MAXimal

home
algo
bookz
forum
about

добавлено: 11 Jun 2008 10:36
редактировано: 4 May 2012 20:32

Префикс-функция. Алгоритм Кнута-Морриса-Пратта

Содержание [скрыть]

Префикс-функция. Определение

Дана строка $s[0 \ldots n-1]$. Требуется вычислить для неё префикс-функцию, т.е. массив чисел $\pi[0 \ldots n-1]$, где $\pi[i]$ определяется следующим образом: это такая наибольшая длина наибольшего собственного суффикса подстроки $s[0 \ldots i]$, совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). В частности, значение $\pi[0]$ полагается равным нулю.

Математически определение префикс-функции можно записать следующим образом:

$$\pi[i] = \max_{k=0 \ldots i} \{ k : s[0 \ldots k-1] = s[i-k+1 \ldots i] \}.$$

Например, для строки "abcabcd" префикс-функция равна: $[0,0,0,1,2,3,0]$, что означает:

- у строки "a" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "ab" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abc" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abca" префикс длины 1 совпадает с суффиксом;
- у строки "abcab" префикс длины 2 совпадает с суффиксом;
- у строки "abcabc" префикс длины 3 совпадает с суффиксом;
- у строки "abcabcd" нет нетривиального префикса, совпадающего с суффиксом.

Другой пример — для строки "aabaaab" она равна: $[0,1,0,1,2,2,3]$.

Тривиальный алгоритм

Непосредственно следуя определению, можно написать такой алгоритм вычисления префикс-функции:

```
vector<int> prefix_function (string s) {
	int n = (int) s.length();
	vector<int> pi (n);
	for (int i=0; i<n; ++i)
		for (int k=0; k<=i; ++k)
			if (s.substr(0,k) == s.substr(i-k+1,k))
				pi[i] = k;
	return pi;
}
```

Как нетрудно заметить, работать он будет за $O(n^3)$, что слишком медленно.

Эффективный алгоритм

Этот алгоритм был разработан Кнутом (Knuth) и Праттом (Pratt) и независимо от них Моррисом (Morris) в 1977 г. (как основной элемент для алгоритма поиска подстроки в строке).

Первая оптимизация

Первое важное замечание — что значение $\pi[i+1]$ не более чем на единицу превосходит значение $\pi[i]$ для любого i.

Действительно, в противном случае, если бы $\pi[i+1] > \pi[i] + 1$, то рассмотрим этот суффикс, оканчивающийся в позиции $i+1$ и имеющий длину $\pi[i+1]$ — удалив из него последний символ, мы получим суффикс, оканчивающийся в позиции i и имеющий длину $\pi[i+1]-1$, что лучше $\pi[i]$, т.е. пришли к противоречию. Иллюстрация этого противоречия (в этом примере $\pi[i-1]$ должно быть равно 3):

$$\underbrace{\overbrace{s_0\ s_1}^{\pi[i-1]=2}\ s_2\ s_3}_{\pi[i]=4}\ \ldots\ \underbrace{s_{i-3}\ \overbrace{s_{i-2}\ s_{i-1}}^{\pi[i-1]=2}\ s_i}_{\pi[i]=4}$$

(на этой схеме верхние фигурные скобки обозначают две одинаковые подстроки длины 2, нижние фигурные скобки — две одинаковые подстроки длины 4)

Таким образом, при переходе к следующей позиции очередной элемент префикс-функции мог либо увеличиться на единицу, либо не измениться, либо уменьшиться на какую-либо величину. Уже этот факт позволяет нам снизить асимптотику до $O(n^2)$ — поскольку за один шаг значение могло вырасти максимум на единицу, то суммарно для всей строки могло произойти максимум n увеличений на единицу, и, как следствие (т.к. значение никогда не могло стать меньше нуля), максимум n уменьшений. В итоге получится $O(n)$ сравнений строк, т.е. мы уже достигли асимптотики $O(n^2)$.

Вторая оптимизация

Пойдём дальше — **избавимся от явных сравнений подстрок**. Для этого постараемся максимально использовать информацию, вычисленную на предыдущих шагах.

Итак, пусть мы вычислили значение префикс-функции $\pi[i]$ для некоторого i. Теперь, если $s[i+1] = s[\pi[i]]$, то мы можем с уверенностью сказать, что $\pi[i+1] = \pi[i] + 1$, это иллюстрирует схема:

$$\underbrace{\overbrace{s_0\ s_1\ s_2}^{\pi[i]}\ \overbrace{s_3}^{s_3=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}\ \ldots\ \underbrace{\overbrace{s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]}\ \overbrace{s_{i+1}}^{s_3=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}$$

(на этой схеме снова одинаковые фигурные скобки обозначают одинаковые подстроки)

Пусть теперь, наоборот, оказалось, что $s[i+1] \neq s[\pi[i]]$. Тогда нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) длине $j < \pi[i]$, что по-прежнему выполняется префикс-свойство в позиции i, т.е. $s[0 \ldots j-1] = s[i-j+1 \ldots i]$:

$$\overbrace{\underbrace{s_0\ s_1}_{j}\ s_2\ s_3}^{\pi[i]}\ \ldots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$

Действительно, когда мы найдём такую длину j, то нам будет снова достаточно сравнить символы $s[i+1]$ и $s[j]$ — если они совпадут, то можно утверждать, что $\pi[i+1] = j+1$. Иначе нам надо будет снова найти меньшее (следующее по величине) значение j, для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения j кончатся — это происходит, когда $j = 0$. В этом случае, если $s[i+1] = s[0]$, то $\pi[i+1] = 1$, иначе $\pi[i+1] = 0$.

Итак, общая схема алгоритма у нас уже есть, нерешённым остался только вопрос об эффективном нахождении таких длин j. Поставим этот вопрос формально: по текущей длине j и позиции i (для которых выполняется префикс-свойство, т.е. $s[0 \ldots j-1] = s[i-j+1 \ldots i]$) требуется найти наибольшее $k < j$, для которого по-прежнему выполняется префикс-свойство:

$$\overbrace{\underbrace{s_0 \ s_1}_k \ s_2 \ s_3}^j \ \ldots \ \overbrace{s_{i-3} \ s_{i-2} \ \underbrace{s_{i-1} \ s_i}_k}^j \ s_{i+1}$$

После столь подробного описания уже практически напрашивается, что это значение k есть не что иное, как значение префикс-функции $\pi[j-1]$, которое уже было вычислено нами ранее (вычитание единицы появляется из-за 0-индексации строк). Таким образом, находить эти длины k мы можем за $O(1)$ каждую.

Итоговый алгоритм

Итак, мы окончательно построили алгоритм, который не содержит явных сравнений строк и выполняет $O(n)$ действий.

Приведём здесь итоговую схему алгоритма:

- Считать значения префикс-функции $\pi[i]$ будем по очереди: от $i = 1$ к $i = n-1$ (значение $\pi[0]$ просто присвоим равным нулю).
- Для подсчёта текущего значения $\pi[i]$ мы заводим переменную j, обозначающую длину текущего рассматриваемого образца. Изначально $j = \pi[i-1]$.
- Тестируем образец длины j, для чего сравниваем символы $s[j]$ и $s[i]$. Если они совпадают — то полагаем $\pi[i] = j+1$ и переходим к следующему индексу $i+1$. Если же символы отличаются, то уменьшаем длину j, полагая её равной $\pi[j-1]$, и повторяем этот шаг алгоритма с начала.
- Если мы дошли до длины $j=0$ и так и не нашли совпадения, то останавливаем процесс перебора образцов и полагаем $\pi[i] = 0$ и переходим к следующему индексу $i+1$.

Реализация

Алгоритм в итоге получился удивительно простым и лаконичным:

```
vector<int> prefix_function (string s) {
	int n = (int) s.length();
	vector<int> pi (n);
	for (int i=1; i<n; ++i) {
		int j = pi[i-1];
		while (j > 0 && s[i] != s[j])
			j = pi[j-1];
		if (s[i] == s[j])  ++j;
		pi[i] = j;
	}
	return pi;
}
```

Как нетрудно заметить, этот алгоритм является **онлайновым** алгоритмом, т.е. он обрабатывает данные по ходу поступления — можно, например, считывать строку по одному символу и сразу обрабатывать этот символ, находя ответ для очередной позиции. Алгоритм требует хранения самой строки и предыдущих вычисленных значений префикс-функции, однако, как нетрудно заметить, если нам заранее известно максимальное значение, которое может принимать префикс-функция на всей строке, то достаточно будет хранить лишь на единицу большее количество первых символов строки и значений префикс-функции.

Применения

Поиск подстроки в строке. Алгоритм Кнута-Морриса-Пратта

Эта задача является классическим применением префикс-функции (и, собственно, она и была открыта в связи с этим).

Дан текст t и строка s, требуется найти и вывести позиции всех вхождений строки s в текст t.

Обозначим для удобства через n длину строки s, а через m — длину текста t.

Образуем строку $s + \# + t$, где символ $\#$ — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Теперь рассмотрим её значения, кроме первых $n+1$ (которые, как видно, относятся к строке s и разделителю). По определению, значение $\pi[i]$ показывает наидлиннейшую длину подстроки, оканчивающейся в позиции i и совпадающего с префиксом. Но в нашем случае это $\pi[i]$ — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i. Больше, чем n, эта длина быть не может — за счёт разделителя. А вот равенство $\pi[i] = n$ (там, где оно достигается), означает, что в позиции i оканчивается искомое вхождение строки s (только не надо забывать, что все позиции отсчитываются в склеенной строке $s + \# + t$).

Таким образом, если в какой-то позиции i оказалось $\pi[i] = n$, то в позиции $i - (n+1) - n + 1 = i - 2n$ строки t начинается очередное вхождение строки s в строку t.

Как уже упоминалось при описании алгоритма вычисления префикс-функции, если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. В нашем случае это означает, что нужно хранить в памяти лишь строку $s + \#$ и значение префикс-функции на ней, а потом уже считывать по одному символу строку t и пересчитывать текущее значение префикс-функции.

Итак, алгоритм Кнута-Морриса-Пратта решает эту задачу за $O(n+m)$ времени и $O(n)$ памяти.

Подсчёт числа вхождений каждого префикса

Здесь мы рассмотрим сразу две задачи. Дана строка s длины n. В первом варианте требуется для каждого префикса $s[0\ldots i]$ посчитать, сколько раз он встречается в самой же строке s. Во втором варианте задачи дана другая строка t, и требуется для каждого префикса $s[0\ldots i]$ посчитать, сколько раз он встречается в t.

Решим сначала первую задачу. Рассмотрим в какой-либо позиции i значение префикс-функции в ней $\pi[i]$. По определению, оно означает, что в позиции i оканчивается вхождение префикса строки s длины $\pi[i]$, и никакой больший префикс оканчиваться в позиции i не может. В то же время, в позиции i могло оканчиваться и вхождение префиксов меньших длин (и, очевидно, совсем не обязательно длины $\pi[i] - 1$). Однако, как нетрудно заметить, мы пришли к тому же вопросу, на который мы уже отвечали при рассмотрении алгоритма вычисления префикс-функции: по данной длине j надо сказать, какой наидлиннейший её собственный суффикс совпадает с её префиксом. Мы уже выяснили, что ответом на этот вопрос будет $\pi[j-1]$. Но тогда и в этой задаче, если в позиции i оканчивается вхождение подстроки длины $\pi[i]$, совпадающей с префиксом, то в i также оканчивается вхождение подстроки длины $\pi[\pi[i]-1]$, совпадающей с префиксом, а для неё применимы те же рассуждения, поэтому в i также оканчивается и вхождение длины $\pi[\pi[\pi[i]-1]-1]$ и так далее (пока индекс не станет нулевым). Таким образом, для вычисления ответа мы должны выполнить такой цикл:

```
vector<int> ans (n+1);
for (int i=0; i<n; ++i)
        ++ans[pi[i]];
for (int i=n-1; i>0; --i)
        ans[pi[i-1]] += ans[i];
```

Здесь мы для каждого значения префикс-функции сначала посчитали, сколько раз он встречался в массиве $\pi[]$, а затем посчитали такую в некотором роде динамику: если мы знаем, что префикс длины i встречался ровно $\mathrm{ans}[i]$ раз, то именно такое количество надо прибавить к числу вхождений его длиннейшего собственного суффикса, совпадающего с его префиксом; затем уже из этого суффикса (конечно, меньшей чем i длины) выполнится "пробрасывание" этого количества к своему суффиксу, и т.д.

Теперь рассмотрим вторую задачу. Применим стандартный приём: припишем к строке s строку t через разделитель, т.е. получим строку $s + \# + t$, и посчитаем для неё префикс-функцию. Единственное отличие от первой задачи будет в том, что учитывать надо только те значения префикс-функции, которые относятся к строке t, т.е. все $\pi[i]$ для $i \geq n+1$.

Количество различных подстрок в строке

Дана строка s длины n. Требуется посчитать количество её различных подстрок.

Будем решать эту задачу итеративно. А именно, научимся, зная текущее количество различных подстрок, пересчитывать это количество при добавлении в конец одного символа.

Итак, пусть k — текущее количество различных подстрок строки s, и мы добавляем в конец символ c. Очевидно, в результате могли появиться некоторые новые подстроки, оканчивавшиеся на этом новом символе c. А именно, добавляются в качестве новых те подстроки, оканчивающиеся на символе c и не встречавшиеся ранее.

Возьмём строку $t = s + c$ и инвертируем её (запишем символы в обратном порядке). Наша задача — посчитать, сколько у строки t таких префиксов, которые не встречаются в ней более нигде. Но если мы посчитаем для строки t префикс-функцию и найдём её максимальное значение $\pi_{\max}$, то, очевидно, в строке t встречается (не в начале) её префикс длины $\pi_{\max}$, но не большей длины. Понятно, префиксы меньшей длины уж точно встречаются в ней.

Итак, мы получили, что число новых подстрок, появляющихся при дописывании символа c, равно $s.\mathrm{length}() + 1 - \pi_{\max}$.

Таким образом, для каждого дописываемого символа мы за $O(n)$ можем пересчитать количество различных подстрок строки. Следовательно, за $O(n^2)$ мы можем найти количество различных подстрок для любой заданной строки.

Стоит заметить, что совершенно аналогично можно пересчитывать количество различных подстрок и при дописывании символа в начало, а также при удалении символа с конца или с начала.

Сжатие строки

Дана строка s длины n. Требуется найти самое короткое её "сжатое" представление, т.е. найти такую строку t наименьшей длины, что s можно представить в виде конкатенации одной или нескольких копий t.

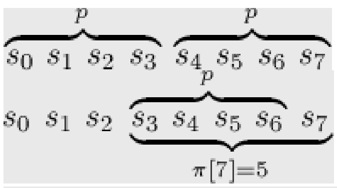
Понятно, что проблема является в нахождении длины искомой строки t. Зная длину, ответом на задачу будет, например, префикс строки s этой длины.

Посчитаем по строке s префикс-функцию. Рассмотрим её последнее значение, т.е. $\pi[n-1]$, и введём обозначение $k = n - \pi[n-1]$. Покажем, что если n делится на k, то это k и будет длиной ответа, иначе эффективного сжатия не существует, и ответ равен n.

Действительно, пусть n делится на k. Тогда строку можно представить в виде нескольких блоков длины k, причём, по определению префикс-функции, префикс длины $n-k$ будет совпадать с её суффиксом. Но тогда последний блок должен будет совпадать с предпоследним, предпоследний - с предпредпоследним, и т.д. В итоге получится, что все блоки блоки совпадают, и такое k действительно подходит под ответ.

Покажем, что этот ответ оптимален. Действительно, в противном случае, если бы нашлось меньшее k, то и префикс-функция на конце была бы больше, чем $n-k$, т.е. пришли к противоречию.

Пусть теперь n не делится на k. Покажем, что отсюда следует, что длина ответа равна n. Докажем от противного — предположим, что ответ существует, и имеет длину p (p делитель n). Заметим, что префикс-функция необходимо должна быть больше $n-p$, т.е. этот суффикс должен частично накрывать первый блок. Теперь рассмотрим второй блок строки; т.к. префикс совпадает с суффиксом, и и префикс, и суффикс покрывают этот блок, и их смещение друг относительно друга k не делит длину блока p (а иначе бы k делило n), то все символы блока совпадают. Но тогда строка состоит из одного и того же символа, отсюда $k = 1$, и ответ должен существовать, т.е. так мы придём к противоречию.



$$s_4 = s_3, \quad s_5 = s_4, \quad s_6 = s_5, \quad s_7 = s_6 \quad \Longrightarrow \quad s_0 = s_1 = s_2 = s_3$$

Построение автомата по префикс-функции

Вернёмся к уже неоднократно использованному приёму конкатенации двух строк через разделитель, т.е. для данных строк s и t вычисление префикс-функции для строки $s + \# + t$. Очевидно, что т.к. символ $\#$ является разделителем, то значение префикс-функции никогда не превысит $s.\mathrm{length}()$. Отсюда следует, что, как упоминалось при описании алгоритма вычисления префикс-функции, достаточно хранить только строку $s + \#$ и значения префикс-функции для неё, а для всех последующих символов префикс-функцию вычислять на лету:

$$\underbrace{s_0\ s_1\ \dots\ s_{n-1}\ \#}_{\text{need to save}}\underbrace{t_0\ t_1\ \dots\ t_{m-1}}_{\text{need not to save}}$$

Действительно, в такой ситуации, зная очередной символ $c \in t$ и значение префикс-функции в предыдущей позиции, можно будет вычислить новое значение префикс-функции, никак при этом не используя все предыдущие символы строки t и значения префикс-функции в них.

Другими словами, мы можем построить **автомат**: состоянием в нём будет текущее значение префикс-функции, переходы из одного состояния в другое будут осуществляться под действием символа:

$$s_0\ s_1\ \dots\ s_{n-1}\ \#\ \underbrace{\dots}_{\pi[i-1]} \Longrightarrow s_0\ s_1\ \dots\ s_{n-1}\ \#\ \underbrace{\dots}_{\pi[i-1]} + t_i \Longrightarrow s_0\ s_1\ \dots\ s_{n-1}\ \# \dots \underbrace{t_i}_{\pi[i]}$$

Таким образом, даже ещё не имея строки t, мы можем предварительно построить такую таблицу переходов $(\mathrm{old\_}\pi, c) \rightarrow \mathrm{new\_}\pi$ с помощью того же алгоритма вычисления префикс-функции:

```
string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

s += '#';
int n = (int) s.length();
vector<int> pi = prefix_function (s);
vector < vector<int> > aut (n, vector<int> (alphabet));
for (int i=0; i<n; ++i)
        for (char c=0; c<alphabet; ++c) {
                int j = i;
                while (j > 0 && c != s[j])
                        j = pi[j-1];
                if (c == s[j])  ++j;
                aut[i][c] = j;
        }
```

Правда, в таком виде алгоритм будет работать за $O(n^2 k)$ (k — мощность алфавита). Но заметим, что вместо внутреннего цикла while, который постепенно укорачивает ответ, мы можем воспользоваться уже вычисленной частью таблицы: переходя от значения j к значению $\pi[j-1]$, мы фактически говорим, что переход из состояния (j, c) приведёт в то же состояние, что и переход $(\pi[j-1], c)$, а для него ответ уже точно посчитан (т.к. $\pi[j-1] < j$):

```
string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

s += '#';
int n = (int) s.length();
vector<int> pi = prefix_function (s);
vector < vector<int> > aut (n, vector<int> (alphabet));
for (int i=0; i<n; ++i)
        for (char c=0; c<alphabet; ++c)
                if (i > 0 && c != s[i])
                        aut[i][c] = aut[pi[i-1]][c];
                else
                        aut[i][c] = i + (c == s[i]);
```

В итоге получилась крайне простая реализация построения автомата, работающая за $O(nk)$.

Когда может быть полезен такой автомат? Для начала вспомним, что мы считаем префикс-функцию для строки $s + \# + t$, и её значения обычно используют с единственной целью: найти все вхождения строки s в строку t.

Поэтому самая очевидная польза от построения такого автомата — **ускорение вычисления префикс-функции** для строки $s + \# + t$. Построив по строке $s + \#$ автомат, нам уже больше не нужна ни строка s, ни значения префикс-функции в ней, не нужны и никакие вычисления — все переходы (т.е. то, как будет меняться префикс-функция) уже предпосчитаны в таблице.

Но есть и второе, менее очевидное применение. Это случай, когда строка t **является гигантской строкой, построенной по какому-либо правилу**. Это может быть, например, строка Грея или строка, образованная рекурсивной комбинацией нескольких коротких строк, поданных на вход.

Пусть для определённости мы решаем **такую задачу**: дан номер $k \le 10^5$ строки Грея, и дана строка s длины $n \le 10^5$. Требуется посчитать количество вхождений строки s в k-ю строку Грея. Напомним, строки Грея определяются таким образом:

$$g_1 = "a"$$
$$g_2 = "aba"$$
$$g_3 = "abacaba"$$
$$g_4 = "abacabadabacaba"$$
$$\ldots$$

В таких случаях даже просто построение строки t будет невозможным из-за её астрономической длины (например, k-ая строка Грея имеет длину $2^k - 1$). Тем не менее, мы сможем посчитать значение префикс-функции на конце этой строки, зная значение префикс-функции, которое было перед началом этой строки.

Итак, помимо самого автомата также посчитаем такие величины: $G[i][j]$ — значение автомата, достигаемое после "скармливания" ему строки g_i, если до этого автомат находился в состоянии j. Вторая величина — $K[i][j]$ — количество вхождений строки s в строку g_i, если до "скармливания" этой строки g_i автомат находился в состоянии j. Фактически, $K[i][j]$ — это количество раз, которое автомат принимал значение $s.\mathrm{length}()$ за время "скармливания" строки g_i. Понятно, что ответом на задачу будет величина $K[k][0]$.

Как считать эти величины? Во-первых, базовыми значениями являются $G[0][j] = j$, $K[0][j] = 0$. А все последующие значения можно вычислять по предыдущим значениям и используя автомат. Итак, для вычисления этих значений для некоторого i мы вспоминаем, что строка g_i состоит из g_{i-1} плюс i-ый символ алфавита плюс снова g_{i-1}. Тогда после "скармливания" первого куска (g_{i-1}) автомат перейдёт в состояние $G[i-1][j]$, затем после "скармливания" символа char_i он перейдёт в состояние:

$$\mathrm{mid} = \mathrm{aut}[\ G[i-1][j]\ ][\mathrm{char}_i]$$

После этого автомату "скармливается" последний кусок, т.е. g_{i-1}:

$$G[i][j] = G[i-1][\mathrm{mid}]$$

Количества $K[i][j]$ легко считаются как сумма количеств по трём кускам g_i: строка g_{i-1}, символ char_i, и снова строка g_{i-1}:

$$K[i][j] = K[i-1][j] + (\mathrm{mid} == s.\mathrm{length}()) + K[i-1][mid]$$

Итак, мы решили задачу для строк Грея, аналогично можно решить целый класс таких задач. Например, точно таким же методом решается **следующая задача**: дана строка s, и образцы t_i, каждый из которых задаётся следующим образом: это строка из обычных символов, среди которых могут встречаться рекурсивные вставки других строк в форме $t_k[\mathrm{cnt}]$, которая означает, что в это место должно быть вставлено cnt экземпляров строки t_k. Пример такой схемы:

$$t_1 = "abdeca"$$
$$t_2 = "abc" + t_1[30] + "abd"$$
$$t_3 = t_2[50] + t_1[100]$$
$$t_4 = t_2[10] + t_3[100]$$

Гарантируется, что это описание не содержит в себе циклических зависимостей. Ограничения таковы, что если явным образом раскрывать рекурсию и находить строки t_i, то их длины могут достигать порядка 100^{100}.

Требуется найти количество вхождений строки s в каждую из строк t_i.

Задача решается так же, построением автомата префикс-функции, затем надо вычислять и добавлять в него переходы по целым строкам t_i. В общем-то, это просто более общий случай по сравнению с задачей о строках Грея.

Задачи в online judges

Список задач, которые можно решить, используя префикс-функцию:

- UVA #455 **"Periodic Strings"** [сложность: средняя]
- UVA #11022 **"String Factoring"** [сложность: средняя]

- UVA #11452 **"Dancing the Cheeky-Cheeky"** [сложность: средняя]
- SGU #284 **"Grammar"** [сложность: высокая]