Конспект лекций по строковым алгоритмам

Глинских Георгий

22 июня 2025 г.

Глава 1

Поиск подстроки в строке

1.1 Вводные замечания

Строки – конечные последовательности символов над множеством, называемым алфавитом. Здесь и далее я буду полагать, что алфавит – множество чисел $\{1, \ldots, \sigma\}$ и что строки индексируются с 1[1]:

$$w = w(1)w(2)\ldots w(n) = w(1, n).$$

Введем понятия периода и грани строки.

Определение 1.1.1. *Периодом строки назовем число p, такое, что*

$$\forall i : s(i) = s(i + p).$$

Утверждение 1.1.1. *Следующие определения равнозначны:*

1. *p – период строки w.*
2. *$s(p + 1, n)$ и суффикс, и префикс w.*
3. *Существует слова a, b: b префикс a и $w = a^k b$.*

Доказательство. $1 \Rightarrow 2$. Посимвольно сравнивая, $s(1, n - p) = s(p + 1, n)$.

$2 \Rightarrow 3$. Положим $a = s(1, p)$. Тогда видно, что $s(p + 1, 2p) = s(1, p) = a$, Аналогично

$$\forall l : s((l - 1)p + 1, lp) = a.$$

Остается взять $b = s(kp + 1, n)$, где k – наибольшее из чисел l.

$3 \Rightarrow 1$. Очевидно.

□

Определение 1.1.2. *Гранью строки назовем суффикс, одновременно являющийся префиксом.*

[1]Так удобнее для математиков, но хуже для программистов.

Утверждение 1.1.2. *Число $0 < p \leq n$ – период s согда $s(1, n-p)$ – грань s.*

Доказательство. Заметим, что $s(1, n-p) = s(p+1, n)$ по условию, и тогда достаточно воспользоваться пунктом 2 предыдущего утверждения. □

> Важно мыслить объекты, которые обрабатываются алгоритмами, очень (очень!) большой размерности. Тогда будет появляться нужная интуиция.

1.2 Алгоритм Крошмора-Перрена

При помощи периода можно улучшить обычный алгоритм поиска подстроки в строке. Далее будем называть его Ord, его сложность $T(\mathrm{Ord}) = O(|\mathtt{text}||\mathtt{pat}|)$.

> Инвариант цикла в Ord – вхождения с началом в $\mathtt{text}(1, i)$ рассмотрены.

Пусть теперь известен p – минимальный период строки `pat`. Тогда этот алгоритм можно улучшить для нахождения серии вхождений (вхождения, расположенные достаточно близко друг к другу). Заметим, что вхождения находятся на расстоянии не меньше p (иначе противоречие с минимальностью периода), следовательно, при нахождениии вхождения, можно не смотреть на следущие p возможных позиций. Получим алгоритм Per.

> Инвариант цикла в Per – для нахождения вхождения остается рассмотреть $|\mathtt{pat}| - b + 1$ позиций. Иными словами, $\mathtt{pat}(1, b) = \mathtt{text}(i, i+b)$.

Теперь рассмотрим наконец алгоритм Крошмора-Перрена. Его будем обозначать TW: в англоязычной литературе его называют two-way алгоритмом. Он быстрый: $T(\mathrm{TW}) = O(|\mathtt{text}| + |\mathtt{pat}|), S(\mathrm{TW}) = O(1)$.

Определение 1.2.1. *Локальным периодом разложения $w = ab$ назовем число q такое, что*

$$\forall i \in \{|a| - q, \ldots, |a|\}: \quad s(i) = s(i+q).$$

Заметим, что число p – период w будет локальным периодом для любого разложения.

Определение 1.2.2. *Критическим назовем разложение $w = ab$, для которого минимальный локальный период совпадает с периодом строки w.*

Утверждение 1.2.1. *У любой строки есть критическое разложение $w = ab$: $|a| = u < p$, где p – период w.*

Доказательство. Заметим сначала, что любой локальный период q для критического разложения (a, b) с условием $q < |b|$ делится на p. Из критичности, $q \geq p$. Тогда $(xy)^k x$ с $|xy| = p, |x| < p$ будет префиксом b. Но тогда x будет суффиксом a и префиксом b. Пришли к противоречию: удалось найти локальный период $|x| < p \leq q$.

□

Рассмотрим алгоритм TW1. Идейно он работает так: зафиксируем критическое разложение $\texttt{pat} = ab$ как в утверждении 1.2.1 и сначала пытаемся найти вхождение b слева направо, после чего проверим справа налево, будет ли это вхождение иметь вид $ab = \texttt{pat}$.

В начале каждой итерации главного цикла соблюдается инвариант $\texttt{pat}(1, b) = \texttt{text}(i, i + b)$ (аналогично алгоритму Per).

Утверждение 1.2.2. *Алгоритм TW1 находит все вхождения* $\texttt{pat}$ *в* $\texttt{text}$ *со сложностью*

$$T(\text{TW1}) = O(|\texttt{text}|), \quad S(\text{TW1}) = O(1).$$

Доказательство. Допустим, i пробегает позиции $\{i_k\}$, и в позиции $i_k < i' < i_{k+1}$ было вхождение, которое мы не выведем. Одно из двух: мы при сканировании перескочили этот индекс или мы не распознали вхождение по этому индексу.

Если мы перешли к i_{k+1} через строку 8, то мы не можем пропустить вхождение в силу критичности разложения и $|a| < p$. Формально, $i' - i_k < d - u$, тогда по утверждению 1.2.1, $i - i_k = Ap$. Следовательно, $\texttt{text}(i + c) \neq \texttt{pat}(c) = \texttt{pat}(c - Ap) = \texttt{text}(i' - c - i')$. Противоречие.

Если мы перешли к i_{k+1} через строку 15, то мы не можем пропустить в силу того, что p – наименьший локальный период.

Итак, алгоритм находит все вхождения.

Мы используем лишь несколько переменных, откуда $S(\text{TW1}) = O(1)$. Для оценки времени работы $T(\text{TW1})$, заметим, что в 6 и 12 строке мы не касаемся символа дважды. □

Далее необходимо найти разложение и период. Идейно надо сделать пользоваться алгоритмом TW2.

Утверждение 1.2.3. *Алгоритм TW3 находит все вхождения* $\texttt{pat}$ *в* $\texttt{text}$ *со сложностью*

$$T(\text{TW1}) = O(|\texttt{text}|), \quad S(\text{TW1}) = O(1).$$

Доказательство. Ясно, что $p' < p$. Причем p' – период $\texttt{pat}$ согда верно условие в строке 2. Остальное доказательство повторяет доказательство утверждения 1.2.2. □

Утверждение 1.2.4. *В алгоритме* TW2

$$q = max\{|a|, |b|\} + 1 < p.$$

Рис. 1.1: Обычный алгоритм поиска подстроки в строке (Ord)

```
i = 1
пока i + |pat| <= |text|:  # O(|text|)
  если text(i, i + |pat|) = pat(1, |pat|):  # O(|pat|)
    верни i
  i += 1
```

Рис. 1.2: Улучшенный алгоритм поиска подстроки в строке (Per)

```
i = 1, b = 1
пока i + |pat| <= |text|:  # O(|text| / p)
  c = max { c : pat(b, c) = text(i + b, i + c) }  # O(|pat|)
  если c < |pat|:
    i += 1, b = 1
  иначе:  # c = |pat|
    верни i
    i += p, b = |pat| - p + 1
```

Рис. 1.3: Алгоритм TW1

```
# вход: text: str, pat: str, u: num, p: num
# ограничения: u < p, u критическая для pat, p - период pat
i = 1, b = 1
пока i + |par| <= |text|:
  c = max(u, b)
  d = max { d : pat(c, d) = text(i + c, i + d) }
  если d < |pat|:
    i += d - u
    b = 1
  иначе:  # d = |pat|
    c = u - 1
    d = min { d >= b : pat(d, c) = text(i + d, i + c) }
    если d < b:
      верни i
    i += p
    b = |pat| - p + 1
```

Рис. 1.4: Алгоритм TW2

```
# вход: крит. pat = ab, |a| < p. p' - период b.
если a - суффикс b(1, p'):
  p = p'
  алгоритм TW1
иначе:
  q = max {|a|, |b|} + 1
  алгоритм TW3.
```

Рис. 1.5: Алгоритм TW3

```
# вход: text: str, pat: str, u: num, q: num
# ограничения: u < p, u критическая для pat
i = 1
пока i + |par| <= |text|:
  c = u
  d = max { d : pat(c, d) = text(i + c, i + d) }
  если d < |pat|:
    i += d - u
  иначе:  # d = |pat|
    c = u - 1
    d = min { d >= 0 : pat(d, c) = text(i + d, i + c) }
    если d = 0:
      верни i
    i += q
```

Доказательство. Известно, что $|a| < p$. Достаточно показать, что $|b| < p$. Это так в силу того, что иначе если $p'|p$, то p' – локальный период. Иначе найдется период еще меньше, чем p'. □

Займемся необходимой предобработкой. Обозначим через $<$ лексикографический порядок символов, а через $\lessdot$ – обратный к лексикографическому. Продолжим их на строки.

Утверждение 1.2.5 (Волшебное разложение)**.** *Пусть* $\texttt{pat} = ab = cd$ *где* b *и* d *– лексикографически максимальные по* $<$ *и* $\lessdot$ *суффиксы. Если* $|b| < |d|$, *то* $\texttt{pat} = ab$ *критичное. Иначе* $\texttt{pat} = cd$ *критичное.*

Доказательство. Если строка из одинаковых символов, то очевидно. Иначе не умаляя общности, пусть $|b| < |d|$.

Заметим, что локальный период разложение $\texttt{pat} = ab$ больше $|a|$. Ведь в противном случае можно прийти к противоречию с максимальностью b.

Осталось показать, что q – минимальный период для разложения $\texttt{pat} = ab$ будет также периодом $\texttt{pat}$. Используем такой факт: для любых строк x, y: $x < y$ и $x \lessdot y$, x будет префиксом y. Используем его для строк $\texttt{pat}(|c| + q, \cdot)$ и $\texttt{pat}(|c|, \cdot)$.

Очевидно, $\texttt{pat}(|c|+q, \cdot) \lessdot \texttt{pat}(|c|, \cdot)$ в силу выбора c. По порядку $<$ первые символы совпадают, а для остальных порядок, как между $\texttt{pat}(|a|, \cdot) = b$ и $\texttt{pat}(|a| + q, \cdot)$. Получили требуемое условие. □

Осталось найти максимальные суффиксы по заданным порядкам. Для этого используем модификацию алгоритма Дюваля (TW4). Идейно: читаем строку слева направа и для каждого префикса находим максимальный суффикс и его период.

Утверждение 1.2.6 (Дюваль)**.** *Пусть* $\texttt{pat}(i, k)$ *– максимальный суффикс строки* $\texttt{pat}(1, k)$.

1. *Если* $\texttt{pat}((k+1) - p) = \texttt{pat}(k+1)$, *то* $\texttt{pat}(i, k+1)$ *новый суффикс, его период* p.

2. *Если* $\texttt{pat}((k+1) - p) > \texttt{pat}(k+1)$, *то* $\texttt{pat}(i, k+1)$ *новый суффикс, его период* $k - i$ *(тривиальный).*

3. *Если* $\texttt{pat}((k+1) - p) < \texttt{pat}(k+1)$, *то максимальный суффикс не может начинаться в позициях* $[0, i + \{d : d|p, d > r\}]$.

Доказательство. Случай 1 очевиден. В остальных случаях от противного, и рассматриваем грани. Там можно вывести противоречие с максимальностью суффикса на предыдущем шаге. □

Для оценки сложности заметим, что значение $2i + j$ за k итераций увеличивается не меньше, чем на k.

Рис. 1.6: Алгоритм TW4 (Дюваля)

```
# вход: pat: str
p = <j - i>
i = 1, j = 2
пока j <= |pat|:
  d = max { j + d <= |pat| : pat(i, i + d) = pat(j, j + d) }
  если j + d > |pat|: выйди из цикла
  если pat(i + d) > pat(j + d): j += d + 1
  иначе: i += (d / p + 1), j = i + 1
верни pat(i, j - 1), p
```

Глава 2

Поиск нескольких подстрок в строке

2.1 Алгоритм Ахо-Корасик

Назовем множество строк D, которое будем называть словарем. Задача состоит в нахождении всех вхождений всех слов из D в строку `text`.

Задача решается в два этапа: составление вспомогательной структуры данных P и обработка строки `text`. Есть наивный алгоритм TR: в нем в качестве вспомогательной структуры берется бор, а вершины - структуры:

```
root : V = < корень бора >

v : V = {
  .repr = < строка на пути root - v >  # формально
  .term = < .repr является словом словаря D >
  .next = < (-): char -> V : .next(c).repr = .repr + c >
}
```

Сложность алгоритма TR будет зависить от реализации $(-).next$. Если использовать отображение или отсортированный массив пар, то время доступа $O(\log \sigma)$. Если lookup-таблицу или массив, то $O(1)$. Для удобства будем представлять вершины числами $\{1, 2, \dots\}$ и что $.next$ реализован lookup-таблицей.

Улучшим алгоритм TR: давайте добавим поле $v.link$, в котором будем указываем на вершину для которой $u.repr$ будет наидлиннейшим суффиксом из $v.repr$. $v.report$ будет указывать на вершину, которая при спуске по $v.link$ будет удовлетворять $.term = true$. Тогда получаем алгоритм Ахо-Корасик АНК

Утверждение 2.1.1. *Алгоритм* АНК *работает корректно и*

$$T(\text{АНК}) = O(|\texttt{text}| \log \sigma + |D|)$$

Доказательство. Каждая итерация первого цикла увеличивает $j = i - |v.repr| \leq i$. Тогда он отработает за $O(|\texttt{text}|\log\sigma)$. Второй цикл каждый раз вызывается для разных слов, так что он будет вызван $O(|D|)$ раз.

Корректность следует из того, что АНК – просто модицикация алгоритма TR. □

Осталось научиться находить поле $v.link$. Для проверки максимального символа мы можем отрезать по одному символу $v.repr$ и смотреть, в $u.repr$ какой вершины мы перешли. Получим алгоритм АНК1.

Утверждение 2.1.2. $T(\text{АНК1}) = O(\log\sigma\sum_{d\in D}|d|)$.

Доказательство. Это верно из того, что на пути $root- < v : v.repr = d >$ суммарное время работы для всех вершин будет $O(|d|)$:

$$\sum n_i \leq \sum |v_i.link.repr| - |v_d.link| + 1 = |v_d.link.repr| + |d|$$

□

Если в словаре только одно слово, то получаем алгорит Кнута-Морриса-Пратта. Тогда вместо дерева достаточно использовать массив значений. Вариант выше занимает $S(\text{АНК}) = O(|V|)$ памяти, а автоматный вариант $O(\sigma|V|)$ памяти.

Рис. 2.1: Алгоритм TR

```
v = root
для i = 1; i < |text| и v != nil; i += 1:
  если v.term: нашли слово v.repr в позиции i
  v = v.next(text(i))
```

Рис. 2.2: Алгоритм AHK

```
v = root
для i = 1; i < |text| и v != nil; i += 1:
  пока v != root и v.next(text(i)) = nil: v = v.link
  v = v.next(text(i))
  если v = nil:
    v = root
  для u = v; u != root; u = u.report:
    если u.term:
      нашли слово u.repr в позиции i - |u.repr|
```

Рис. 2.3: Алгоритм AHK1

```
qu(1) = root.link = root.report = root
k = 2;
для i = 1; i < k; i += 1:
  для v,c из { (v,c) qu(i).next(c) = v }:
    qu[k++] = v
    v.link = nil
    для p = qu[i]; p.report != nil и v.link = nil; p = p.link:
      v.link = p.link.next(c)
    если v.link = nil:
      v.link = root
    v.report = v.link
    пока !v.report.term:
      v.report = v.link.report
```

Глава 3

Строковые индексы

3.1 Суффиксный массив

По строке `text` необходимо построить структуру данных, с помощью которой можно находить вхождения строки `pat` в строку `text`.

Определение 3.1.1. *Суффиксный массив SA содержит перестановку чисел $\{1, 2, \dots, |\texttt{text}|\}$, причем $\texttt{text}(SA(i), \cdot) < \texttt{text}(SA(j), \cdot)$ для $i < j$.*

3.1.1 Алгоритм Карккайнена-Сандерса

Допустим, что алфавит представлен числами $\{1, 2, \dots, |\texttt{text}|\}$. Идейно можно бить строку на фрагменты длины $2, 4, 8, \dots$ и сортировать их слиянием: тогда сложность $T(n) = O(n) + T(\frac{n}{2}) = cn \sum_k \frac{1}{2^k} = O(n)$.

Идейно: рекурсивно отсортируем все суфииксы: сначала на позициях не кратных трем, затем оставшиеся и выполним слияние.

Пусть на каком-то этапе отсортированными оказываются строки t_1, t_2 (порязрядной сортировкой). И в строке t_1 все тройки меньше, чем в строке t_2. Пронумеруем тройки числами $\{1, \frac{2}{3}|\texttt{text}|\}$. Сформируем строку $t_1\$t_2$ с бесконечно малым символом \$. Рассмотрим суффиксный массив SA_{12} для этой строки. Его занесем в $ISA_{12}(SA(x)) = x$.

Для сортировки оставшихся троек: если первые символы различны, то порядки определяются непосредственно, иначе – по построенным порядкам. Тогда, получается, достаточно отсортировать пары $(\texttt{text}(3k), ISA_{12}(k))$.

Для слияния так же переиспользуем порядок, зафиксированный в ISA_{12}.

3.2 Суффиксное дерево

Определение 3.2.1. *Массив LCP, содержит на позиции i наибольший общего префикс строк $\texttt{text}(SA(i), \cdot)$ и $\texttt{text}(SA(i+1), \cdot)$*

Если построить структуру, вычисляющую $\min\{LCP(x) : i < x < j\}$ за $O(1)$, то можно построить наибольший общий префикс любой пары суффиксов за $O(1)$. LCP можно построить за $O(|\texttt{text}|)$ используя SA, ISA по алгоритму Касаи и др.

Определение 3.2.2. *Суффиксный бор – бор, в котором пути отмеченны всеми подстроками строки* `text`*. Терминальными вершинами считаются суффиксы.*

Определение 3.2.3. *Суффиксным деревом (сжатым суффиксным бором) назовем суффиксный бор без вершин с одним сыном. При удалении вершин соответствующие метки ребер склеиваются.*

Метками ребер оказываются подстроки. Будем хранить указатели на них. Тогда вершины являются структурами:

```
root : V = < корень бора >

v : V = {
  .repr = < строка на пути root - v >  # формально
  .term = < .repr является суффиксом text >
  .next = < по символу переходим в вершину,
            на ребре до которой данный символ первый>
  .par = < родитель >
  .beg, .end = < text(.beg, .end) = str(.par, v) >
}
```

Из определения полей $.beg$, $.end$, очевидно, у суффиксного дерева $|V| \le 2|\texttt{text}|$.

Суффиксное дерево можно построить по SA и LCP. Для этого надо вставлять суффиксы в порядке $(\texttt{text}(SA(1), \cdot), \texttt{text}(SA(2), \cdot), \ldots)$. При вставке очередного суффикса $\texttt{text}(SA(1), \cdot)$, при вставке поднимаемся снизу этого пути, создавая вершину на нужном месте.

3.2.1 Упрощенный алгорим Вейнера

Идейно: надо добавлять по одному суффиксу, двигаясь справа налево по строке `text`. На каждом шаге создаем новую вершину, и крепим ее к какой-то вершине или разбиваем ребро для ее прикрепления.

Определение 3.2.4. *Префиксной ссылкой назовем ссылку на вершину по символу, для которой на пути из корня до вершины строка получается дописыванием данного символа к текущей строке.*

Заметим, что префиксных ссылок $O(|\texttt{text}|)$, ведь в каждую вершину ведет не более одной ссылки. Для удобства еще оказывается удобным ввести фиктивный корень: он будет родителем корня, и длина строки на $fake{-}root$ равна 1.

Утверждение 3.2.1. *У каждой вершины на пути для нового суффикса есть вершина на пути для старого суффикса, для которой префиксная ссылка ведет в нее.*

Доказательство. Заметим, что путь для нового суффикса тот же, что и для старого, если удалить первый символ на пути.

Возможны две ситуации: если нижняя вершина на новом пути терминальная, то мы нашли требуюмею вершину: достаточно прикрепить к ней.

Иначе, в нижней вершине ветвление. Тогда пользуемся переносом одной ветви на другую. □

Теперь начнем вставлять суффикс.

Утверждение 3.2.2. *Если вершина существует или не существует в дереве, то есть вершина, которая по новому символу префиксной ссылкой переводит нас в новую вершину. Для вершин ниже таких ссылок не существует.*

Доказательство. Из предыдущего утверждения, если вершина, к которой будем крепиться, существует, то условие выполняется.

Если не существует, то если бы нарушалось второе условие, имело бы место противоречие: вершина не существует, а к ней прикреплена еще вершина. □

Если нужно разбивать ребро, то используя утверждение, спускаемся по одному символу, и в момент, когда символы различаются, создаем вершину и крепим к ней новый суффикс. Важно: при спуске вниз достаточно сверять только первый символ: по утверждению.

Оценим время работы алгоритма. Если мы при подъеме просматриваем k_i вершин, то совокупная сложность будет $O(\sum k_i \log \sigma)$. Если длина старой верви m, а новой m', то $k - 4 \le m - m'$. Тогда сложность

$$\sum k_i < \sum (m - m' + 4) = 4|\texttt{text}| + m_{|}\texttt{text}| \le O(|\texttt{text}|).$$

3.2.2 Алгоритм Укконена

Пусть в каждой вершине храняться все прежние поля, и новые:

```
root : V = < корень бора >

v : V = {
  .repr, .term, .next, .beg, .end

  .slink = < суффиксная ссылка >
}
```

Определение 3.2.5. *Суффиксная ссылка указывает на вершину, у которой строка на пути получается добавлением одного символа к пути к заданной вершине.*

Для корная там можно хранить вспомогательные значения.

Утверждение 3.2.3. *Суффиксная ссылка определена на любой некорневой вершине.*

Доказательство. Доказательство аналогично доказательству первого утверждения у алгоритма Вейнера. □

Мы будем последовательно добавлять префиксы строки `text`.

При добавлении нового префикса терминальное состояние меняется. Поэтому будем хранить только листья, корень, и вершины, у которых больше двух сыновей.

> Инвариант: на каждом шаге известен максимальный неуникальный суффикс. Для его хранения будем хранить вершину, и длину ребра в нее от родителя.

Утверждение 3.2.4. *Суффиксы максимального неуникального суффикса тоже неуникальны.*

Утверждение 3.2.5. *Суффиксное дерево будет совпадать с бором, построенным на строке, а число листьев в боре будет равно длине максимального неуникального суффикса.*

Доказательство. Суффиксы короче максимального неуникального не будут префиксами строки. Следовательно, в боре число листьев совпадает с длиной максимального неуникального суффикса.

Для неуникального суффикса длины больше максимального неуникального суффикса будет иметь самое левое вхождение, тогда он будет префиксом части строки, и попадет в бор. □

Теперь при добавлении нового префикса в дерево, добавится $m - m'$ новых листьев, где m, m' – старая и новая длина максимального суффикса.

Удобно полагать, что $.end = \infty$. Тогда нужно будет только вставлять новые листья, а дописывание символов на ребрах происходит автоматически. Так же необязательно хранить суффиксные ссылки.

Если не нужно обновлять максимальный суффикс, то достаточно добавить вершину, разбив ребро ниже от запомненной вершины. Иначе для просмотра всех кандидатов на максимальные суффиксы спускаемся по $.slink$ (в случае чего, спускаясь вниз по ребрам), и добавляем вершину по одному новому символу, то тех пор, как только переход будет по одному символу переход от строки.

Получили алгоритм UOK.

Оценим время работы алгоритма UOK. Пусть у нас есть указатели p_1, p_2, т.е. длина максимального неуникального суффикса и его длина + длина пути до w + 1. Каждая итерация цикла пока увелививает p_1 ровно на один,

Рис. 3.1: Алгоритм Укконена (UOK1)

```
# разбиение ребра
# вход: w, v, len, c
u = new_node {.beg = v.beg, .end=v.beg+len}
u.next(c) = new_node {.beg=i-1, .end=+inf}
u.next(text(v.beg+len)) = v
w.next(text(v.beg)) = u
v.beg += len  # => v.len -= len
верни u
```

Рис. 3.2: Алгоритм Укконена (UOK)

```
# добавление символа c
k++
prev = root
пока k > 0:
  для v = w.next(text(i-k)); v != null и k > v.len:
    w = v
    k -= w.len
    v = w.next(text(i-k))
  если v = null:
    w.next(text(i - k)) = new_node {.beg=i-k, .end=+inf}
    prev = prev.slink = w
  иначе если text(v.beg + k) != c:
    prev = prev.slink = <разбиение ребра>(w, v, k-1, c)
  иначе:
    prev.slink = w
    закончить
  если w != root: w = w.slink
  иначе k--
```

а p_2 на каждой итерации для увеличивается хотя бы на один. $p_1 \leq p_2$, следовательно, оба цикла использует не более $2n$ операций. Тогда итоговое время работы $O(n \log \sigma)$.

3.2.3 Возможные оптимизации

Можно использовать не указатели и new, а использовать статический массив достаточного размера. Вместо указателей тогда можно просто использовать индексы.

В алгоритме Вейнера стоит завести стек.

Для реализации $.next$, $.link$ стоит завести глобальную таблицу, которая будет по $V \times char$ определять значение.

3.3 Индексы

3.4 FM-индекс

Определение 3.4.1. *Преобразование Барроуза-Уилера по строке* **`text`** *образует строку BWT, такую, что*

$$BWT(i) = \begin{cases} \boldsymbol{\mathtt{text}}(SA(i)-1), & SA(i) = 0, \\ \boldsymbol{\mathtt{text}}(n), & SA(i) \neq 0. \end{cases}$$

Если в к строке добавить символ конца строки, то по построенному массиву можно восстановить исходный текст. Далее будем работать со строками с таким символом.

Определение 3.4.2. *Рангом $rank(a, i)$ назовем число символов a в строке BWT до позиции i невключительно.*

Тогда BWT можно закодировать массивом $runs_a(i) = \{.beg, .rank\}$, где $.beg$ – начало очередной серии, $.rank$ – длина этой серии. Его можно вычислить с помощью бинарного поиска. Дополнительно: $.len = (i+1).rank - (i).rank$.

Введем еще массив C на символах: $C(a) = \sum_{b<a} |j : \mathtt{text}(j) = b|$.

FM-индекс состоит из массивов $runs_a, C$. Будем предполагать, что $\sigma << r$, где r – общее число серий. Размер FM-индекса будет $O(r)$.

3.4.1 Алгоритм обратного поиска

Идейно: считывая строку справа налево, находим самый длинный интервал, все суффиксы из которого начинаются на прочитанную строку. Полученный алгоритм назовем алгоритмом обратного поиска (RS).

Утверждение 3.4.1. *Алгоритм RS корректен и работает за $O(n \log r)$.*

Доказательство. Мысленно можем продлить символы из BWT теми суффиксами, которые их продолжают. Тогда верно, что

$$l_i = C(a) + rank(a, l_{i+1}), \quad r_i - l_i = rank(a, r_{i+1}) - rank(a, l_{i+1}).$$

Отсюда следует корректность. Сложность очевидна. □

Введем в $rans_a(i)$ дополнительное поле

$$.sa = SA(runs_a(i).beg + runs_a(i).len - 1).$$

Также введем массивы $first, firstrun$, где будут запомнены первое вхождение первого элемента серии и перевод индексов первого вхождения в индекс предыдущей серии.

$$first : SA(runs_a(i).beg).$$

3.5 R-индекс

Он состоит из массивов $runs_a, first, firstrun$. Обратный поиск вычисляет последовательно интервалы, последний из которых содержит все вхождения исходной строки. В процессе обратного поиска будем поддерживать известным значение $SA(r_i)$.

Если $BWT(r_{i+1}) = a = \mathtt{text}(i)$, то $SA(r_i) = SA(r_{i+1}) - 1$. В противном случае, $SA(r_i) = runs_a(j).sa - 1$, где $runs_a(j)$ – последняя перед r_{i+1}.

Для нахождения $SA(k-1)$ по $SA(k)$ если известное значение на границе серий, можно бинарным поиском найти $firstrun(i)$, которое указывало бы на $runs_a(j)$. Иначе воспользуемся утверждением (идейно: добавляем символы, пока не попадем на границу серии).

Утверждение 3.5.1. *Пусть $k > 1$ и $BWT(k) = BWT(k-1)$. Тогда строки $\mathtt{text}(SA(k-1), \cdot)$ и $\mathtt{text}(SA(k), \cdot)$ расположенны в соседних позициях в SA.*

Доказательство. Очевидно от противного. □

Сложность нахождения k вхождений будет

$$O(|\mathtt{text}| \log |\mathtt{pat}| + k \log |\mathtt{text}|).$$

Зафиксируем последовательность из $r \log \frac{|\mathtt{text}|}{2} + O(r)$ битов. В нем можно сохранить r чисел из r битов. Посчитаем, сколько занимают поля используемых массивов. Результаты приведены в таблице.

Рис. 3.3: Алгоритм обратного поиска (RS)

```
l = 1, r = n
для i = n, n-1, ...:
  a = text(i)
  l = C(a) + rank(a, l)
  r = C(a) + rank(a, r)
```

Рис. 3.4: Нахождение $SA(r_i)$

```
# дано l_i, r_i, sa = SA(r_a)
для k = r_i..l_i:
  i = max < i : first(i) <= sa >
  h = sa - first(i)
  sa = runs(firstrun(i)).sa + h
```

Рис. 3.5: Используемая память

Поле	Биты
$runs_a(\cdot).beg$	$\sum r_a \log \frac{\lvert\texttt{text}\rvert}{r_a} + O(r)$
$runs_a(\cdot).rank$	$\leq \sum r_a \log \frac{\lvert\texttt{text}\rvert}{r_a} + O(r)$
$runs_a(\cdot).sa$	$r \log \lvert\texttt{text}\rvert$
$first(\cdot)$	$r \log \frac{\lvert\texttt{text}\rvert}{2} + O(r)$
$firstrun(\cdot)$	$r \log \lvert\texttt{text}\rvert$