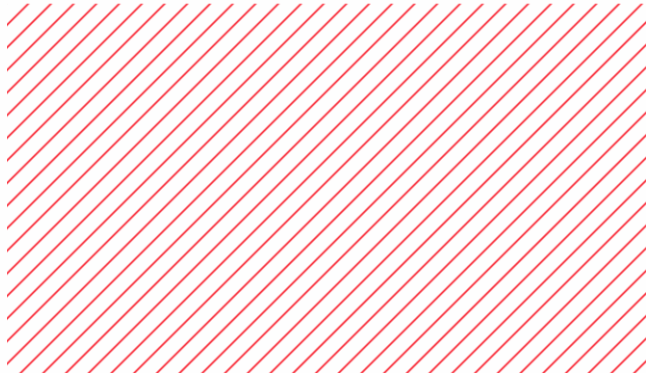
академия больших данных

mail.ru group

Поиск строк.

Мацкевич Степан

Алгоритмы и структуры данных



План

- **Поиск подстроки в строке**
  - Префикс-функция
  - Z-функция
- **Бор**
- **Алгоритм Ахо-Корасик**

Строки

Алфавит $A = \{a_0, a_1, \dots, a_{K-1}\}$

Строка $S = a_{i_0} a_{i_1} \dots a_{i_{n-1}}$, n – длина

Подстрока = совпадающая подпоследовательность

Суффикс – подстрока конца строки

Префикс – подстрока начала строки

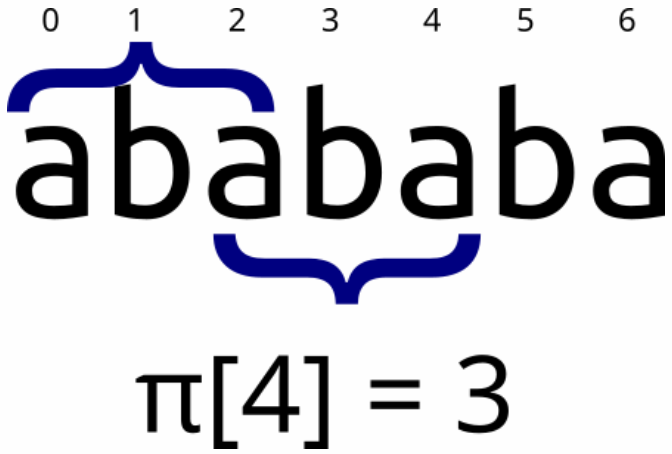
Собственный суффикс S – суффикс, не совпадающий с S.

preFIX

Префикс-функция

Префикс-функция

Префикс-функция (π-функция) – длина самого длинного собственного суффикса S[0..i], являющегося префиксом S[0..i].



Префикс-функци

Еще пример:

S	a	b	c	d	a	b	c	c	a	b	c	d	a	b	a
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
π[i]	0	0	0	0	1	2	3	0	1	2	3	4	5	6	1

Вычисление префикс-функции

Вычисление «в лоб» – $O(n^3)$.

Можно вычислить за $O(n)$.

Если $S[i] == S\left[\pi[i-1]\right]$, то

$$\pi[i] = \pi[i-1] + 1$$

Если $S[i] == S\left[\pi[\pi[i-1]-1]\right]$, то

$$\pi[i] = \pi[\pi[i-1]-1] + 1$$

И так далее до 0.

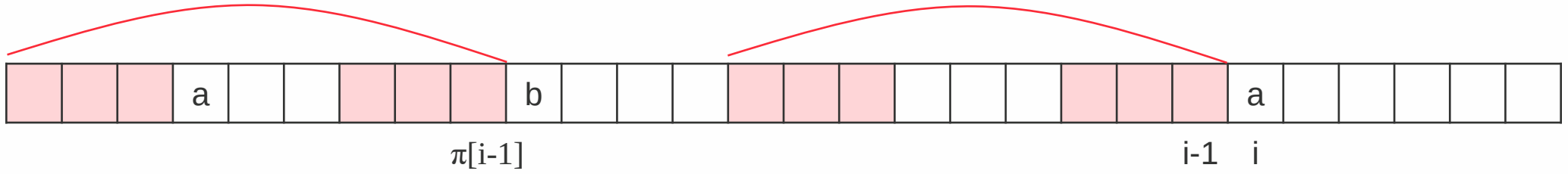
Вычисление префикс-функции

По сути перебираем все возможные суффиксы S[0..i-1], являющиеся префиксами.

И ищем среди них самый длинный, который можно продлить символом S[i].

Перебираем их в порядке уменьшения длины:

$\pi[i-1], \pi[\pi[i-1]-1], \pi\left[\pi\left[\pi[i-1]\right]-1\right], \ldots$



Алгоритм Кнута-Морриса-Пратта

Поиск вхождений подстроки. Конкатенируем шаблон, сентилен и текст.

Вычисляем π[].
Ищем где π[i] = length(pattern).

Время работы T = O(|pattern| + |text|).

Алгоритм Кнута-Морриса-Пратта

Память M = O(|pattern|).

Для достижения такого объема памяти не храним конкатенированную строку и не храним значения префикс-функции для всей строки pattern$text.
Вычисление префикс-функции для индексов i > |pattern| остается без изменений:
Также храним предыдущее значение префикс-функции.
Значения префикс-функции для индексов больше или равных |pattern| не потребуется, так как

$$|\pi| \leq |pattern|.$$

Z-функция

Z-функция

Z(s, i) - длиннейший префикс подстроки, начинающийся с позиции i в s, который также является префиксом s. Такой отрезок будем называть **отрезком совпадения.**

Пример:

a	b	c	d	a	b	c	c	a	b	c	d	a	b	a
15	0	0	0	3	0	0	0	6	0	0	0	2	0	1

В 0 позиции отрезком совпадения является вся строка,
а в 8 - подстрока “abcdab”.

Вычисление z-функции

Храним индексы [l,r] позиции самого правого отрезка совпадения.
Изначально l = r = 0. Пусть для всех k из {1, 2, … , i - 1} z(k) подсчитана. Разберём теперь два случая:

1. i > r. В этом случае просто будем проверять совпадение букв s[z[i]] и s[i + z[i]], положив изначально z[i] = 0 и увеличивать z[i]. Т.к. мы получили новый отрезок совпадения (в случае z[i] > 0) обновим l = i, r = i + z[i] - 1
2. i <= r. В этом случае мы можем воспользоваться уже подсчитанными значениями, а именно z[l-i]. Однако, оно может оказаться слишком большим. Поэтому положим z[i] = min(z[l-i], r-i+1) и далее проверим бОльшие значения z[i] тривиальным алгоритмом из п.1

Алгоритм Кнута-Морриса-Пратта (2)

Поиск вхождений подстроки. Конкатенируем шаблон, сентилен и текст.

Вычисляем z[].
Ищем где z[i] = length(pattern).

Время работы T = O(|pattern| + |text|).

Алгоритм Кнута-Морриса-Пратта (2)

Память M = O(|pattern|).

Для достижения такого объема памяти не храним конкатенированную строку и не храним значения z-функции для всей строки pattern$text.
Вычисление z-функции для индексов i > |pattern| остается без изменений:
Также храним текущие значения параметров l и r.
Значения z-функции для индексов больше или равных |pattern| не потребуется, так как $|r - l| \leq |pattern|$.

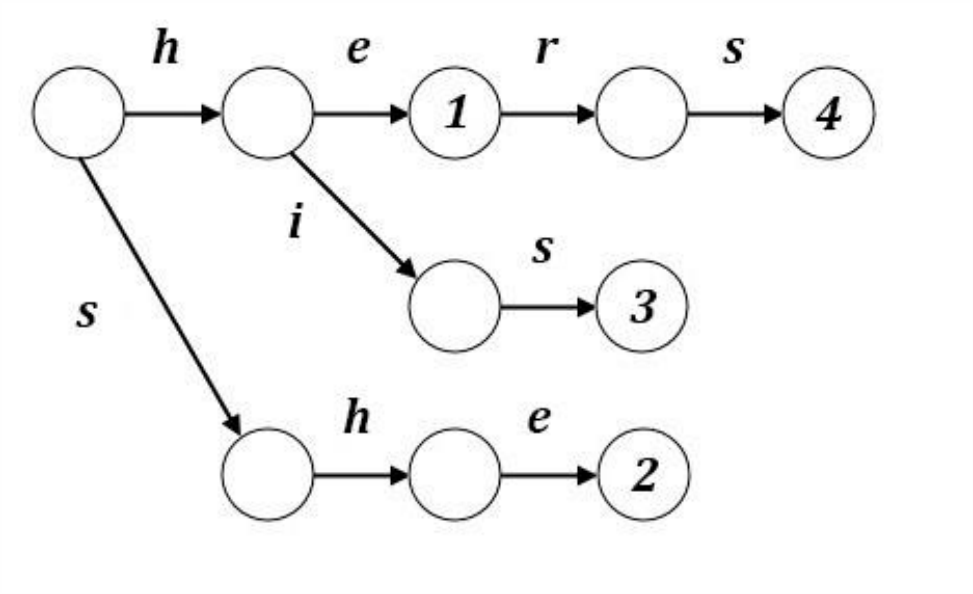
Бор

Бор

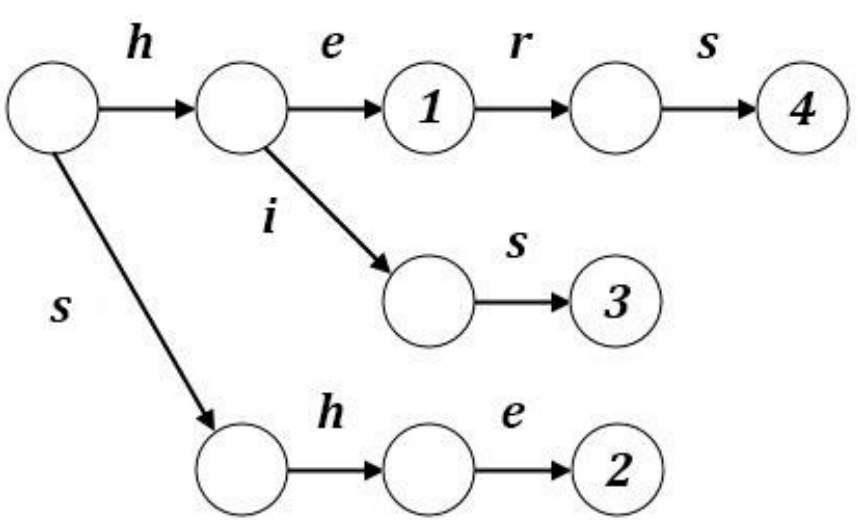
Бор – дерево слов.

Англ. – **trie (трай)**.
Trie = tree + retreival

На картинке бор для множества
{“hers”, “his”, “she”, “he”}



Как хранить переходы?



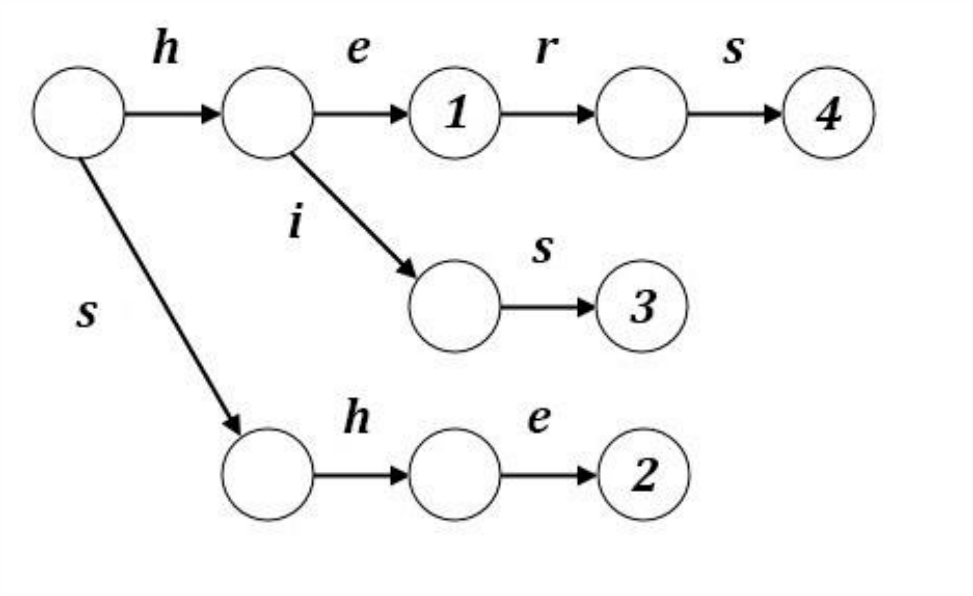
K – размер алфавита. Варианты:

1) unordered_map<char, Node*>
   Переход стоит O(1) в среднем.
2) map<char, Node*>
   Переход стоит O(log(K)).
   Но можно перебирать по порядку.
3) vector<Node*> или
   array<256, Node*>, если символ однобайтный
   Переход стоит O(1).
   Расходуется память.

Построение бора

Добавляем каждое слово от корня.
По буквам переходим к следующему узлу.
Если перехода нет – строим его.
Если слово закончилось – помечаем вершину как терминальную.

Строится за O(n), n – суммарная длина слов в множестве.



Поиск строки в боре

Задача. Есть множество строк $\{p_i\}$и строка s длины n.
Проверить, есть ли строка s в множестве $\{p_i\}$.

Хеш-таблица? O(n) в среднем.

Бор! Построенный по строкам p_i.

Спускаемся в боре по символа строки s.

O(n) гарантированно.

Алгоритм Ахо-Корасик

Алгоритм Ахо-Корасик (Aho, Corasick, 1975)

Задача:
$P = \{ P1, P2, \dots , Pk \}$ - шаблоны. $n = \sum |Pi|$.

T - текст. $m = |T|$.
Найти все появления шаблонов из P в тексте T.

“Простое” решение: $O(n + k * m)$ последовательным применением k линейных алгоритмов.

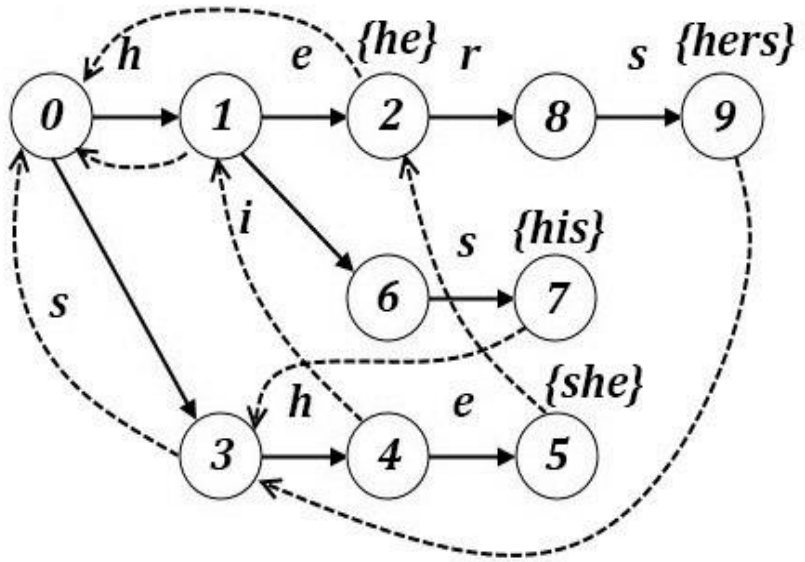
Ахо-Корасик: $O(n + m + z)$, z - количество появлений шаблонов.

Бор с суффиксными ссылками

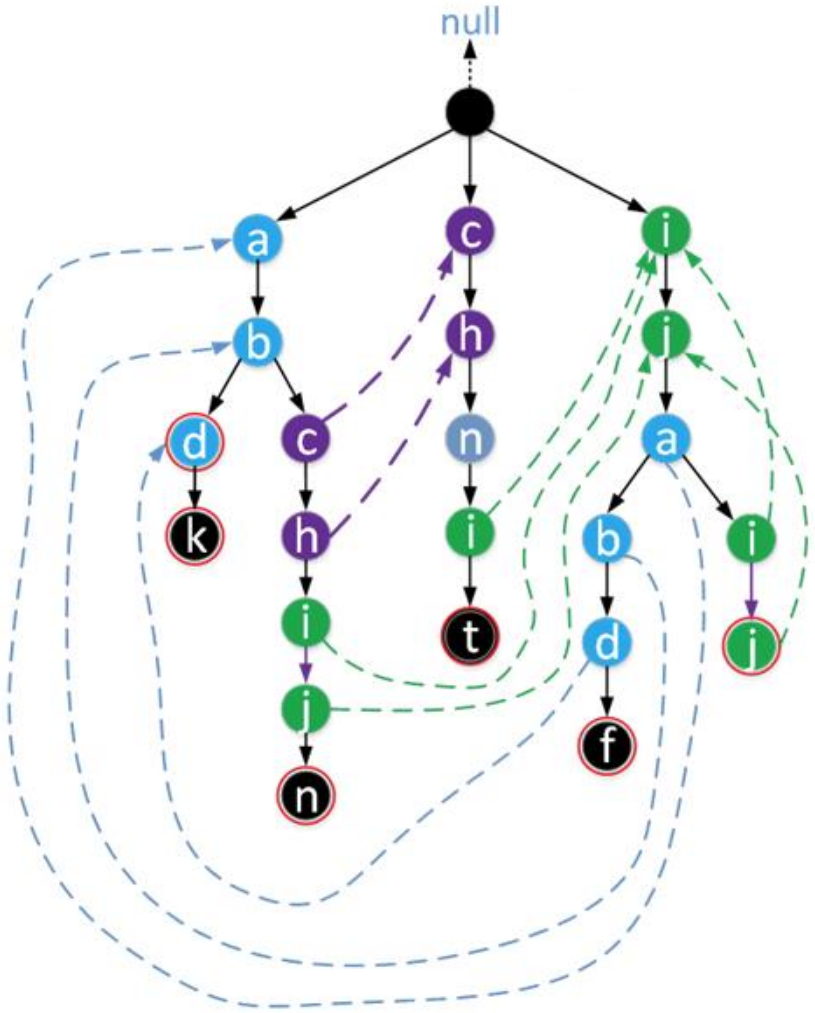
Суффиксная ссылка f(s) указывает на самый длинный суффикс из s, являющийся префиксом в боре.

Пример:

Максимальный суффикс из “she”, являющийся префиксом бора - “he”.



Еще один пример бора с суффиксными ссылками



Алгоритм Ахо-Корасик

1. Строим бор шаблонов с суффиксными ссылками.

2. Обрабатываем символы текста по одному.
   Начальное положение – корень бора.

   Основная идея – сразу вывести все шаблоны, оканчивающиеся в обработанном символе текста.
   Все такие шаблоны являются суффиксами друг другу.

Обработка одного символа текста

Обработав символ, хотим находиться в боре в самой длинном префиксе шаблона, являющейся подстрокой текста и оканчивающейся в обработанном символе.

Обработаем очередной символ a. Находимся в узле v

1) Если из v есть переход по a, то переходим по нему.
2) Если нет, то переходим в наибольший суффикс, являющийся префиксом шаблона – там может быть переход по a. Этот переход = переход по суффиксной ссылке.
   Перебираем f(v), f(f(v)),..., пока не найдем переход по a, либо не упремся в корень.
3) Перейдем по a.
4) Выведем в ответ все шаблоны, являющиеся суффиксом текущей вершины.

Удобно сделать из корня переходы по всем отсутствующим символам в корень же.

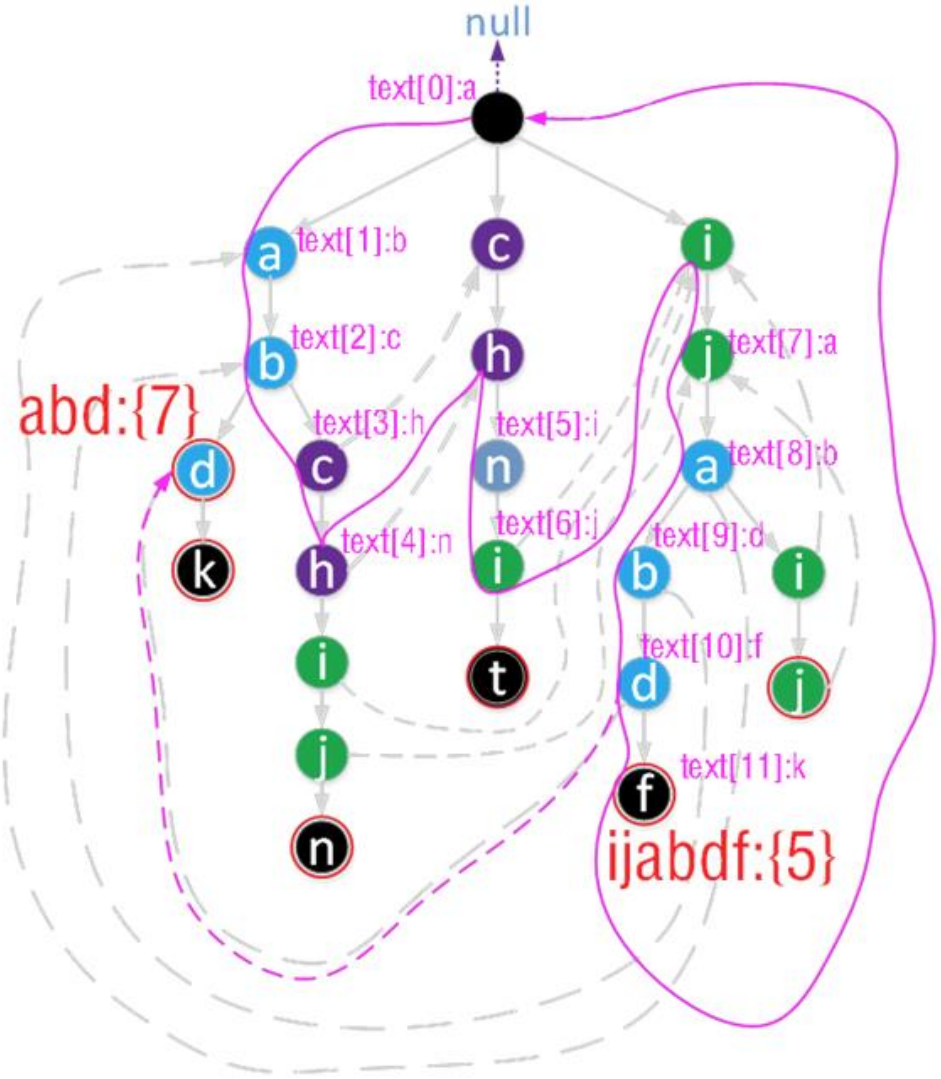
Пример работы алгоритма

Строки-шаблоны:

- abd
- abdk
- abdchijn
- chnit
- ijabdf
- ijaij

Текст:

abchnijabdfk



Алгоритм Ахо-Корасик

Состояние – узлы бора. Начальное состояние - корень s.

Обозначим:

Функция перехода: g(s,a) - возвращает состояние, в которое можно перейти из состояния s по символу a.

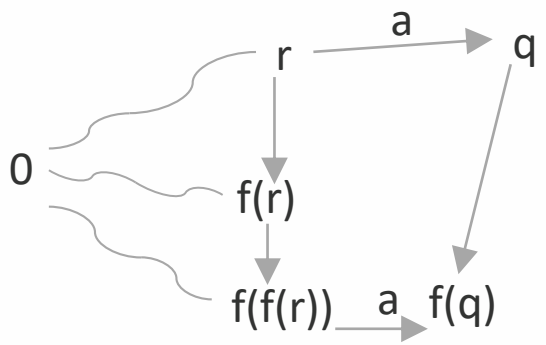
Функция неудачи: f(s) - возвращает состояние, в которое нужно перейти при просмотре неподходящего символа – суффиксная ссылка.

Выходная ф-я: Out(s) - множество шаблонов, которые обнаруживаются при переходе в s.

Псевдокод поиска подстрок А-К

```
q = 0; // Начинаем из корня
for( i = 0; i < m; i++ ) {
    while( q != 0 && g( q, T[i] ) == 0 ) {
        q = f( q );
    }
    q = g( q, T[i] ); // Выполняем переход, если возможно.
    if( out( q ) != 0 )
        print( out( q ) );
}
```

Вычисление суффиксных ссылок



Надо найти самый длинный суффикс родителя, который может быть продолжен по символу, входящему в v.

Если v – не корень, то

`f(v) = goto( f( parent(v) ), in(v) )`,

где parent(v) – родительская вершина v, in(v) – метка на ребре бора, ведущем в v.

Построение бора А-К с суффиксными ссылками. Вариант 1

1. Строим бор, в конце слова out(v) = { Pi }. Корень зацикливаем по оставшимся символам.
2. Считаем f(q) обходом в ширину. out(q) = out(q) U out(f(q))
3. Убираем зацикленность корня.

Время работы может быть больше O(n).

Автомат Ахо-Корасик.

Состояние - вершина бора.
Начальное (стартовое) состояние - корень (0).

Функция перехода: goto(v, a) - возвращает состояние, соответствующее переходу из состояния s по символу a.

Если в исходном боре из v есть переход по символу a в вершину u, то goto(v, a) = u.
Если в исходном боре из v нет перехода по символу a,
то проходим по суффиксной ссылке и ищем переход по символу a уже от f(v):
goto(v, a) = goto(f(v), a)

Построение автомата А-К. Вариант 2.
Ленивое построение для задачи поиска всех вхождений шаблонов.

1. Строим бор, в конце слова out(v) = { P_i }. Корень зацикливаем по оставшимся символам.
2. Выполняем переходы по символам текста goto(v, T[j]):

Если есть ребро в боре от текущего состояния, переходим по нему – goto(v, a) совпадает с этим узлом.

Если ребра в боре нет, то используем либо закэшированное значение goto(v, a), либо вычисляем его и сохраняем в кэше.

Перебираем все длинные суффиксные ссылки вместо out(v).

Поиск подстрок с помощью автомата Ахо-Корасик.

```
// Сам алгоритм.
void AhoKorasik({P}, T) {
    BuildTrie();
    int q = 0; // Стартовая.
    for( int i = 0; i < m; i++ ) {
        q = goto( q, T[i] );
        print( out( q ) );
    }
}
```

Поиск подстрок с помощью автомата Ахо-Корасик.

```
// Функция перехода.
int goto( q, a ) {
    if( g[q, a] != -1 ) // В исходном боре есть переход или уже вычисляли
        return g[q, a];
    if( q == 0 ) // Корень
        return 0;
    return g[q, a] = goto( f( q ), a );
}

// Суффиксная ссылка.
int f( q ) {
    if( f_cache[q] != -1 )
        return f_cache[q];
    if( q == 0 || parent( q ) == 0 ) // Корень или родитель - корень
        return 0;
    return f_cache[q] = goto( f( parent( q ) ), in( q ) );
}
```

Поиск подстрок с помощью автомата Ахо-Корасик.

```
// Функция ответа.
std::vector<int> out( int q ) {
    std::vector<int> result;
    if( !IsTerminal( q ) )
        q = TermLink( q );
    for( ; q != 0; q = TermLink( q ) )
        // Пока не дошли до корня по длинным суффиксным ссылкам
        result.emplace_back( q );
    return result;
}
```

Оценка времени работы

Обработка одного символа текста складывается из:

1. Одного вызова goto.
   Ленивое построение переходов для каждого узла и символа строит переход по a не более одного раза.
   Ленивое построение суффиксной ссылки выполняется не более одного раза для каждого узла
2. Проходу по терминальным ссылкам.
   Ленивое построение терминальных ссылок – по 1 вызову на каждую вершину бора
   Каждый переход по терминальной ссылке к корню – находит один ответ или упирается в корень. Количество ответов – k.

Общее время – O(n + m + k)

Спасибо за внимание!