могут существовать квадраты с началами в блоке $T_i[0 \dots 2^{n-3}]$ и касающиеся позиции 2^{n-1}. Поэтому разбиваем строку на 16 блоков длины 2^{n-4} и аналогично первому шагу ищем квадраты.

- На третьем шаге ищем квадраты xx такие, что $|x| \in \{2^{n-4}+1 \dots 2^{n-3}\}$ и касаются позиции 2^{n-1}. Область поиска квадратов можно сузить до подстроки $T_i[2^{n-2} \dots 3 \cdot 2^{n-2}]$, так как если квадрат начинается левее или правее этой подстроки, то он не сможет касаться позиции разреза, а следовательно, полностью лежит либо в T_l, либо в T_r. Таким образом, мы рассматриваем только половину от исходной строки. Дальнейшие шаги аналогичны случаю 2.

- На четвертом шаге ищем квадраты xx такие, что $|x| \in \{2^{n-5} + 1 \dots 2^{n-4}\}$ и касающиеся позиции 2^{n-1}. Область поиска сужаем до подстроки $T_i[3 \cdot 2^{n-3} \dots 5 \cdot 2^{n-3}]$, то есть область уменьшилась ровно в половину от предыдущего шага. Разбиваем подстроку на 8 блоков и проделываем действия аналогичные первому шагу.

Оценка сложности: На каждом шаге алгоритма мы разбиваем строку на 8 (16) блоков одинаковой длины, запускаем алгоритм поиска подстроки для каждого блока, который работает за $O(n^3)$. Получаем не более пары блоков, которые могут образовывать квадрат. Эту пару мы расширяем в обоих направлениях с помощью алгоритма поиска образца в тексте за время $O(n^4)$. В итоге сложность каждого шага в худшем случае равна $O(n^4)$. Всего таких шагов по длине $|x|$ будет $n+1$, следовательно сложность обработки каждого правила в худшем случае будет порядка $O(n^5)$. Итоговая сложность обработки ПП $\mathcal{T}$ в худшем случае равна $O(n^6)$.

5.5.2 Поиск всех квадратов в строке

Определение. Под *квадратом* будем понимать строку xx, где $x \in \Sigma^*$. Если нет вхождений строки x внутри xx, отличных от вхождений начинающихся в позициях 0 и $|x|$, тогда такую строку xx будем называть квадратом. Иначе такую строку xx будем называть *повтором*.

Задача: **Поиск всех квадратов в строке**

Вход: ПП $\mathcal{T}$, которая представляет текст T.

Выход: Таблица (полиномиального размера от входа), которая содержит все семейства квадратов, представленные в сжатом виде.

Логика поиска всех квадратов.

Мы сохраним все идеи из задачи **Свобода строки от квадратов**. То есть мы зафиксируем правило $T_i \in \mathcal{T}$ и его позицию разреза γ, зафиксируем промежуток длин $|x| \in \{2^{k-1}+1, 2^k\}$ и, наконец, зафиксируем блок B и его локальную область поиска длины 2^{k-1}. Разобьем локальную область поиска на 4-е равных блока ($\frac{|B|}{2}$-блока) длины 2^{k-3}. Найдем все начала вхождений блока B в каждый $\frac{|B|}{2}$-блок.

Лемма 19 (о локальном представлении)**.**
Пусть p – период строки B, $p_1 < p_2 < \cdots < p_k$ такие, что $p_k - p_1 \leq \frac{|B|}{2}$ – позиции начал вхождений B в строке T, тогда последовательность $\{p_i\}$ образует арифметическую прогрессию.

Доказательство: Можно считать, что $k \geq 2$. Возьмем $q = p_{i+1} - p_i$ для некоторого $i \in \{1 \dots k-1\}$ и покажем, что $p = q$. Ясно, что строка B имеет периоды p и q. Так как $p_k - p_1 \leq \frac{|B|}{2}$, то $p \leq q \leq \lceil \frac{|B|}{2} \rceil$ и по теореме Файна-Вилфа (см. стр. 22) мы получаем, что B имеет период

НОД(p, q). Но p по определению длина наименьшего периода B, поэтому $p =$НОД(p, q) и поэтому p должно делить q. Если $q > p$, тогда мы пропустили вхождение строки B в T в позиции $p_i + p$, а этого не может быть, так как мы нашли все вхождения B в T. Поэтому получаем, что $p = q$. $\square$

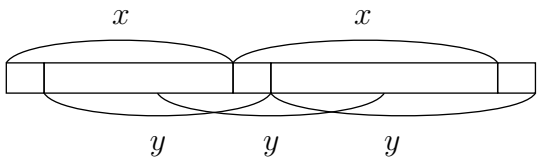
По лемме о локальном представлении мы получаем, что вхождения B внутри $\frac{|B|}{2}$-блока образуют арифметическую прогрессию. Пусть $\langle a, p, t\rangle$ – арифметическая прогрессия вхождений B внутри некоторого $\frac{|B|}{2}$-блока. Поэтому нам остается научится определять какие элементы арифметической прогрессии образуют квадраты.

Простой случай, $t = 1$

Аналогично задаче **Свобода строки от квадратов** мы можем расширить пару блоков. В зависимости от величины расширения мы либо не найдем повторов вообще, либо найдем один или несколько повторов, каждый из которых может быть квадратом. Будем говорить, что тройка $(l, r, |x|)$ является семейством повторов, где $l(r)$ – позиция самого левого (правого) центра квадрата в семействе, а $|x|$ – половина длины квадрата. Нам надо научиться эффективно извлекать квадраты из семейства повторов.

Лемма 20 (о проверке)**.**
Семейство повторов содержит квадрат тогда и только тогда, когда все элементы семейства являются квадратами.

Доказательство: Рассмотрим строку T, которая представляет семейство повторов. Без ограничения общности можно считать, что найден квадрат в позиции 0 строки T. Пусть в позиции $0 < h < |x|$ мы нашли повтор yy, который не является квадратом, т.е. существует вхождение строки y между позициями h и $h + |x|$. Тогда строка y является периодической.



Пусть p – период y. Рассмотрим два случая:

- Если $p \geq h$, тогда $T[0, h]$ является суффиксом блока p, т.е. блок p можно представить в виде $zT[0, h]$, для некоторой строки z. Тогда строка x представима в виде: $T[0, h]zT[0, h]z \dots zT[0, h]$. Следовательно, x периодическая, противоречие.

- Если $p < h$, тогда рассмотрим суффикс y. В этом случае строку $T[0, h]$ можно представить в виде zyz, где $p = yz$, тогда x имеет вид $zyz \dots yz$. Следовательно x периодическая, противоречие. □

Из леммы следует, что мы можем извлечь за полиномиальное время квадраты из любого семейства повторов.

Сложный случай, $t > 1$

Определение. Пусть у нас зафиксирован некоторый блок B, который начинается в позиции b_0 текста T, и арифметическая прогрессия $\langle a, p, t\rangle = \{p_i\}(i \in \{1 \dots t\})$ его вхождений внутри некоторого $\frac{|B|}{2}$-блока. Определим α_L и α_R как *позиции обрыва p-периодичности* строки B слева и справа от нее соответственно. Если индексы удовлетворяют следующим неравенствам: $b_0-(p_t-b_0)+|B| \le \alpha_L, \alpha_R < p_1+|B|$, то будем называть их определенными. Иначе считаем, что они не определены. Ясно, что $p_t - b_0$ значение наибольшего $|x|$ для фиксированной пары $B, \langle a, p, t\rangle$, поэтому начало любого квадрата не может быть расположено правее позиции $b_0 - (p_t - b_0) + |B|$ и нам не важно где оборвется p-периодичность левее этой позиции.

Аналогично, определим γ_L и γ_R как *позиции обрыва p-периодичности* слева от позиции p_1 и справа от позиции $p_t + |B|$ соответственно. Будем говорить, что позиции γ_L и γ_R определены, если $b_0 \le \gamma_L$ и $\gamma_R < 2p_t - b_0$, иначе – не определены.

Если все индексы определены, тогда строки $T[\alpha_L + 1 \dots \alpha_R - 1]$, $T[\gamma_L+1 \dots \gamma_R-1]$ имеют период p и выполнены следующие соотношения:

$$\begin{gathered} T[\alpha_L] \ne T[\alpha_L + p], \qquad T[\alpha_R] \ne T[\alpha_R - p], \\ b_0 - (p_t - b_0) + |B| \le \alpha_L < b_0, \qquad b_0 + |B| \le \alpha_R < p_1 + |B| \end{gathered}$$

$$\begin{gathered} T[\gamma_L] \ne T[\gamma_L + p], \qquad T[\gamma_R] \ne T[\gamma_R - p], \\ b_0 \le \gamma_R < p_1, \qquad p_t + |B| \le \gamma_R < 2p_t - b_0 \end{gathered}$$

Лемма 21 (о взаимосвязи)**.**
Если хотя бы один из индексов α_R или γ_L определен, тогда и другой тоже определен, при этом $\alpha_R - \gamma_L \le p$.

Доказательство: Из определения α_R и γ_L мы получаем, что $T[b_0 \dots \alpha_R - 1]$ и $T[\gamma_L+1 \dots p_t+|B|-1]$ имеют период p, $T[\alpha_R] \ne T[\alpha_R - p]$ и $T[\gamma_L] \ne T[\gamma_L + p]$.

Если $p < \alpha_R - \gamma_L$, то из периодичности строки $T[b_0 \dots \alpha_R - 1]$ следует, что $T[\gamma_L] = T[\gamma_L+p]$, получаем противоречие с определением γ_L. Поэтому $\alpha_R - \gamma_L \le p$ или по крайней мере одно из значений α_R, γ_L не определено. Пусть α_R не определено, тогда $T[b_0 \dots q_1+|B|-1]$ имеет период длины p и мы получаем, что γ_L не может быть определено. Аналогично мы можем провести доказательство в случае, когда не определено γ_L. $\square$

Лемма 22 (о взаимосвязи)**.**
Пусть α_R, γ_L не определены, тогда ни один повтор, закрепленный за блоком B, не является квадратом.

Доказательство: Из неопределенности α_R, γ_L мы получаем, что строка $T[b_0 \dots p_t + |B| - 1]$ имеет период длины p. Рассмотрим любое $p_i \in < a, p, t >$ и зафиксируем $|x| = q_i - b_0$. Так как $T[b_0 \dots b_0 + |B| - 1] = T[p_i \dots p_i + |B| - 1]$, получаем что $T[b_0 \dots b_0 + p_t - p_i + |B| - 1] = T[p_i \dots p_t + |B| - 1]$. Поэтому подстрока $T[b_0 \dots p_t + |B| - 1]$ имеет период длины $|x|$. Но $p \leq \frac{|B|}{2} < |x|$, поэтому по теореме Файна-Вилфа (см. стр. 22) p делит $|x|$.

Пусть xx повторение, которое закреплено за блоком B и начинается в позиции s, тогда

$$x = T[s \dots s + |x| - 1] = T[s + |x| \dots b_0 + |x| - 1] \cdot T[b_0 \dots s + l - 1],$$

поэтому x имеет период длины p,следовательно не является квадратом. □

Леммы о взаимосвязи показывают нам, что ключевое значение имеет взаимное расположение α_R и γ_L.

Лемма 23 (о семействах повторов)**.**
Пусть α_R, γ_L определены, тогда

1. *Повторы, которые закреплены за блоком B и центрированы в позициях h таких, что $h \leq \gamma_L$, существуют только, если значение α_L определено. Эти повторения образуют семейство, которое соответствует $|x| = p_i - b_0$, обеспечивая тем самым существование некоторого $p_i \in \langle a, p, t \rangle$ такого, что $\gamma_L - \alpha_L = p_i - b_0$.*

2. *Повторы, которые закреплены за блоком B и центрированы в позициях h таких, что $\alpha_R < h$, существуют только, если значение γ_R определено. Эти повторения образуют семейство, которое соответствует $|x| = p_j - b_0$, обеспечивая тем самым существование некоторого $p_j \in \langle a, p, t \rangle$ такого, что $\gamma_R - \alpha_R = p_j - b_0$.*

Если $\alpha_R < \gamma_L$, тогда повторения, центрированные в позициях $\alpha_R < h \leq \gamma_L$, существуют только если оба значения α_L и γ_R определены и $\gamma_R - \alpha_R = \gamma_L - \alpha_L$.

Доказательство:

Пусть $xx = T[h - |x| \dots h + |x| - 1]$ повторение с центром в позиции h и закрепленное за блоком B такое, что $|x| = p_i - b_0$ для некоторого $p_i \in < a, p, t >$. Предположим, что $b_0 + |B| \leq h \leq \gamma_L$. Рассмотрим два случая:

- Если α_L не определено или $\alpha_L < \gamma_L - |x|$, тогда из свойства периодичности в определениях α_L и γ_L получаем $T[\gamma_L - |x| + p] = T[\gamma_L + p]$ и $T[\gamma_L - |x|] = T[\gamma_L - |x| + p]$. Так как мы зафиксировали повторение xx, то $T[\gamma_L - |x|] = T[\gamma_L]$. Поэтому $T[\gamma_L] = T[\gamma_L + p]$, что приводит к противоречию с определением γ_L.

- Если $\alpha_L > \gamma_L - |x|$, тогда из свойства периодичности в определениях α_L и γ_L мы получаем $T[\alpha_L + p] = T[\alpha_L + |x| + p]$ и $T[\alpha_L + |x|] = T[\alpha_L + |x| + p]$. Так как мы зафиксировали повторение xx, то $T[\alpha_L] = T[\alpha_L + |x|]$. Поэтому $T[\alpha_L] = T[\alpha_L + p]$, что приводит к противоречию с определением α_L.

Поэтому повторение xx может существовать только если $\alpha_L = \gamma_L - |x|$. Другими словами, существует некоторое $p_i \in \langle a, p, t \rangle$ такое, что $\gamma_L - \alpha_L = p_i - b_0$. Поскольку α_L, γ_L и b_0 фиксированы, то q_i единственно (если вообще существует). □

Используя эту лемму мы можем получить не более двух семейств повторов и выделить из них квадраты с помощью леммы о проверке (см. стр. 42). Однако, мы не рассмотрели единственный случай:

Лемма 24 (о семействах квадратов)**.**
Пусть α_R, γ_L определены и $\gamma_L < \alpha_R$, тогда для каждого $|x| = p_i - b_0$ могут существовать повторения с центрами в позициях h таких, что $\gamma_L < h \leq \alpha_R$. Все такие семейства повторов являются квадратами, которые центрированы в позициях $max(\alpha_L + |x|, \gamma_L) < h \leq min(\alpha_R, \gamma_R - |x|)$. Такое семейство квадратов непусто тогда и только тогда, когда $|x| < min(\alpha_R - \alpha_L, \gamma_R - \gamma_L)$.

Доказательство:

Рассмотрим повторения $T[h - |x| \dots h - 1] = T[h \dots h + l - 1]$, которые соответствуют $|x| = p_i - b_0$ и их центры удовлетворяют условию $\gamma_L < h \leq \alpha_R$. Покажем, что такие повторы существуют тогда и только тогда, когда $\alpha_L + |x| < h$ и $h \leq \gamma_R - |x|$.

Пусть $h \leq \alpha_L + |x|$, тогда $T[\alpha_L] = T[\alpha_L + |x|]$. Так как $\gamma_L < h$, мы получаем, что $T[\alpha_L + |x|] = T[\alpha_L + |x| + p]$. Но тогда $T[\alpha_L] = T[\alpha_L + p]$ в противоречие в определением α_L. Аналогичная ситуация, если $\gamma_R - |x| < h$.

С другой стороны, если $max(\alpha_L + |x|, \gamma_L) < h \leq min(\alpha_R, \gamma_R - |x|)$, то строки $T[h - |x| \dots h - 1]$, $T[h \dots h + |x| - 1]$ имеют период длины p. Так как $T[b_0 \dots b_0 + |B| - 1] = T[p_i \dots p_i + |B| - 1]$, получаем что $T[h - |x| \dots h - 1] = T[h \dots h + |x| - 1]$. Остается показать, что эти повторения являются квадратами. Пусть $T[h - |x| \dots h - 1] = z^j$ для некоторого $j > 1$, тогда $T[h - |x| \dots h - 1]$ имеет периоды длины p и $|z|$, По по теореме Файна-Вилфа (см. стр. 22) p делит $|z|$. Но тогда $T[h - p \dots h - 1] = T[h \dots h + p - 1]$ и $\alpha_R - \gamma_L \geq 2p$, в противоречие с первой леммой о взаимосвязи (стр. 42). □

По этой лемме мы можем получить семейство семейств повторов, поэтому будем его хранить в виде: $(\alpha_L, \alpha_R, \gamma_L, \gamma_R, \langle a, p, t \rangle)$.

Алгоритм поиска квадратов

1. **Итерация по всем правилам $\mathcal{T}$.**

   Лемма 25 (о локализации)**.**
   Пусть зафиксированы позиции начала и конца некоторого квадрата в строке T, тогда существует правило $T_i \in \mathcal{T}$ такое, что квадрат полностью содержится в строке T_i и касается ее позиции разреза.

   Поэтому для каждого правила мы заинтересованы лишь в поиске всех квадратов, касающихся позиции разреза, поскольку в противном случае мы можем гарантировать что этот квадрат был найден ранее. Ясно, что для правил длины 1 и 2 мы можем указать все квадраты в явном виде. Поэтому зафиксируем произвольное правило $T_i = T_l \cdot T_r$ и предполагаем, что для всех предшественников мы вычислили квадраты.

2. **Итерация по длинам квадратов**. Все квадраты в строке, касающиеся позиции разреза, мы разбиваем по длинам. Пусть $T_i = 2^n$, тогда мы будем последовательно искать квадраты с $|x| \in \{2^{n-2} + 1 \dots 2^{n-1}\}, \{2^{n-3} + 1 \dots 2^{n-2}\}, \dots, \{1, 2\}$. Очевидно, что, в зависимости от длин искомых квадратов, нас будет интересовать лишь

некоторая часть строки T_i. Зафиксируем произвольный промежуток длин: $\{2^{k-1}+1 \dots 2^k\}$, тогда нам достаточно рассмотреть 2^k-окрестность позиции разреза.

3. **Разбиение области поиска**. Разобьем область поиска на 8 блоков одинаковой длины – 2^{k-2}. Каждый из этих блоков мы будем обрабатывать отдельно, поэтому зафиксируем некоторый блок B и еще 4 блока справа от него (если невозможно взять 4 блока справа, тогда мы можем взять их слева от блока). Аналогично задаче **Свобода строки от квадратов**, нас интересуют только два центральных блока. Каждый из них мы снова разбиваем попалам, т.е. до длины 2^{k-3}. Зафиксируем произвольную пару: B и некоторый $\frac{|B|}{2}$-блок, которому соответствует арифметическая прогрессия $\langle a, p, t\rangle$ начал вхождений B внутри этого блока.

4. Если прогрессия состоит из одного элемента, тогда аналогично задаче **Свобода строки от квадратов** мы расширяем пару блоков. Если расширение успешно, тогда мы получаем семейство повторов, т.е. фиксированную длину $|x|$ и множество центров повторов. С помощью Леммы(о проверке) мы можем определить является ли это семейство повторов квадратами. Если прогрессия содержит более одного элемента, тогда мы расширяем p-периодичность блоков B и $\frac{|B|}{2}$ в обоих направлениях. Тем самым мы получим значения $\alpha_L, \alpha_R, \gamma_L, \gamma_R$. Если оба α_R и γ_L определены, тогда мы получаем по лемме о семействе повторов (см. стр. 44) и по лемме о семействе квадратов (см. стр. 45) либо не более 2-х семейств повторов, либо семейство семейств квадратов, представленное в виде: $(\alpha_L, \alpha_R, \gamma_L, \gamma_R, \langle a, p, t\rangle)$.

В результате будет построена таблица квадратов (ST-таблица).

Обсуждение алгоритма.

Оценим сложность алгоритма: для каждого правила $T_i \in \mathcal{T}$ и промежутка длин $|x| \in \{2^{k-1}+1 \dots 2^k\}$ мы 5 раз запускаем алгоритм поиска образца в тексте и получаем 5 пар вида $B, \langle a, p, t\rangle$. Для каждой такой пары мы либо расширяем пару блоков (если $t = 1$), это осуществляется за $O(n^4)$ (см. задача **Свобода строки от квадратов**), либо расширяем периодичность строки B и $\frac{|B|}{2}$-блока (если $t > 1$), это можно осуществить аналогично расширению пары блоков. Поэтому нам потребуется $O(n^4)$

времени для заполнения одно клетки ST-таблицы. В итоге, на заполнение ST-таблицы нам необходимо $O(n^6)$ времени и $O(n^2)$ пространства.

Покажем насколько удобной является ST-таблица. Например, мы можем задавать ей следующие вопросы:

1. Проверить, существует ли квадрат в позиции i текста T. (Позицию i можно считать центром/началом/концом квадрата). Покажем в случае когда i является центром квадрата.

   Просматриваем все правила из $\mathcal{T}$, которые содержат позицию i. Их порядка $O(n)$. Для каждого такого правила мы просматриваем столбец длин $|x|$. Поэтому нам необходимо научиться понимать, содержит ли семейство квадрат с центром в позиции i. Если семейство задано в виде: $(l, r, |x|)$, тогда мы просто проверяем принадлежность i промежутку (l, r). Если семейство представлено в виде $(\alpha_L, \alpha_R, \gamma_L, \gamma_R, \langle a, p, t\rangle)$, тогда надо проверить принадлежит ли i промежутку $(max(\alpha_L + q_1 - b_0, \gamma_L), min(\alpha_R, \gamma_R - q_1 + b_0))$. Если принадлежит, тогда останавливаемся и квадрат существует. Иначе продолжаем поиск.

   В худшем случае за $O(n^2)$ мы ответим на этот вопрос.

2. Пусть ПП $\mathcal{P}$ размера m выводит некоторый квадрат. Входит ли этот квадрат в строку T?

   Мы можем определить длину квадрата $|x|$ за $O(m)$. Просматриваем только строку длин квадратов, которой соответствует значение $|x|$. Теперь надо определить, принадлежит ли наш квадрат семейству. Если семейство имеет вид $(l, r, |x|)$, тогда мы берем подграмматику для строки $T[l - |x|, r + |x|]$ и запускаем алгоритм поиска вхождений $\mathcal{P}$. Если семейство задано в виде $(\alpha_L, \alpha_R, \gamma_L, \gamma_R, \langle a, p, t\rangle)$, тогда надо понять содержит ли оно квадраты длины $|x|$. Для этого решаем уравнение $|x| = q_1 + p \cdot i - b_0$ относительно i. Если такое i существует, тогда ему соответствует семейство $(max(\alpha_L + q_1 - b_0, \gamma_L), min(\alpha_R, \gamma_R - q_1 + b_0), |x|)$. А такие семейства мы умеем обрабатывать.

   В худшем случае за $O(m^3 n)$ мы ответим на этот запрос. Более того, мы сможем сохранить информацию о местах всех вхождений $\mathcal{P}$ в строке T, а также о их количестве. Ясно, что алгоритм поиска

сжатого образца в тексте решает аналогичную задачу, но ему необходимо времени порядка $O(n^2m)$ и, как правило, число n гораздо больше числа m. В таких случаях удобней использовать поиск по ST-таблице.

3. Найти информацию о всех квадратах, начинающихся в данной позиции. Всего существует $O(n)$ правил, которые содержат данную позицию.

   Лемма 26 (Крошмор и Риттер [19])**.**
   Если существует три квадрата xx, yy, zz такие, что $|x| < |y| < |z|$ и начинаются в одной позиции строки, тогда $|x| + |y| \leq |z|$.

   Из Леммы можно сделать вывод, что для каждого промежутка длин $|x|$ существует не более двух квадратов, начинающихся в одной позиции. Поэтому всего существует порядка $O(n)$ квадратов, начинающихся в данной позиции и значит мы за $O(n^2)$ сможем собрать информацию о всех $O(n)$ квадратах в явном виде. Еще за $O(n^3)$ мы сможем построить ПП, которая выводит все квадраты, начинающиеся в данной позиции (для каждого квадрата берем подграмматику из $\mathcal{T}$ за $O(n^2)$, а потом осуществляем $O(n)$ конкатенаций).

4. Найти информацию о всех квадратах фиксированной длины. Ясно, что для этого нам достаточно пройтись по строке ST-таблице, которая соответствует заданной длине $|x|$, и определить соответствует ли семейство значению $|x|$. То есть за время $O(n)$ мы сможем извлечь информацию о всех квадратах фиксированной длины.

5. Нетрудно понять, что за $O(n^2)$ мы сможем определить сколько квадратов (с повторами) содержится в строке T.

Замечание. Большая часть лемм взята из работы [2], алгоритмы приведенные в этом параграфе были придуманы автором работы.

6 Полиномиально неразрешимые задачи

6.1 Задача вложимости сжатых строк

Определение. Будем говорить, что строка P *вкладывается* в строку T, кратко $P \hookrightarrow T$, если существуют позиции $1 \leq i_1 < i_2 < \cdots < i_m \leq n$, такие что $P[k] = T[i_k]$ для всех $1 \leq k \leq m$.

В качестве задачи поиска мы могли бы рассмотреть задачу, которая должна найти множество позиций $\{i_1, \ldots i_m\}$ по которым можно осуществить вложение (конечно мы должны хранить это множество позиций в сжатом виде). Но оказывается, что представление текста с помощью ПП не позволяет решить эту задачу за полиномиальное время. Поэтому мы поставим задачу в виде вопроса:

Задача: **Вложимость сжатого текста**

Вход: ПП $\mathcal{P}, \mathcal{T}$, которые выводят строки P и T соответственно

Вопрос: Верно ли, что $P \hookrightarrow T$?

Оценивать сложность этой задачи мы будем в несколько этапов. Сначала мы покажем, что задача принадлежит классу PSPACE (оценка сверху). Далее покажем, что **Вложимость сжатого текста** является NP-трудной путем сведения задачи **Сумма размеров** к данной. Целый параграф мы будем моделировать логические операции с помощью ПП. В итоге мы сможем показать, что задача **Вложимость сжатого текста** является Θ_2^p-трудной (оценка снизу).

Предложение 1 (о верхней границе)**.**
Задача **Вложимость сжатого текста** *принадлежит* PSPACE.

Доказательство: Можно переформулировать жадный алгоритм, который решает задачу вложимости на строках, в терминах ПП. Например, можно реализовать следующую логику: глобально мы храним указатель на строку P – i_p и указатель на T – i_t. Изначально $i_p = i_t = 0$. Распаковываем символ $P[i_p]$ и запускаем поиск первого вхождения этого символа в $\mathcal{T}$, начиная с позиции i_t. И так далее мы пробегаем по всем символам строки P. Если на каком-то шаге мы не нашли символ $P[i_p]$ или $i_t = |T|$, тогда говорим "нет". Если мы дошли до символа $P[|P|]$ и нашли его вхождение в T после i_t, то останавливаемся и говорим "да". □

6.1.1 NP-трудность

Напомним формулировку известной задачи **Сумма размеров**:

Вход: целые числа $w_1, \ldots, w_n, t$, представленные в двоичном виде.

Вопрос: существуют ли такой набор $x_1, \ldots, x_n \in \{0, 1\}$ такой, что

$$\sum_{i=1}^{n} x_i \cdot w_i = t?$$

Теорема 3 (о NP-трудности)**.**
Задача **Вложимость сжатого текста** *является* NP*-трудной.*

Доказательство: Доказательство будем проводить с помощью полиномиального сведения задачи **Сумма размеров** к задаче **Вложимость сжатого текста**. Пусть $t, \bar{w} = w_1, \ldots, w_n$ входные данные задачи **Сумма размеров**, будем предполагать, что $n > 1$. Мы собираемся построить ПП $\mathcal{G}, \mathcal{H}$ такие, что существует подмножество $\{w_1, \ldots, w_n\}$ с суммой, равной t, тогда и только тогда, когда $G \hookrightarrow H$.

Введем необходимые определения. Положим $s = w_1 + \cdots + w_n$ и $N = 2^n s$. Предполагаем, что $t < s$. Пусть $x \in \{0, \ldots, 2^n - 1\}$ – целое число, тогда через x_i $(1 \leq i \leq n)$ обозначим i-й бит двоичного представления x. Поэтому, $x = \sum_{i=1}^{n} x_i 2^{i-1}$. Определим $x \circ \bar{w} = \sum_{i=1}^{n} x_i w_i$, поэтому $x \circ \bar{w}$ – сумма подмножества $\{w_1, \ldots, w_n\}$, закодированного с помощью x. Таким образом, $(t, \bar{w})$ – успешный вход для задачи **Сумма размеров** тогда и только тогда, когда существует $x \in \{0, \ldots, 2^n - 1\}$ такое, что $x \circ \bar{w} = t$. Теперь мы можем определить строки G, H:

$$h_1 = \prod_{x=0}^{2^{n-1}} (10^s) = (10^s)^{2^n}, \qquad h_2 = 0^{2N}, \qquad h_3 = \prod_{x=0}^{2^n - 1} (0^{x \circ \bar{w}} 10^{s - x \circ \bar{w}}),$$

$$h_4 = 0^{t+1}, \qquad h_0 = h_1 h_2 h_3 h_4, \qquad h = h_0^{5N},$$

$$g_0 10^{3N+t} 10^{N+1}, \qquad g = g_0^{5N-1}$$

Покажем, что строки g, h могут быть порождены ПП полиномиального размера относительно входа $t, \bar{w}$. Заметим, что во всех строках, кроме h_3, мы используем константное число конкатенаций и экспонент с полиномиально большим числом битов. Эти конструкции могут быть легко реализованы с помощью ПП. Построим ПП для строки h_3. Заметим, что

в h_3 между любыми соседними единичками находится в точности n ноликов, т.е. строку h_3 можно представить в виде:

$$h_3 = \prod_{i=0}^{2^{n-1}} P \cdot 1,$$

где $P =$«$10\ldots0$» – строка длины $s+1$. Для строки P легко простить ПП $\mathcal{P}$ размера $\log s + 2$. Например, если n – четное, тогда грамматика имеет вид: $P_1 = 1, P_2 = 0, P_3 = P_2 \cdot P_2, \ldots, P_{\log s+1} = P_{\log s} \cdot P_{\log s}, P_{\log s+2} = P_1 \cdot P_{\log s+1}$. Ясно, что нам еще необходимо n правил для того, чтобы представить конкатенацию из $2^n - 1$ строк P. Поэтому итоговый размер грамматики для h_3 будет порядка $O(n \log s)$.

Теперь мы покажем, что $g \hookrightarrow h$ тогда и только тогда, когда существует $x \in \{0\ldots2^n-1\} : x \circ \bar{w} = t$. Предположим, что существует $x \in \{0\ldots2^n-1\}$ такое, что $x \circ \bar{w} = t$. Рассмотрим префикс $h_1h_2h_3h_4h_1$ строки h. Мы можем вложить $g_0 = 10^{3N+t}10^{N+1}$ в $h_1h_2h_3h_4h_1$:

1. Отобразим первую единичку из g_0 в единичку из блока с номером x строки h_0. При этом в строке h_0 остается еще $N-(x-1)s$ ноликов, тогда отобразим следующие $N-(x-1)s$ ноликов из g_0 в префикс строки h_0;

2. Отобразим следующие $2N$ ноликов строки g_0 в h_1;

3. До появления второй единички из g_0 остается вложить еще $(x-1)\cdot s + t$ ноликов. Ясно, что мы сможем отобразить их внутрь строки h_3, при этом мы как раз остановимся перед единичкой в блоке с номером x. Поэтому вкладываем вторую единичку из g_0 в единичку блока с номером x строки h_3

4. Остается вложить $N+1$ ноликов. В строке h_3 у нас еще остается $N-(x-1)s-t$ ноликов, $t+1$ нолик в строке h_4 и значит нам надо вложить оставшиеся $(x-1)s$ ноликов в строку h_1.

Очень важно отметить то, что после этого вложения мы снова возвращаемся в единичку в x-м блоке строки h_1. Это наблюдение показывает, что g_0^k может быть вложено в $h_0^{k+1} = (h_1h_2h_3h_4)^{k+1}$ для каждого $k \geq 1$. В частности $g = g_0^{5N-1} \hookrightarrow h_0^{5N} = h$.

Проведем доказательство в обратную сторону. Предположим, что $g \hookrightarrow h$. Доказательство будем проводить от противного, предположим, что $x \circ \bar{w} \neq t$ для всех $x \in \{0 \dots 2^n - 1\}$. Ясно, что при вложении $g \hookrightarrow h$ не каждый нолик из h может иметь прообраз в g. Давайте оценим общее число таких неиспользованных ноликов. Вложение $g \hookrightarrow h$ содержит $5N-1$ отдельных вложений g_0 в h. В g_0 существуют две единички, между которыми расположено в точности $3N+t$ ноликов. Мы утверждаем, что в h не существует пары единичек между которыми расположено $3N+t$ ноликов. Для того чтобы показать это, мы рассмотрим два случая:

- Пусть левая единичка из g_0 вложена в блок с номером y строки h_1. После прочтения $3N+t$ ноликов в h мы попадем в позицию $t+1$ блока с номером y строки h_3 (не забываем, что $t < s$). Поскольку $y \circ \bar{w} \neq t$, то $(t+1)$-й символ блока с номером y строки h_3 равен 0.

- Пусть теперь левая единичка из g_0 вложена в блок с номером y строки h_3. После прочтения $3N+t$ ноликов в h, мы попадаем в строку h_2, которая вообще не содержит единичек.

Таким образом, мы показали, что при каждом вложении g_0 в h между любыми двумя образами единичек из g_0 должно существовать по крайней мере $3N+t+1$ ноликов в h. Поэтому, для каждого вложения $g_0 = 10^{3N+t}10^{N+1}$ нам необходимо по крайней мере $3N+t+1+N+1 = 4N+t+2$ нолика в h. Так как $g = g_0^{5N-1}$, то нам необходимо по крайней мере $(4N+t+2)\cdot(5N-1) = 5N\cdot(4N+t+1)+(N-t-2) > 5N\cdot(4N+t+1)$ ноликов в h. В последнем неравенстве заметим, что $N = s \cdot 2^n \geq 4s > s+2 > t+2$. Мы получаем противоречие со строением h, мы знаем, что h содержит в точности $5N \cdot (4N+t+1)$ нолика. $\square$

6.1.2 Моделирование логических операций

Предложение 2 (операция отрицания)**.**
Для ПП $\mathcal{G}$ и $\mathcal{H}$ над терминальным алфавитом $\Sigma, |\Sigma| \geq 1$, мы можем построить за полиномиальное время ПП $\mathcal{G}'$ и $\mathcal{H}'$ над терминальным алфавитом Σ такие, что $G \hookrightarrow H \Leftrightarrow G' \not\hookrightarrow H'$.

Доказательство: Пусть $G = g_1 \dots g_k$ и $H = h_1 \dots h_m$. Для $a \in \Sigma$ положим $X_a = (a_1 \dots a_n)^{m+1}$, где $\{a_1 \dots a_n\} \in \Sigma \setminus \{a\}$ (если $n = 0$, тогда $X_a = \varepsilon$). Пусть $a \in \Sigma$ произвольная буква, тогда построим ПП $\mathcal{G}'$ и $\mathcal{H}'$

так, что $G' = Ha = h_1 \dots h_m a$ и $H' = X_{g_1} g_1 \dots X_{g_k} g_k$. Такие ПП могут быть построены за полиномиальное время из $\mathcal{G}, \mathcal{H}$. Для $\mathcal{G}'$ это очевидно. Для $\mathcal{H}'$ мы заменим каждое терминальное правило, выводящее символ a, на новое нетерминальное правило правило $A = X_a \cdot a$. Осталось показать, что ПП $\mathcal{G}', \mathcal{H}'$ являются подходящими.

Предположим, что $G \not\hookrightarrow H$, тогда мы можем представить H в виде: $H = R_1 g_1 \dots R_l g_l R_{l+1}$, где $l < k$ и для $1 \leq i \leq l+1$, строка R_i не содержит букву g_i. Так как $|R_i| \leq m$, для каждого $1 \leq i \leq l+1$ верно, что $R_i \hookrightarrow X_{g_i}$. Поэтому мы можем вложить префикс $H = R_1 g_1 \dots R_l g_l R_{l+1}$ строки G' в префикс $X_{g_1} g_1 \dots X_{g_l} g_l X_{g_{l+1}}$ строки H'. Последняя буква a строки G' может быть также отображена в $X_{g_{l+1}}$(если $a \neq g_{l+1}$, важно, что $|X_{g_{l+1}}| > m$, поэтому R_{l+1} не полностью занимает $X_{g_{l+1}}$) или она может быть отображена в g_{l+1} (если $a = g_{l+1}$).

Теперь предположим, что $G \hookrightarrow H$, тогда мы можем представить H в виде: $H = R_1 g_1 \dots R_k g_k R$, где для $1 \leq i \leq k$ строка R_i не содержит букву g_i. Мы утверждаем что

$$\forall 1 \leq i \leq k : R_1 g_1 \dots R_i g_i \not\hookrightarrow X_{g_1} g_1 \dots X_{g_{i-1}} g_{i-1} X_{g_i} \tag{2}$$

Доказательство осуществляем индукцией по i. В случае $i = 1$ утверждение очевидно, так как g_1 не входит в X_{g_1}. Теперь предположим, что (2) верно для некоторого $i \geq 1$ и более того

$$R_1 g_1 \dots R_{i+1} g_{i+1} \hookrightarrow X_{g_1} g_1 \dots X_{g_i} g_i X_{g_i+1} \tag{3}$$

Напомним, что символ g_{i+1} строки $R_1 g_1 \dots R_{i+1} g_{i+1}$ не входит в суффикс X_{g_i+1} строки $X_{g_1} g_1 \dots X_{g_i} g_i X_{g_i+1}$. Поэтому, из (3) следует, что

$$R_1 g_1 \dots R_i g_i R_k g_k \hookrightarrow X_{g_1} g_1 \dots X_{g_k-1} g_{k-1} X_{g_k},$$

следовательно

$$R_1 g_1 \dots R_i g_i R_{i+1} \hookrightarrow X_{g_1} g_1 \dots X_{g_{i-1}} g_{i-1} X_{g_i}.$$

Но это противоречит (3).

Для $i = k$ из (3) следует $R_1 g_1 \dots R_k g_k \not\hookrightarrow X_{g_1} g_1 \dots X_{g_{k-1}} g_{k-1} X_{g_k}$. Но тогда $G' = R_1 g_1 \dots R_k g_k R a \not\hookrightarrow X_{g_1} g_1 \dots X_{g_{k-1}} g_{k-1} X_{g_k} g_k = H'$. $\square$

Из этих двух доказательств следует, что задача **Вложимость сжатого текста** является coNP-трудной.

Предложение 3 (операция И)**.**
Для ПП $\mathcal{G}_1, \mathcal{H}_1, \mathcal{G}_2, \mathcal{H}_2$ над терминальным алфавитом $\Sigma, |\Sigma| \geq 2$, мы можем построить за полиномиальное время ПП $\mathcal{G}, \mathcal{H}$ над терминальным алфавитом Σ такие что

$$G_1 \hookrightarrow H_1 \text{ и } G_2 \hookrightarrow H_2 \Leftrightarrow G \hookrightarrow H$$

Доказательство: Без ограничения общности можно считать, что $\mathcal{G}_1$ и $\mathcal{G}_2$ (соответственно $\mathcal{H}_1$ и $\mathcal{H}_2$) имеют непересекающиеся множества нетерминалов. Пусть S_i (соответственно T_i) начальный нетерминал G_i (соответственно H_i). Пусть $N = 1 + max\{|H_1|, |H_2|\}$. Тогда $\mathcal{G}$ (соотв. $\mathcal{H}$) содержит все правила $\mathcal{G}_1$ и $\mathcal{G}_2$ (соотв. $\mathcal{H}_1$ и $\mathcal{H}_2$) и дополнительное правило вида $S \to S_1 1^N 0 1^N S_2$ (соотв. $T \to T_1 1^N 0 1^N T_2$), где $0, 1 \in \Sigma$. Здесь S (соотв. T) начальный нетерминал $\mathcal{G}$ (соотв. $\mathcal{H}$). Поэтому

$$G = G_1 1^N 0 1^N G_2 \text{ и } H = H_1 1^N 0 1^N H_2.$$

Очевидно, что если $G_1 \hookrightarrow H_1$ и $G_2 \hookrightarrow H_2$, тогда $G \hookrightarrow H$. В обратную сторону, заметим, что если $G_1 1^N 0 1^N G_2$ может быть вложено в $H_1 1^N 0 1^N$, тогда в силу выбора N, 0-ь в позиции $|G_1| + N + 1$ в строке $G_1 1^N 0 1^N G_2$ не может быть отображен ни в префикс H_1, ни в суффикс H_2 строки H. Поэтому, этот 0-ь отобразится в 0-ь в позиции $|H_1| + N + 1$ строки $H_1 1^N 0 1^N H_2$. Откуда следует, что $G_1 \hookrightarrow H_1$ и $G_2 \hookrightarrow H_2$. □

Предложение 4 (операция ИЛИ)**.**
Для ПП $\mathcal{G}_1, \mathcal{H}_1, \mathcal{G}_2, \mathcal{H}_2$ над терминальным алфавитом $\Sigma, |\Sigma| \geq 2$, мы можем построить за полиномиальное время ПП $\mathcal{G}, \mathcal{H}$ над терминальным алфавитом Σ такие, что

$$G_1 \hookrightarrow H_1 \text{ или } G_2 \hookrightarrow H_2 \Leftrightarrow G \hookrightarrow H.$$

Доказательство: Без ограничения общности считаем, что $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{H}_1, \mathcal{H}_2$ имеют попарно непересекающиеся множества нетерминалов. Пусть S_i (соотв. T_i) начальный нетерминал $\mathcal{G}_i$ (соотв. $\mathcal{H}_i$). Пусть $N = 1 + |G_1| + |G_2|$. Тогда $\mathcal{G}$ содержит все правила $\mathcal{G}_1$ и $\mathcal{G}_2$ и дополнительное правило $S = S_1 \cdot 0 \cdot 1^N \cdot 0 \cdot S_2$. ПП $\mathcal{H}$ содержит все правила $\mathcal{G}_1, \mathcal{H}_1, \mathcal{G}_2, \mathcal{H}_2$ и дополнительное правило $T = T_1 \cdot 0 \cdot 1^N \cdot S_1 \cdot 0 \cdot S_2 \cdot 1^N \cdot 0 \cdot T_2$. В итоге мы получаем, что

$$G = G_1 \cdot 0 \cdot 1^N \cdot 0 \cdot G_2 \text{ и } H = H_1 \cdot 0 \cdot 1^N \cdot G_1 \cdot 0 \cdot G_2 \cdot 1^N \cdot 0 \cdot H_2$$

Очевидно, что если $G_1 \hookrightarrow H_1$ или $G_2 \hookrightarrow H_2$, тогда $G \hookrightarrow H$. Для доказательства в обратную сторону предполагаем, что $G = G_1 \cdot 0 \cdot 1^N \cdot 0 \cdot G_2$ может быть вложено в $H = H_1 \cdot 0 \cdot 1^N \cdot G_1 \cdot 0 \cdot G_2 \cdot 1^N \cdot 0 \cdot H_2$. Обозначим через B – блок единичек длины N строки G. Тогда возникают следующие случаи:

- Если единичка из B отображена в H_1 – префикс строки H, тогда $G_1 \hookrightarrow H_1$;
- Если единичка из B отображена в блок единичек длины N строки H, тогда 0 в позиции $|G_1| + 1$ строки G не может быть отображен правее нолика в позиции $|H_1| + 1$ строки H. И тогда $G_1 \hookrightarrow H_1$;
- Если единичка из B отображена в H_2 – суффикс строки H или во второй блок из единичек длины N строки H, то аналогично предыдущим двум случаям мы получим, что $G_2 \hookrightarrow H_2$;
- Остается случай, когда каждая единичка из B отображена в подстроку $G_1 \cdot 0 \cdot G_2$, но он невозможен, так как $N > |G_1 \cdot G_2|$; $\square$

6.1.3 Θ_2^p-трудность

Напомним, что класс Θ_2^p – это класс всех задач, которые могут быть приняты детерминированной полиномиальной машине Тьюринга с доступом к NP-оракулу, который отвечает на все вопросы параллельно.

Предложение 5.
Если $A \subseteq \{0,1\}^*$ NP-полная, тогда следующая задача является Θ_2^p-полной:

Вход: *Логическое выражение C с начальными элементами, помеченными словами над бинарным алфавитом.*

Вопрос: *Вычисляет ли C значения $true$, когда каждый входной элемент g, который помечен словом $w \in \{0,1\}^*$, вычисляет значение $true$ (соотв. $false$) если $w \in A$ (соотв. $w \notin A$).*

Доказательство: Для доказательства принадлежности Θ_2^p покажем, что мы можем вычислить все входные элементы C параллельно, используя язык A в качестве оракула. Тогда, все выражение может быть вычислено за полиномиальное время. Следующее утверждение взято из работы [11]: Задача, в которой спрашивается для заданного набора строк

$w_1, w_2, \ldots, w_n \in \{0,1\}^*$, является ли число $|\{i \ : w_i \in A\}|$ нечетным, является Θ_2^p-трудной. Если мы возьмем логическое выражение для контроля четности, то эта проблема может быть легко закодирована в логическое выражение с входами из A в качестве начальных элементов. □

Теорема 4 (о Θ_2^p-трудности)**.**
Даже для ПП с бинарным терминальным алфавитом, задача **Вложимость сжатого текста** *является Θ_2^p-трудной.*

Доказательство: Пусть C выражение, чьи начальные элементы помечены входами из NP-полной задачи **Сумма размеров**. С помощью удвоения аргумента, мы можем считать, что элементы отрицания относятся непосредственно к переменным, но не к выражениям. Определим по индукции для каждого элемента c строки $u(c)$ и $v(c)$ и потом проверим, что:

- c вычисляет $true$ тогда и только тогда, когда $u(c) \hookrightarrow v(c)$
- $u(c)$ и $v(c)$ могут быть порождены ПП "маленького" размера

Если c – входной элемент, не содержащий отрицания и помеченный входом I из задачи **Сумма размеров**, тогда $u(c) = g$ и $v(c) = h$, где g и h две строки, которые построены по I в доказательстве теоремы о NP-трудности (см. стр. 51). Если c – входной элемент, содержащий отрицание и помеченные входом I из задачи **Сумма размеров**, тогда мы снова по I строим строки g и h, как в доказательстве теоремы о NP-трудности. Далее мы применяем к g и h конструкции из доказательства предложений о логических операциях (см. стр. 53-56) и обозначаем результирующие строки $u(c)$ и $v(c)$ соответственно. Для И и ИЛИ элементов мы используем конструкции из предложения об операции И, предложения о операции ИЛИ: Если c это И элемент с входом c_1 и c_2, тогда

$$u(c) = u(c_1) \cdot 1^N \cdot 0 \cdot 1^N \cdot u(c_2) \text{ и } v(c) = v(c_1) \cdot 1^N \cdot 0 \cdot 1^N \cdot v(c_2), \quad (4)$$

$$\text{где } N = 1 + max(|v(c_1)|, |v(c_2)|).$$

Если c является ИЛИ элементом со входом c_1, c_2, тогда

$$u(c) = u(c_1) \cdot 0 \cdot 1^N \cdot 0 \cdot u(c_2) \text{ и } v(c) = v(c_1) \cdot 0 \cdot 1^N \cdot u(c_1) \cdot 0 \cdot u(c_2) \cdot 1^N \cdot 0 \cdot v(c_2) \quad (5)$$

где $N = 1 + |u(c_1)| + |u(c_2)|$.

Из теоремы о NP-трудности и предложений о логических операциях получаем, что C вычисляет $true$ тогда и только тогда, когда $u(o) \hookrightarrow v(o)$, где o – выходной элемент C.

Остается понять, что для каждого элемента c, строки $u(c)$ и $v(c)$ могут быть порождены ПП размера полиномиально ограниченного размером выражения (который равен числу элементов плюс размер входа задачи **Сумма размеров**). Заметим, что если мы определим $n(c) = max(|u(c)|, |v(c)|)$, тогда мы получаем $h(c) \leq 8 \cdot max(n(c_1), n(c_2)) + 5$ в случае если c является И или ИЛИ элементом со входом c_1, c_2. Это означает, что $n(c)$ экспоненциально ограничена размером выражения C. Более того, мы можем вычислить двоичное представление длин $|u(c)|$ и $|v(c)|$ для каждого элемента c за полиномиальное время. Поэтому, мы можем построить ПП полиномиального размера для факторов 1^N в (4) и (5). Отсюда следует, что для каждого элемента $c, u(c)$ и $v(c)$ могут быть порождены ПП полиномиального размера. □

Введем задачи **Наибольшая общая подпоследовательность** (соотв. **Наименьшая общая подпоследовательность**), которые для конечного множества строк R и $n \in \mathbb{N}$ спрашивают, существует ли строка w такая, что $|w| \geq n$ и для всех $v \in R$ $w \hookrightarrow v$ (соотв. $|w| \leq n$ и для всех $v \in R$ $v \hookrightarrow w$). Известно, что эти задачи являются NP-полными, но для $|R| = 2$ они могут быть решены за полиномиальное время. Если все строки в R представлены в виде ПП, то мы можем решить эти задачи в PSPACE.

Следствие. Задачи **Наибольшая общая подпоследовательность**, **Наименьшая общая подпоследовательность** для строк представленных в виде ПП являются Θ_2^p-трудными, даже если $|R| = 2$.

Доказательство: Для $u, v \in \Sigma^*$ мы имеем $u \hookrightarrow v$ тогла и только тогда, когда на входе $(\{u, v\}, |u|)$ (соотв. $(\{u, v\}, |v|)$) в задачу **Наибольшая общая подпоследовательность** (соотв. **Наименьшая общая подпоследовательность**) мы получим ответ $true$. □

Замечание Представленные выше утверждения взяты из работы Ю. Лифшица и М. Лори [25].

6.2 Сжатое расстояние Хэмминга

Определение. *Расстоянием Хэмминга* между двумя строками одинаковой длины называется количество несовпадающих букв на одинаковых позициях текста. Поэтому над парой сжатых строк, которые выводят тексты одинаковой длины, мы можем ввести операцию взятия расстояния Хэмминга и будем ее обозначать через $HD(\mathcal{T}, \mathcal{P})$.

Определение. Будем говорить, что функция *принадлежит классу* #P, если существует такая недетерминированная полиномиальная машина Тьюринга M, что значением функции будет число принимающих веток M на соответствующих входных данных. Иначе говоря, существует полиномиально-вычислимая функция $G(x, y)$, такая что $f(x) = \#\{y : G(x, y) =$"да"$\}$.

Будем говорить, что функция f имеет [1]-*Тьюринг сведение* к функции g, если существуют такие полиномиально-вычислимые функции E и D, что $f(x) = D(g(E(x)))$. Функция называется #P-*полной* (относительно [1]-Тьюринг сведений), если она принадлежит классу #P, и любая другая функция этого класса имеет к ней [1]-Тьюринг сведение.

Теорема 5 (о #P-полноте)**.**
Операция взятия расстояния Хэмминга для пары сжатых текстов является #P*-полной.*

Доказательство: **Принадлежность** #P**.** Для расстояния Хэмминга в качестве функции G мы можем взять сравнение строк по одной позиции: $G(T, P, y) =$ "да", если $T[y] \neq P[y]$. Тогда количество y, дающих ответ "да", в точности равно расстоянию Хэмминга. Функция G – полиномиально вычислима, так как мы легко можем распаковать пару символов и сравнить их.

#P**-полнота.** Напомним формулировку #P-полной версии задачи **Сумма размеров**: даны целые числа $w_1, \ldots, w_n, t$ в двоичной записи. Требуется определить, сколько существует наборов $x_1, \ldots, x_n \in \{0, 1\}$ таких, что $\Sigma_{i=1}^{n} x_i \cdot w_i = t$? Иначе говоря, сколько подмножеств $W = \{w_1, \ldots, w_n\}$ имеют сумму элементов равную t? Для доказательства полноты задачи **Сжатого расстояния Хэмминга** достаточно свести к ней другою полную задачу в рассматриваемом классе. Построим [1]-Тьюринг сведение от **Суммы размеров** к **Сжатому расстоянию Хэмминга**. Зафиксируем входные данные для суммы размеров. Построим две ПП так, чтобы по расстоянию Хэмминга между строками, которые они вы-

водят, можно было определить ответ для **Суммы размеров**.

Идея конструкции заключается в следующем: пусть $s = w_1 + \cdots + w_n$. Оба текста будут иметь длину $(s+1)2^n$. Мысленно мы будем представлять их себе как последовательность 2^n блоков по $s+1$ символов каждый. Первый текст T будет кодировать число t. Все его блоки будут иметь вид: «$0 \ldots 0\ 1\ 0 \ldots 0$», где единственная единичка стоит в позиции $t+1$. Блоки второго текста S будут соответствовать всем возможным подмножествам W. В каждом из них единственная единичка будет стоять на позиции, равной сумме элементов подмножества плюс один. Формально, тексты T и S можно записать так: $T = (0^t \cdot 1 \cdot 0^{s-t})^{2^n}, P = \prod(0^{\overline{x} \circ \overline{w}})10^{s-\overline{x} \circ \overline{w}})$, где $\overline{x} \circ \overline{w} = \Sigma_{i=1}^{n} x_i \cdot w_i$. Похожие строки мы уже встречали в предыдущей главе. Строка P (будем называть ее *строка Лори*) впервые появилась в работе М. Лори [15]. Лори доказал, что зная входные параметры задачи о сумме размеров, мы можем построить ПП полиномиального размера, описывающие строки P и T. Заметим теперь, что расстояние Хэмминга между T и P равно удвоенному числу тех подмножеств W, чья сумма не равна t. Таким образом, ответ для суммы размеров можно получить по формуле: $2^n - \frac{1}{2}HD(\mathcal{T}, \mathcal{P})$. $\square$

Замечание. Представленные выше утверждения взяты работы Ю. Лифшица [26].

6.3 Операция перетасовки букв

В этом параграфе мы будем предполагать, что $\Sigma = \{0, 1\}$. Сначала рассмотрим вспомогательную задачу, для которой существует полиномиальный алгоритм.

6.3.1 Задача Сжатые рациональные преобразования

Определение. *Детерминированный трансдьюсер* это – набор

$$A = (\Sigma, \Gamma, Q, q_0, \delta),$$

где Σ – входной алфавит, Γ – выходной алфавит, Q – множество состояний, $q_0 \in Q$ – стартовое состояние и $\delta = (\delta_Q, \delta_\Gamma)$ – функция перехода, где $\delta_Q : Q \times \Sigma \to Q$ и $\delta_\Gamma : Q \times \Sigma \to \Gamma^*$. Мы обозначим $(\delta_Q(q,a), \delta_\Gamma(q,a))$ через $\delta(q,a)$. Расширим функцию δ_Q на Σ^*, положив $\delta_Q^*(q, \varepsilon) = q$ и $\delta_Q^*(q, wa) = \delta_Q(\delta_Q^*(q,w), a)$ для каждой $w \in \Sigma^*$. Расширение для функции δ_Γ немного

сложнее: мы положим $\delta^*_\Gamma(q,\varepsilon)=\varepsilon$ и $\delta^*_\Gamma(q,wa)=\delta^*_\Gamma(q,w)\delta_\Gamma(\delta^*_Q(q,w),a)$, где $w\in\Sigma^*$.

Рассмотрим множество $S=\{w \mid \delta_\Gamma(q,a)=w, q\in Q, a\in\Sigma\}$ и определим *размер трансдьюсера* как $|A|=|Q|+\sum_{w\in S}|w|$. Теперь мы готовы сформулировать задачу:

Задача: **Сжатые рациональные преобразования**:

Вход: детерминированный трансдьюсер A и ПП $\mathcal{T}$, которая выводит текст T.

Выход: ПП, порождающая $A(T)$.

Теорема 6 (о рациональных преобразованиях)**.**
Пусть заданы трансдьюсер A и ПП $\mathcal{T}$, тогда существует $O(|A|\cdot|\mathcal{T}|)$ алгоритм, который решает задачу о **Сжатых преобразованиях**.

Доказательство: Зафиксируем $A=(\Sigma,\Gamma,Q,q_0,\delta)$ и $|\mathcal{T}|$. Вычислим RT-таблицу такую, что в клетке (T_k,q), где $T_k\in\mathcal{T}$ и $q\in Q$ мы храним значение $RT(T_k,q)=\delta^*_Q(q,T_k)$. RT-таблица может быть вычислена за время $O(|Q|\cdot|\mathcal{T}|)$: начиная с правил меньшей длины, мы фиксируем $T_k=T_l\cdot T_r\in\mathcal{T}$ и для всех $q\in Q$ вычисляем значение $RT(T_k,q)$ как $RT(T_r,RT(T_l,q))$. Если $T_k=a$, где $a\in\Sigma$, тогда $RT(T_k,q)=\delta_Q(q,a)$.

Построим новую ПП $\mathcal{S}$ с помощью преобразования каждого правила $T_k\in\mathcal{T}$ в $|Q|$ правил $T(k,q)$ вида:

$$T(k,q)=\begin{cases} T(l,q)T(r,T(T_l,q)), \text{ если } T_k=T_l\cdot T_r \\ \mathcal{S}(q,a), \text{ если } T_k=a, a\in\Sigma, \end{cases}$$

где $\mathcal{S}(q,a)$ это ПП такая, что $S(q,a)=\delta_\Gamma(q,a)$

Если положить $T(n,q_0)$ в качестве последнего правила $\mathcal{S}$, мы получим $S=A(T)$ по построению. Каждое $\mathcal{S}(q,a)$ имеет размер не более $|\delta_\Gamma(q,a)|$, следовательно $\mathcal{S}=O(|A|\cdot|\mathcal{S}|)$. $\square$

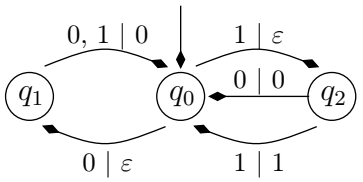
Из теоремы можно сделать вывод, что свойство сжатия строк прямолинейными программами сохраняется при применении рациональных преобразований к строке. Более формально:

Следствие. Зафиксируем рациональный трансдьюсер A, тогда $g(A(w))=O(g(w))$ для любого $w\in\Sigma^*$, где $g(w)$ – минимальный размер ПП, которая порождает строку w.

Пример. Рассмотрим следующий рациональный трандьюсер:

$$A=(\{0,1\},\{0,1\},\{q_0,q_1,q_2\},q_0,\delta),$$

где $\delta(q_0, 0) = (q_1, \varepsilon), \delta(q_1, 0) = \delta(q_1, 1) = \delta(q_2, 0) = (q_0, 0), \delta(q_0, 1) = q_2, \varepsilon$ и $\delta(q_2, 1) = (q_0, 1)$.



Такой трандьюсер читает символы строки над $\{0, 1\}^*$ парами и, если встретилась строка "11", заменяет ее на "1", а остальные строки: "00", "01", "10" заменяет на "0".

6.3.2 Операция перетасовки букв

Определение. Введем новые операции над строками. Пусть $P, T \in \{0, 1\}^N$, где $N > 0$, тогда $(P \curlywedge T)[i] = P[i] \wedge T[i]$ для $i = 1 \dots N$ – операция *поэлементного И*, а $P \curlyvee T = P[1]T[1]P[2]T[2] \dots P[N]T[N]$ – операция *перетасовки букв*. Введем *обратные* операции L и R для перетасовки букв: $L(w) = w[1]w[3] \dots w[2N-1], R(w) = w[2]w[4] \dots w[2N]$, где $w \in \{0, 1\}^{2N}$.

Определение. Для любой строки $w \in \Sigma^*$ обозначим через $b(w)$ – такое число, чье двоичное представление совпадает с w.

Для начала докажем две технических леммы, одна из которых позволяет преобразовать конструкции, использующие логические выражения, в конструкции, использующие ПП.

Лемма 27 (техническая)**.**
Для каждой строки $w \in \{0.1\}^*, |LZ(w)| \geq |Fact(w) \cap 10^*1|$, где $Fact(w)$ – множество всех подстрок строки w.

Доказательство: LZ-факторизация строки w содержит LZ-фактор для каждого первого вхождения 10^t1 с различным значением t. В самом деле, каждая строка 10^t1 не может быть фактором 10^s1 при $s \neq t$. Поэтому алгоритм факторизации вынужден будет создать новый LZ-фактор, соответствующий строке 10^t1, в итоге $|LZ(w)| \geq |Fact(w) \cap 10^*1|$. □

Лемма 28 (техническая)**.**
Пусть C логическое выражение, вычисляющее булеву функцию $f(x)$ от переменных $x_1, \dots, x_n$. Тогда существуют ПП $\mathcal{G}$ и $\mathcal{H}$ такие, что

- $|\mathcal{G}|, |\mathcal{H}| = |C|^{O(1)}$;
- $|G| = |H| = 2^{n+m}$, *где* $m = n^{O(1)}$;
- $f(x) = 1 \Rightarrow \exists!\ z \in \{0,1\}^m : G[b(xz)] = H[b(xz)] = 1$;
- $f(x) = 0 \Rightarrow \forall z \in \{0,1\}^m : G[b(xz)] \wedge H[b(xz)] = 0$;

Доказательство: Из C легко построить 3-КНФ φ, которая вычисляет f, с помощью добавления логических переменных $y_1, \ldots, y_{|C|}$ к начальным $x_1, \ldots, x_n$ и введения клозов вида $y_k = x_i \circ x_j$ для каждого элемента k, вычисляющего $x_i \circ x_j$, где $\circ \in \{\vee, \wedge\}$. Тогда $\varphi(x, y) = 1$ для уникальной пары (x, y) такой, что $f(x) = 1$, тогда как $f(x) = 0$ на всех остальных значениях.

Используя сведение **3-SAT** к **Сумме размеров**, мы можем свести $\varphi(x, y)$ ко входу $I_N(\alpha, \beta, \gamma, t)$ **Суммы размеров**, где $|\gamma| = n^{O(1)}$ и при этом выполняется:

- если $\varphi(x, y) = 1$, тогда $\exists!\ w : x \cdot \alpha + y \cdot \beta + w \cdot \gamma = t$;
- если $\varphi(x, y) = 0$, тогда $x \cdot \alpha + y \cdot \beta + w \cdot \gamma \neq t$ для каждой w;

Пусть $\xi(I_n)$ и $\xi'(I_n)$ строки Лори (определение сток Лори смотри в параграфе Сжатое расстояние Хэмминга), которые ассоциированы с входом $I_n(\alpha, \beta, \gamma, t)$. Тогда у нас есть два слова длиной $2^{n+m+|\gamma|}$ такие, что

- $f(x) = 1 \Rightarrow \exists!\ z \in \{0,1\}^{m+|\gamma|} : \xi(I_n)[b(xz)] = \xi'(I_n)[b(xz)] = 1$;
- $f(x) = 0 \Rightarrow \forall z \in \{0,1\}^{m+|\gamma|} : \xi(I_n)[b(xz)] \wedge \xi'(I_n)[b(xz)] = 0$;

И мы уже знаем, что такие строки могут быть попрождены ПП полиномиального размера относительно входа задачи **Сумма размеров**. □

Теорема 7 (о поэлементном И)**.**
Существует бесконечное число пар слов (w, w') *одинаковой длины такие, что* $g(w), g(w') = n^{O(1)}$ *и* $g(w \curlywedge w') = O(2^n)$.

Доказательство:

Пусть C логическое выражение, которое вычисляет булеву функцию

$$f(x, y) = \begin{cases} 1, & \text{если } b(x) = b(y)^2, \\ 0, & \text{если } b(x) \neq b(y)^2. \end{cases}$$

Выражение C может быть представлено с помощью $O(|x|^2)$ переменных и $O(|x|^2)$ 3-клозов.

Рассмотрим вход $I_n(\alpha, \beta, \gamma, t)$ **Суммы размеров**, определенный для C как во второй технической лемме (см. стр. 62). Зафиксируем представление $q(s)$ – s-го полного квадрата s^2 и положим $z(s)$ уникальная строка такая, что $q(s) \cdot \alpha + z(s) \cdot (\beta \odot \gamma) = t$ (где через $\beta \odot \gamma$ обозначена конкатенация двоичных векторов). Пусть $\xi(I_n)$ и $\xi'(I_n)$ – строки Лори, ассоциированные с I_n, и пусть $\xi = \xi(I_n) \wedge \xi'(I_n)$. Позиция t_s s-й единички в ξ равно $b(q(s)z(s))$. Так как $z(s)$ уникально и $(s+1)^2 - s^2 > 1$ для $s > 0$, мы получаем

$$s^2 \cdot 2^M \leq t_s < s^2 \cdot 2^M + 2^{M+1}, \text{ где } M = |z(s)|.$$

Отсюда следует, что

$$(2s+1)2^M - 2^{M+1} \leq t_{s+1} - t_s < (2s+1)2^M + 2^{M+1}.$$

Поэтому $t_{s+1} - t_s \neq t_{j+1} - t_j$ для $s \neq j$. В результате зафиксируем $\widehat{s} = max\{s : s^2 < 2^n\}$, при этом выполнено:

$$|Fact(\xi) \cap 10^*1| = |\{10^{t_{s+1}-t_s}1 : 1 \leq s \leq \widehat{s}\}| = \widehat{s} \geq 2^{n/2} - 1.$$

По первой технической лемме (см. стр. 62) получаем, что $|LZ(\xi)| \geq |Fact(\xi) \cap 10^*1|$. Следовательно $g(\xi) = O(2^{n/2})$, тогда как $g(\xi(I_n)), g(\xi'(I_n)) = n^{O(1)}$ по второй технической лемме. □

Следствие. Существует бесконечно число пар строк (w, w') одинаковой длины таки, что $g(w), g(w') = n^{O(1)}$ и $g(w \curlyvee w') = O(2^n)$

Доказательство: В Примере представлен рациональный трандьюсер Q такой, что $A(x \curlyvee y) = x \curlywedge y$. Следовательно результат теоремы о поэлементном И может быть расширен на операцию перетасовки букв. □

Замечание. Представленные выше утверждения взяты из работы А. Бертони, К. Шофрю и Р. Радичьони [4].

7 Заключение

В данной работе мы рассмотрели широкий класс классических строковых задач в терминах прямолинейных программ. Для большей части задач мы смогли представить полиномиальные алгоритмы (которые являются наилучшими из известных нам). Для оставшихся задач мы показали, что для них не существует полиномиального алгоритма в терминах сжатых строк. Также была приведена оценка сложности этих задач относительно полиномиальной иерархии.

8 Благодарности

- **Волкову Михаилу Владимировичу** за приятные беседы и равномерное отношение к жизни.
- **Ананичеву Дмитрию Сергеевичу** и всему составу семинара «Компьютерные науки» за то, что слушали мои доклады, задавали много «странных вопросов» и находили ошибки в моих рассуждениях.
- **Юре Лифщицу** за оптимизм и советы.
- Родителям (**Александре Дмитриевне** и **Александру Алексеевичу**) за терпение и понимание.
- Брату **Саше** и его жене **Марине** за поддержку и внимание.
- **Оле** за минуты отдыха, проведенные в ее обществе.
- **Негодаеву Мишке** за его отношение к жизни.

Список литературы

[1] *Alberto Apostolico*, *Dany Breslauer and Zvi Galil*. Optimal parallel algorithms for periods, palindromes and squares. In Proceedings of International Colloquium on Automata, Languages and Programming (ICALP 1992), pp. 296-307, 1992.

[2] $AlbertoApostolicoandDanyBreslauer$. An Optimal $O(loglogn)$ Time Parallel Algorithm for Detecting all Squares in a String. Society of Industrial and Applied Mathematics Journal (SIAM), 1995.

[3] *Alberto Apostolico*, *Dany Breslauer and Zvi Galil* Parallel detection of all palindromes in a string. Theoretical Computer Science Journal, pp. 163-173, 1995.

[4] *Alberto Bertoni*, *Christian Choffrut*, *Roberto Radicioni* Literal Shuffle of Compressed Words. In Proceedings of Theoretical Computer Science 2008 (TCS 2008).

[5] *Amirhood Amir*, *Gary Benson and Martin Farach* Let sleeping files lie: Pattern matching in Z-compressed files. In Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), 1994.

[6] *Ayumi Shinohara*, *Marek Karpinski*, *Wojciech Rytter* Pattern-matching for strings with short descriptions. In Proceeding of Symposium on Combinatorial Pattern Matching (CPM 1995), pp. 205-214. Springer-Verlag, 1995.

[7] *Christos H. Papadimitriou* Computational Complexity. Addison Wesley, 1994.

[8] *Dan Gusfield* Algorithms on Strings, Trees and Sequences. Cambridge University Press, 1997.

[9] *Donald Knuth* The Art of Computing Vol.II: Seminumerical Algorithms. Second Edition. Addison-Wesley 1981.

[10] *Jacob Ziv and Abraham Lempel* A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, pp. 337-343, 1977.

[11] *K. W. Wagner* More complicated questions about maxima and minima, and some closures of NP. Theoretical Computer Science Journal, pp. 53-80, 1987.

[12] *L. Gasieniec, M. Karpinski, W. Plandowski and W.Rytter* Efficient algorithms for Lempel-Ziv encoding (extended abstract). In Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996), pp. 392-403. Springer-Verlag, 1996.

[13] *Marek Karpinski, Wojciech Rytter, Ayumi Shinohara* An efficient pattern-matching algorithm for strings with short descriptions. Nordic Journal of Computing, pp. 172-186, 1997.

[14] *Markus Lohrey* Word problems on compressed word. In Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), pp. 906-918. Springer-Verlag, 2004.

[15] *Markus Lohrey* Word problems and membership problems on compressed words. Society of Industrial and Applied Mathematics Journal (SIAM), pp. 1210-1240, 2006.

[16] *Martin Farach and Mikkel Thorup* String matching in Lempel-Ziv compressed strings. In Proceedings of the 27-th Annual ACM Symposium on Theory of Computing (STOC 1995), pp. 703-712, ACM Press, 1995.

[17] *Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda* An improved pattern matching algorithm for strings in terms of straight line programs. In Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997), pp. 1-11. Springer-Verlag, 1997.

[18] *M. Hirao, A. Shinohara, M. Takeda, and S. Arikawa* Fully compressed pattern matching algorithm for balanced straight-line programs. In Proceedings of 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000), pp. 132-138. IEEE Computer Society, 2000.

[19] *Maxime Crochemore and Wojciech Rytter*. Text Algorithms. Oxford University Press, New York, 1994.

[20] *Michael Garey and David Johnson* Computers and Intractability: a Guide of the Theory of NP-completness. Freeman, 1979.

[21] *P. Cegielski, I. Guessarian, Y. Lifshits and Y. Matiyasevich* Window sunsequence problems for compressed texts. In Proceedings of 1st International Symposium Computer Science in Russia (CSR 2006), pp. 127-136. Springer-Verlag, 2006

[22] *T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa* Collage system: a unifying framework for compressed pattern matching. Theoretical Computer Science Journal, pp. 253-272, 2003.

[23] *Y. Ishida, S. Inenaga, A. Shinohara, M. Takeda* Full incremental LCS computation. In Proceedings of Fundamentals of Computation Theory (FCT 2005), pp. 563-574, Springer-Verlag 2005.

[24] *Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa* Pattern matching in texts compressed by using antidictionaries. In Proceeding of Symposium on Combinatorial Pattern Matching (CPM 1999), pp. 37-49. Springer-Verlag, 1999.

[25] *Yury Lifshits and Markus Lohrey* Quering and embedding compressed texts. In Proceeding of International Symposium on Mathematical Foundations of Computer Science (MFCS 2006), pp. 681-692. Springer-Verlag, 2006.

[26] *Yury Lifshits*. Processing Compressed Texts: A Tractability Border, In Proceedings of Symposium on Combinatorial Pattern Matching (CPM 2007), pp. 228-240. Springer 2007.

[27] *W. Matsubara, S. Inenega, A. Ishino, A. Shinohara, T. Nakamura, K. Hashimoto* Computing Longest Common Substring and All Palindromes from Compressed Strings. In Proceeding of Software Seminar (SOFSEM 2008) 2008.

[28] *Wojciech Plandowski* Testing equivalence of morphisms on context-free languages. In Proceedings of Second Annual European Symposium on Algorithms (ESA 1994), pp. 460-470. Springer-Verlag, 1994.

[29] *Wojciech Rytter* Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science Journal, pp. 211-222, 2003.