МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 8304 _____ Масалыкин Д.Р.
Преподаватель Размочаева Н.В.
_____

Санкт-Петербург
2020

Цель работы

Изучить алгоритм Ахо-Корасик для нахождения набора образцов в строке, а также имплементировать его на языке C++.

Задание

1. Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:
Первая строка содержит текст T ($1 \le |T| \le 100000$)
Вторая — число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P = \{p1, \ldots, pn\}$ $1 \le |pi| \le 75$
Все строки содержат символы из алфавита $\{A, C, G, T, N\}$ Выход:

Все вхождения образцов из P в T.
Каждая вхождение образца в текст представить в виде двух чисел — i p
Где i — позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1)
Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample input	Sample output
NTAG 3 TAGT TAG T	2 2 2 3

2. Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с *джокером*.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который «совпадает» с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения P в текст T.

Например образец ab??c? с джокером ? встречается дважды в тексте *xabvccbababcax*.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределенной длины. В шаблон входит хотя бы один символ не джокер, т. е.

шаблоны вида ??? недопустимы. Все строки содержат символы из алфавита $\{A, C, G, T, N\}$.

Вход:

Текст (T, $1 \le |T| \le 100000$)
Шаблон (P, $1 \le |P| \le 40$)
Символ джокера

Выход:
Строки с номерами позиций вхождения шаблона (каждая строка содержит только один номер).
Номера должны выводиться в порядке возрастания.

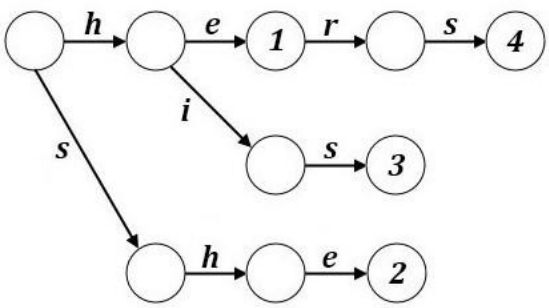
Sample input	Sample output
ACTANCA A$$A$ $	1

Индивидуальное задание, вариант 3

Вычислить длину из самой длинной цепочки суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

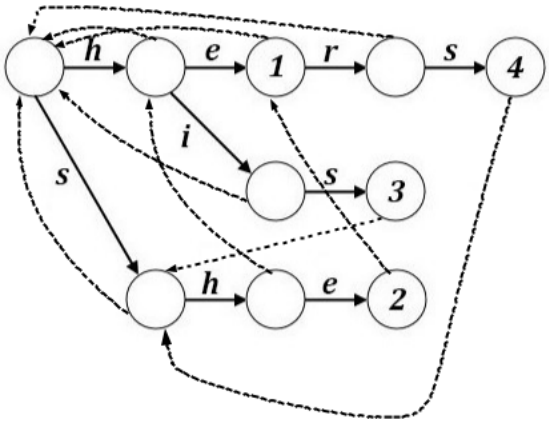
Структура данных бор (Trie))

Префиксное дерево (*бор*, *луч*, *нагруженное дерево*, англ. *trie*) — структура данных, позволяющая хранить ассоциативный массив, ключами которого являются строки. Представляет из себя подвешенное дерево с символами на ребрах. Строки получаются последовательной записью всех символов хранящихся на ребрах от корня до терминальной вершины. Размер бора линейно зависит от суммы длин всех строк. Поиск в бору занимает время, пропорциональное длине образца.



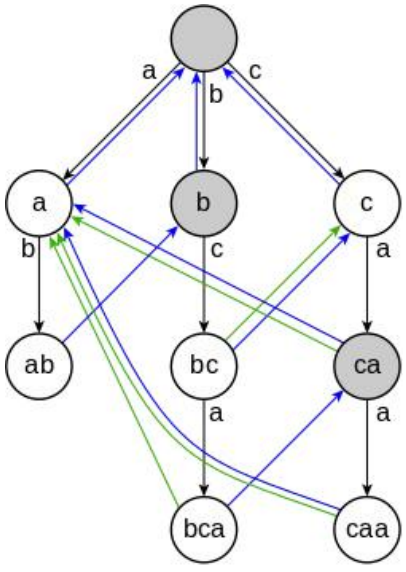
Суффиксные (suffix-link) и конечные (dictionary-link) ссылки

Для использования бора в алгоритме Ахо-Корасик, необходимо преобразовать его в конечный детерминированный автомат. Для этого введем понятие *суффиксной ссылки* (англ. *suffix-link*). Назовем суффиксной ссылкой вершины v указатель на вершину u, такую, что строка u — наибольший собственный суффикс строки v, или если такой вершины нет в боре, то указатель на корень. В частности, ссылка из корень ведет в самого себя. Понадобятся суффиксные ссылки для каждой вершины в боре.



Помимо суффиксных ссылок, в алгоритме Ахо-Корасик могут также использоваться *конечные ссылки* (англ. *dictionary-links*). Конечная ссылка — ближайшая по суффиксным ссылкам конечная вершина.

пример автомата с конечными ссылками.



Белые вершины - конечные. Синие стрелки — суффиксные ссылки. Зеленые стрелки — конечные ссылки.

Описание алгоритмов

Поиск набора образцов из P в строке T

Инициализация:

1. Строится бор из образцов из P.
2. С помощью поиска в ширину находятся все суффиксные ссылки каждой вершины.
    1. Если вершина корень, то ее суффиксная ссылка указывает на саму себя.
    2. Если вершина смежна с корнем, то ее суффиксная ссылка указывает на корень
    3. Иначе нужно брать суффиксную ссылку родителя, если у узла по этой ссылке есть наследник с таким же символом, то суффиксная ссылка указывает на этого наследника. Иначе продолжаем спускаться по суффиксным ссылкам, и проверять их наследников, пока не дойдем до корня.

Основной ход:

1. Пусть i — индекс текущего символа текста. Начальное состояние автомата — корень.
2. Совершается переход в автомате.
    1. Если в текущем состоянии есть прямой наследник с символом $T[i]$, то перейти в это состояние и увеличить i на единицу.
    2. Иначе, если текущее состояние корень, то увеличить i на единицу без изменения состояния.
    3. Иначе перейти по суффиксной ссылке состояния, и уже для этого состояния повторить шаги 1-3 перехода в автомате.
3. Если текущее состояние не корень, то нужно проверить его и весь путь суффиксных ссылок до корня.
    1. Если текущее состояние, является конечным, то образец найден.
    2. Перейти по суффиксной ссылке, если по ссылке не корень, то повторить с предыдущего шага.

Поиск одного образца P с джокером в тексте T.

Инициализация:

1. Найдем все подстроки p_i в P, которые не содержат джокеров
2. Для каждого p_i запомним, где он находится в образце P. Если p_i встречается в P несколько раз, то для такой подстроки запоминаем все места вхождения в P.
3. Для всех p_i из P составляется конечный детерминированный автомат по бору, как описано в предыдущем алгоритме.
4. Пусть количество подстрок разделенных джокерами равно k (учитываются одинаковые подстроки).

Основной ход:

1. Создается массив A длиной $|T|$, в котором будут фиксироваться нахождения подстрок образца.
2. Поиск подстрок образца осуществляется аналогично поиску в предыдущем алгоритме, за исключением того, что каждый раз, когда подстрока образца длиной l найдена в тексте на позиции j, значение массива A на позиции $j - l + 1$, если она больше 0, увеличивается на единицу.
3. После того, как все вхождения подстрок образца найдены в T. Для каждой позиции i в массиве A, если $A[i]$ равен k и $i + |P| \leq |T|$, то шаблон P встречается в позиции i.

Описание классов и их методов

Класс TrieNode:

Класс узла бора. Содержит поля: *unsigned int id* для хранения универсального номера вершины; *map<char, TrieNode*> children* ассоциативный массив наследников, каждый из который связан с родителем символом перехода; *TrieNode* parent* — указатель на родительский узел; *char toParent* — символ перехода от родителя узла к самому узлу; *bool bEnd* — флаг, который говорит,

является ли узел терминальным; *string _str* — строка, для которой узел является терминальной вершиной.

Методы класса:

`Конструктор (sizesizet_t id, TrieNode *parent = nullptr, char toParent = 0))` — На вход принимается целое неотрицательное число *id*, указатель на объект *TrieNode parent* и символ *toParent*. Создает узел с номером *id*, родителем которого является *parent* а символ перехода от родителя — *toParent*.

`bool end(size)t const` — возвращает 1, если узел является терминальным и 0 в ином случае.

`const string& getString(size)t const` — получить строку, если, вершина является терминальной. Если вершина не является терминальной, возвращает пустую строку.

`size_t id(size)t const` — возвращает целое неотрицательное число — номер узла.

Класс ACSearch:

Класс предназначеный для применения алгоритма Ахо-Корасик. Содержит поля: *TrieNode* root* — корень бора; *map<TrieNode*, TrieNode*> suffix* — ассоциативный массив суффиксных ссылок.

Методы класса:

`Конструткор(sizeboolt log_on = 1)` — На вход принимает булеву переменную *log_on*. Создает объект класса, корень инициализируется *nullptr*. Если *log_on* равен 1, то во время вызова методов класса, действия будут записываться в файл лога.

`private void calcSuffixLink(sizeTrieNode*t node)` — принимает на вход указатель на узел бора. Записывает в *suffix*[*node*] узел, на который указывает суффиксная ссылка.

`bool empty(size)t const` — возвращает 1, если бор класса пуст, иначе возвращает 0.

`void insert(sizeconst string key)` — на вход получает строку *key*. Добавляет в бор строку *key*.

`bool remove(sizeconst string key)` — на вход получает строку *key*. Удаляет строку *key* из бора, если там таковая имеется. Возвращает 1, если строка удалена, 0 в ином случае.

`bool isIn(sizeconst string key)` — на вход получает строку *key*. Возвращает 1, если *key* находится в боре и 0 в ином случае.

`vector<TrieNode*> getAllNodes(size)t` — возвращает массив всех узлов бора.

`void turnIntoMachine(size)t` - функция которая должна вызываться, после того, как все образцы добавлены в объект класса. Создает суффиксные ссылки в боре.

`bool isMachine(size)t const` — возвращает 1, если бор превращен в автомат и 0 в ином случае.

`void print_longest_links(size)t` — выводит в стандартный поток вывода наибольшую цепочку суффиксных ссылок и наибольшую цепочку конечных ссылок.

`map<size_t, vector<string>> search(sizeconst string& text)` — получает на входу строку *text*. Возвращает ассоциативный массив массивов, где каждому индексу *size_t* текста соответствует набо образцов *vector*, начинающихся с этой позиции.

Класс WildcradSearch:

Класс аналогичный классу *ACSearch*, за исключением того, что класс имеет следующие уникальные поля: *size_t pattern_size* — длина шаблона; map<string, vector<size_t>> subP_map — таблица, хранящая позиции вхождения *vector* подстроки string в шаблон; subP_count — количество подстрок шаблона, не содержащих джокеров.

Отличающиеся от *ACSearch* методы класса:

`void setPattern(sizeconst string& key, char joker)` — на вход получает строку key и символ joker. Разбивает образец key на подстроки, которые разделяет joker. После чего, по полученным подстрокам собирается автомат.

`vector<size_t> search(sizeconst string& text)` — функция поиска образца. На вход получает строку *text*, на выходе массив целых неотрицательных чисел. Находит сообщенный раннее в объект класса образец в *text* и возвращает массив индексов, где этот образец встречается.

Сложность алгоритма

Найдем сложность построения бора, вершины проходятся или создаются от корня до терминальной каждый раз, когда добавляется образец, соответственно сложность O(n), где n сумма длин всех образцов.

Найдем сложность получения суффиксных ссылок бора, для каждой вершины, нужно проверить всех наследников от суффиксной ссылки, то есть O(n log(k)), k — размер алфавита (логарифм, поскольку используется ассоциативный массив, поиск в котором осуществляется за логарифм).

Поиск осуществляется за время O(L log(k) + t), где L — размер текста, а t — количество вхождений, поскольку для каждого перехода выбирается наследник текущего состояния, а сам алгоритм перебирает все вхождения для текущего индекса.

Итого имеем ***сложность по времени O((n+L)log(k) + n + t)***, где n — суммарная длина всех образцов, L — размер текста, t — количество вхождений образцов в текст, а k — размер алфавита (если учесть, что алфавит зачастую константная величина, то ***сложность по времени O(n+L+t)***).

За счет использования ассоциативного массива, бор не хранит пустых ячеек памяти, соответственно занимаемая память пропорциональна числу ребер и соответственно числу узлов, то есть ***сложность по памяти O(n)***

Хранение решения

Автомат хранится, как связное дерево, где каждый узел (*TrieNode*), хранит ссылки (*TrieNode**), на наследников. Все вхождения запоминаются в ассоциативном массиве map<size_t, vector<string>>, где каждому индексу вхождения соответствует массив образцов.

Использованные оптимизации

Наследники каждого узла хранятся в ассоциативном массиве map стандартной библиотеке, что уменьшает сложность по памяти.

Тестирование

ACSsearch:

input	aabcddcbcabbbacbcaa 6 a ab bc bca c caa
output	1 1 2 1 2 2 3 3 4 5 7 5 8 3 8 4 9 5 10) 1 10) 2 14 1 15 5 16 3 16 4 17 5 17 6 18 1 19 1 longest suffix-link path size is 3 first suffix-link path with this size: 5 -[bca]- 7 -(size_tca)- 1 -[a]- 0) longest dictionary-link path size is 1 first dictionary-link path with this size: 4: bc 6: c
input	takeso fasofast fassofatake sosso sofastake so 4 fast sofa so take
output	1 4 5 3

10) 2
10) 3

```
12 1
20  2
20  3
24 4
29 3
32 3
35 2
35 3
37 1
40  4
45 3
longest suffix-link path size is 2
first suffix-link path with this size:
3 -(size_tfas)- 5 -(size_ts)- 0)
there is no dictionary links
```

input

```
NTAG

3
TAGT
TAG
T
```

output

```
2 2
2 3
longest suffix-link path size is 2
first suffix-link path with this size:
4 -[TAGT]- 1 -[T]- 0)
longest dictionary-link path size is 1
first dictionary-link path with this size:
4: TAGT
1: T
```

input

```
we have not to too big text

4
have
to
too
super_huge_needle_qwertyqwerty
```

output

```
4 1
13 2
16 2
16 3
longest suffix-link path size is 2
first suffix-link path with this size:
14 -(size_tsuper_h)- 1 -(size_th)- 0)
there is no dictionary links
```

input

```
How much wood would a woodchuck chuck if a woodchuck

could chuck wood?
4
wood
woo
would
ould
```

output

```
10)  1
```

10) 2
15 3
16 4
23 1

```
23 2
44 1
44 2
55 4
66 1
66 2
longest suffix-link path size is 2
first suffix-link path with this size:
2 -(size_two)- 8 -(size_to)- 0)
longest dictionary-link path size is 1
first dictionary-link path with this size:
7: would
11: ould
```

WildcradSearch:

input	ACTANCA A$$A$ $
output	1
input	abasdabvcdfbasdabzxcabdcdcbsdabasdabdcdcbzxcabsssabscdsb ab@@@ab@cd@b@@@ @
output	1 16 30)
input	ABCfdBCgaCfABCoioiABCsdBCcsCsABCasBCdfCasAB ABC##BC##C#ABC #
output	1 19
input	qwerqwerqwer qwer@@ @
output	1 5
input	aaaaaaabbaaaaaaa a%%%%a %
output	1 2 5 6 7 10) 11

Вывод

В ходе выполнения лабораторной работы был изучен алгоритм Ахо-Корасик для поиска набора образцов в строке, а также вариант алгоритма для поиска образца с джокерами. Для работы алгоритма был реализован бор, а также функция превращения его в детерминированный автомат с помощью введения суффиксных ссылок. Алгоритмы поиска набора образцов и поиска образца с джокерами имплементированы на языке C++.

ПРИЛОЖЕНИЕ А

Ахо-Корасик без джокера

```
#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <fstream>
#include <queue>
#include <algorithm>

class TrieNode;

class ACSearch
{
   TrieNode *root;
   size_t size;
   std::map<TrieNode*, TrieNode*> suffix;

   std::ofstream log;
   bool logger;

   void calcSuffixLink(TrieNode* node);

public:
   ACSearch(bool log_on = 1);
   ACSearch();

   bool empty() const;

   void insert(const std::string& key);

   bool remove(const std::string &key);

   bool isIn(const std::string& key) const;

   // returns empty vector, if trie is empty
   std::vector<TrieNode*> getAllNodes();

   // sets suffix links;
   void turnIntoMachine();

   // is suffix links map set
   bool isMachine() const;

   void print();

   void print_longest_links();

   std::map<size_t, std::vector<std::string>>
   search(const std::string& text);
};


class TrieNode
{
   friend class ACSearch;
   friend class WildcardSearch;

   size_t _id;
   std::map<char, TrieNode*> children;
   TrieNode *parent;
   char toParent;
   bool bEnd;
   std::string _str;

   TrieNode(size_t id, TrieNode* parent = nullptr, char toParent = 0):
         _id(id), parent(parent), toParent(toParent), bEnd(0)
   {};

public:
   bool end() const
   {
      return bEnd;
   }

   const std::string& getString() const
   {
      return _str;
   }

   size_t id() const
   {
      return _id;
   }
};


ACSearch::ACSearch(bool log_on):
      root(nullptr), size(0), logger(log_on)
{
   if(log_on) log.open("log", std::ofstream::out | std::ofstream::trunc);
}


ACSearch::ACSearch():
      root(nullptr), size(0), logger(1)
{
   log.open("log", std::ofstream::out | std::ofstream::trunc);
}


void ACSearch::calcSuffixLink(TrieNode* node)
{
   if(logger)
   {
      log << "suffix-link calculator initiated for node: " << node->id() << std::endl;
   }

   // suffix link of root is root itself
   if(node == root)
   {
      if(logger) log << "suffix[root] = root" << std::endl;
```

```
        suffix[node] = root;
        return;
    }

    // suffix links of root children are root
    if(node->parent == root)
    {
        if(logger) log << "this is a child of root. suffix = root" << std::endl;
        suffix[node] =  root;
        return;
    }

    // for computate the suffix link
    // we need the suffix link of node parent
    TrieNode *pCurrBetterNode;
    try
    {
        pCurrBetterNode = suffix.at(node->parent);
    }
    catch (std::out_of_range)
    {
        // trying to calculate prefix, before parent
        throw std::out_of_range("can't take first parent suffix, "
                                "check <calcSuffixLink>, "
                                "it's actually broken");
    }

    // and the character, which that moves us to node
    char cParentChar = node->toParent;

    if(logger) log << "parent suffix " << pCurrBetterNode->id();
    if(logger) log << "jump through suffix link" << std::endl;
    while(true)
    {
        // try to find the children of parent suffix
        // with the same key as to current node
        if(pCurrBetterNode->children.find(cParentChar) != pCurrBetterNode->children.end())
        {
            if(logger) log << "child found: " << pCurrBetterNode->children[cParentChar]->id()
                           << " id. Set suffix link on it" << std::endl;
            suffix[node] = pCurrBetterNode->children[cParentChar];
            break;
        }

        // otherwise jump through suffix links
        // until reach root
        if(pCurrBetterNode == root)
        {
            if(logger) log << "suffix path led to root, suffix link set on root" << std::endl;
            suffix[node] = root;
            break;
        }

        try
        {
            pCurrBetterNode = suffix.at(pCurrBetterNode);
            if(logger) log << "Next suffix. Current node is "
                           << pCurrBetterNode->id() << std::endl;
        }
        catch(std::out_of_range)
        {
            // trying to calculate prefix of node, which doesn't
            // have full prefix path
            throw std::out_of_range("can't take next parent suffix, "
                                    "check <calcSuffixLink>, "
                                    "it's actually broken");
        }
    }
}


bool ACSearch::empty() const
{
    return !root;
}


void ACSearch::insert(const std::string& key)
{
    if(logger) log << "add key: " << key << std::endl;
    if(empty()){
        if(logger) log << "trie is empty, root created" << std::endl;
        root = new TrieNode(0);
        size = 1;
    }

    auto pCrawl = root;
    for(auto c: key)
    {
        if(pCrawl->children.count(c) == 0)
        {
            if(logger) log << "create children with character: " << c << std::endl;
            pCrawl->children[c] = new TrieNode(size, pCrawl, c);
            size++;
        }
        else
        {
            if(logger) log << "already have a child with that character, go deeper" << std::endl;
        }
        pCrawl = pCrawl->children[c];
    }
    pCrawl->bEnd = 1;
    pCrawl->_str = key;

    if(logger) log << std::endl;
}


// works bad with nodes IDs
// there is no log outputs, because i don't use this function
bool ACSearch::remove(const std::string &key)
{
    if(empty()) return 0;
    TrieNode *pTail = root;

    for(auto c: key)
```

```
    {
        if(pTail->children.count(c))
        {
            pTail = pTail->children[c];
        }
        else
        {
            return 0;
        }
    }

    while(pTail)
    {
        auto pParent = pTail->parent;
        if(pTail->children.size() == 1)
        {
            if(pParent)
                pParent->children.erase(pTail->toParent);
            delete pTail;
            pTail = pParent;
        }
        else return 1;
    }
    root = nullptr;
    return 1;
}


// there is no log outputs, because i don't use this function
bool ACSearch::isIn(const std::string& key) const
{
    if(empty()) return 0;
    auto pCrawl = root;
    for(const auto c: key)
    {
        if(pCrawl->children.count(c) == 0)
        {
            return false;
        }

        try
        {
            pCrawl = pCrawl->children.at(c);
        }
        catch(std::out_of_range)
        {
            // what the hell, it's not possible
            throw std::out_of_range("can't accsess element, but the key "
                                    "is in, check <isIn> function, "
                                    "it's actually broken");
        }
    }
    return(pCrawl->bEnd);
}



// returns empty vector, if trie is empty
// there is no log outputs, because it's just a BFS
std::vector<TrieNode*> ACSearch::getAllNodes()
{
    std::vector<TrieNode*> nodes;
    if(root == nullptr)
        return nodes;

    // find all graph nodes using breadth first search
    std::queue<TrieNode*> q;
    q.push(root);

    while(!q.empty())
    {
        auto pCurrNode = q.front();
        q.pop();
        nodes.push_back(pCurrNode);

        for(auto child: pCurrNode->children)
        {
            // it is a tree, there can't be cycles,
            // so, there is no need to mark nodes
            q.push(child.second);
        }
    }
    return nodes;
}


void ACSearch::turnIntoMachine()
{
    if(logger) log << "Machinizer initiated" << std::endl;
    if(!root) return;

    suffix.clear();

    // find suffix links using breadth first search
    // which needs queue
    std::queue<TrieNode*> q;

    q.push(root);
    while(!q.empty())
    {
        auto pCurrNode = q.front();
        q.pop();

        if(logger) log << "built suffix link for " << pCurrNode->id() << " node. " << std::endl;
        calcSuffixLink(pCurrNode);
        if(logger) log << "Suffix link leads to " << suffix.at(pCurrNode)->id() << " node"
                       << std::endl << std::endl;

        for(auto child: pCurrNode->children){
            q.push(child.second);
        }
    }
}

bool ACSearch::isMachine() const
```

```
{
    return(!suffix.empty());
}

void ACSearch::print()
{
    auto nodes = getAllNodes();
    for(auto node: nodes)
    {
        std::cout << node->parent << " " << node << " "
                << node->toParent << " " << suffix[node] << std::endl;
    }
}


void ACSearch::print_longest_links()
{
    auto nodes = getAllNodes();

    std::vector<TrieNode*> max_suffix_path;
    std::vector<TrieNode*> max_dictionary_path;

    // check all nodes to find which have longest path
    for(auto node: nodes)
    {
        auto pSuffixCrawl = node;
        std::vector<TrieNode*> current_suffix_path;
        std::vector<TrieNode*> current_dictionary_path;

        while(pSuffixCrawl != root)
        {
            current_suffix_path.push_back(pSuffixCrawl);
            if(pSuffixCrawl->end()) current_dictionary_path.push_back(pSuffixCrawl);

            TrieNode* pNextNode;
            try
            {
                pNextNode = suffix.at(pSuffixCrawl);
            }
            catch(std::out_of_range)
            {
                throw std::out_of_range("Can't access suffix link of vertex, "
                                        "error caused by <print_longest_links> function. "
                                        "Probably, the trie wasn't machinized");
            }

            if(pNextNode->end() && current_dictionary_path.empty())
            {
                current_dictionary_path.push_back(pSuffixCrawl);
            }
            pSuffixCrawl = pNextNode;
        }

        if(current_suffix_path.size() > max_suffix_path.size())
            max_suffix_path = current_suffix_path;
        if(current_dictionary_path.size() > max_dictionary_path.size())
            max_dictionary_path = current_dictionary_path;
    }

    std::cout << "longest suffix-link path size is " << max_suffix_path.size() << std::endl;
    std::cout << "first suffix-link path with this size: " << std::endl;
    for(auto node: max_suffix_path)
    {
        std::cout << node->id() << " -";
        if(node->end()) std::cout << "[" << node->getString() << "]";
        else
        {
            std::cout << "(";
            auto pTrieCrawl = node;
            while(pTrieCrawl != root)
            {
                std::cout << pTrieCrawl->toParent;
                pTrieCrawl = pTrieCrawl->parent;
            }
            std::cout << ")";
        }
        std::cout << "- ";
    }
    std::cout << root->id() << std::endl;

    if(max_dictionary_path.empty())
    {
        std::cout << "there is no dictionary links" << std::endl;
        return;
    }

    std::cout << "longest dictionary-link path size is " << max_dictionary_path.size() << std::endl;
    std::cout << "first dictionary-link path with this size: " << std::endl;
    for(auto node: max_dictionary_path)
    {
        std::cout << node->id() << ": ";
        if(node->end()) std::cout << node->getString() << std::endl;
        else std::cout << "\\\\not an end" << std::endl;
    }

}

std::map<size_t, std::vector<std::string>>
ACSearch::search(const std::string& text)
{
    std::map<size_t, std::vector<std::string>> map_pos;
    if(!root) return map_pos;

    if(logger) log << "search initiated. Initial state is root" << std::endl;

    auto pCurrentState = root;

    for(auto it = text.begin(); it != text.end(); it++)
    {
        char c = *it;
        // calculate new state
        if(logger) log << "~~~calculate new state" << std::endl;
        while(true)
        {
            // if state has child with edge 'c', go in
            if(pCurrentState->children.find(c) != pCurrentState->children.end())
```

```
            {
                try
                {
                    pCurrentState = pCurrentState->children.at(c);
                }
                catch(std::out_of_range)
                {
                    // find failed
                    throw std::out_of_range("Wierd thing in <search>, when trying go to the child");
                }

                if(logger) log << "current state has child with \'" << c << "\' character. "
                               << "Its id is " << pCurrentState->id() << std::endl;
                break;
            }

            // otherwise we gonna go deeper to the root using suffix
            if(pCurrentState == root) break;

            try
            {
                pCurrentState = suffix.at(pCurrentState);
            }
            catch(std::out_of_range)
            {
                throw std::out_of_range("Can't access suffix link of vertex, "
                                        "check <search> function, "
                                        "it's actually broken");
            }

            if(logger) log << "There is no child with \'" << c << "\' character. "
                           << "Set state as suffix link. New state is " << pCurrentState->id() << std::endl;
        }
        if(logger) log << "~~~new state is " << pCurrentState->id() << std::endl << std::endl;


        auto pSuffixCrawl = pCurrentState;
        // check all suffixes of current state

        if(pCurrentState != root && logger)
            log << "new state is not root, we have to check node and its suffix path\n"
                << "there could be ends of patterns" << std::endl;
        while(pSuffixCrawl != root)
        {
            // when match
            if(pSuffixCrawl->end())
            {
                if(logger) log << "state " << pSuffixCrawl->id() << " is end of pattern. "
                               << "pattern found on position "
                               << (it - text.begin()) + 1 - pSuffixCrawl->getString().size() << std::endl;
                size_t index = (it - text.begin()) + 1 - pSuffixCrawl->getString().size();
                map_pos[index].push_back(pSuffixCrawl->getString());
            }

            try
            {
                pSuffixCrawl = suffix.at(pSuffixCrawl);
            }
            catch(std::out_of_range)
            {
                throw std::out_of_range("Can't access suffix link of vertex, "
                                        "check <search> function, "
                                        "it's actually broken");
            }
            if(logger) log << "go deeper on the suffix link. Now, state to check is "
                           << pSuffixCrawl->id() << std::endl;

        }
        if(logger) log << std::endl;
    }
    return map_pos;
}

int main()
{
    ACSearch t(1);

    std::string text;
    std::getline(std::cin, text);

    size_t n;
    std::cin >> n;
    std::vector<std::string> string_lib;
    for(size_t i = 0; i < n; i++)
    {
        std::string str;
        std::cin >> str;
        t.insert(str);
        string_lib.push_back(str);
    }


    t.turnIntoMachine();

    auto map_pos = t.search(text);

    for(auto index_item: map_pos)
    {
        std::sort(index_item.second.begin(), index_item.second.end(),
                  [&string_lib](const std::string& a, const std::string& b)
                  {
                      return std::find(string_lib.begin(), string_lib.end(), a) <
                             std::find(string_lib.begin(), string_lib.end(), b);
                  });
        for(auto string_item: index_item.second)
        {
            size_t string_number = std::find(string_lib.begin(), string_lib.end(), string_item)
                                   - string_lib.begin();
            std::cout << index_item.first+1 << " " << string_number+1 << std::endl;
        }
    }

    t.print_longest_links();

    return 0;
}
```

Приложение В
Вариант с джокером

```
#include <iostream>
#include <queue>
#include <map>
#include <vector>
#include <string>
#include <fstream>
#include <algorithm>

class TrieNode;

class WildcardSearch
{
    // default aho-corasick part
    TrieNode *root;
    size_t size;
    std::map<TrieNode*, TrieNode*> suffix;

    // wildcards addition
    // key - substring of pattern
    // value - vector of positions, where we can find this substring
    size_t pattern_size;
    std::map<std::string, std::vector<size_t>> subP_map;
    size_t subP_count;

    std::ofstream log;
    bool logger;

    // encapasulated default aho-corasick funcitons
    void calcSuffixLink(TrieNode* node);
    void turnIntoMachine();
    void insert(const std::string& key);

public:
    WildcardSearch(bool log_on = 1);
    WildcardSearch();
    ~WildcardSearch();

    bool empty() const;

    void setPattern(const std::string& key, char joker);

    // returns empty vector, if trie is empty
    std::vector<TrieNode*> getAllNodes();

    void print();

    std::vector<size_t> search(const std::string& text);
};

class TrieNode
{
    friend class ACSearch;
    friend class WildcardSearch;

    size_t _id;
    std::map<char, TrieNode*> children;
    TrieNode *parent;
    char toParent;
    bool bEnd;
    std::string _str;

    TrieNode(size_t id, TrieNode* parent = nullptr, char toParent = 0):
            _id(id), parent(parent), toParent(toParent), bEnd(0)
    {};

public:
    bool end() const
    {
        return bEnd;
    }

    const std::string& getString() const
    {
        return _str;
    }

    size_t id() const
    {
        return _id;
    }
};


WildcardSearch::WildcardSearch(bool log_on):
        root(nullptr), logger(log_on)
{
    if(logger) log.open("log", std::ofstream::out | std::ofstream::trunc);
```

```
}

WildcardSearch::WildcardSearch():
      root(nullptr), logger(1)
{
   log.open("log", std::ofstream::out | std::ofstream::trunc);
}


WildcardSearch::~WildcardSearch()
{
   delete root;
   log.close();
}


void WildcardSearch::calcSuffixLink(TrieNode* node)
{

   if(logger)
   {
      log << "suffix-link calculator initiated for node: " << node->id() << std::endl;
   }

   // suffix link of root is root itself
   if(node == root)
   {
      if(logger) log << "suffix[root] = root" << std::endl;

      suffix[node] = root;
      return;
   }

   // suffix links of root children are root
   if(node->parent == root)
   {
      if(logger) log << "this is a child of root. suffix = root" << std::endl;
      suffix[node] =  root;
      return;
   }

   // for computate the suffix link
   // we need the suffix link of node parent
   TrieNode *pCurrBetterNode;
   try
   {
      pCurrBetterNode = suffix.at(node->parent);
   }
   catch (std::out_of_range)
   {
      // trying to calculate prefix, before parent
      throw std::out_of_range("can't take first parent suffix, "
                              "check <calcSuffixLink>, "
                              "it's actually broken");
   }

   // and the character, which that moves us to node
   char cParentChar = node->toParent;

   if(logger) log << "parent suffix " << pCurrBetterNode->id();
   if(logger) log << "jump through suffix link" << std::endl;
   while(true)
   {
      // try to find the children of parent suffix
      // with the same key as to current node
      if(pCurrBetterNode->children.find(cParentChar) != pCurrBetterNode->children.end())
      {
         if(logger) log << "child found: " << pCurrBetterNode->children[cParentChar]->id()
                        << " id. Set suffix link on it" << std::endl;
         suffix[node] = pCurrBetterNode->children[cParentChar];
         break;
      }

      // otherwise jump through suffix links
      // until reach root
      if(pCurrBetterNode == root)
      {
         if(logger) log << "suffix path led to root, suffix link set on root" << std::endl;
         suffix[node] = root;
         break;
      }

      try
      {
         pCurrBetterNode = suffix.at(pCurrBetterNode);
         if(logger) log << "Next suffix. Current node is "
                        << pCurrBetterNode->id() << std::endl;
      }
      catch(std::out_of_range)
      {
         // trying to calculate prefix of node, which doesn't
         // have full prefix path
         throw std::out_of_range("can't take next parent suffix, "
                                 "check <calcSuffixLink>, "
                                 "it's actually broken");
      }
   }
}


void WildcardSearch::turnIntoMachine()
{
   if(logger) log << "Machinizer initiated" << std::endl;
   if(!root) return;

   suffix.clear();

   // find suffix links using breadth first search
   // which needs queue
   std::queue<TrieNode*> q;

   q.push(root);
   while(!q.empty())
   {
      auto pCurrNode = q.front();
```

```
            q.pop();

            if(logger) log << "built suffix link for " << pCurrNode->id() << " node. " << std::endl;
            calcSuffixLink(pCurrNode);
            if(logger) log << "Suffix link leads to " << suffix.at(pCurrNode)->id() << " node"
                        << std::endl << std::endl;

            for(auto child: pCurrNode->children){
               q.push(child.second);
            }
         }
      }


      void WildcardSearch::insert(const std::string& key)
      {
         if(logger) log << "add key: " << key << std::endl;
         if(empty()){
            if(logger) log << "trie is empty, root created" << std::endl;
            root = new TrieNode(0);
            size = 1;
         }

         auto pCrawl = root;
         for(auto c: key)
         {
            if(pCrawl->children.count(c) == 0)
            {
               if(logger) log << "create children with character: " << c << std::endl;
               pCrawl->children[c] = new TrieNode(size, pCrawl, c);
               size++;
            }
            else
            {
               if(logger) log << "already have a child with that character, go deeper" << std::endl;
            }
            pCrawl = pCrawl->children[c];
         }
         pCrawl->bEnd = 1;
         pCrawl->_str = key;

         if(logger) log << std::endl;
      }


      bool WildcardSearch::empty() const
      {
         return !root;
      }


      void WildcardSearch::setPattern(const std::string& key, char joker)
      {
         subP_map.clear();
         subP_count = 0;
         pattern_size = key.size();

         std::string sCurrP; // current pattern

         for(auto it = key.begin(); it != key.end(); it++)
         {
            char c = *it;
            size_t index = it - key.begin();
            if(c != joker)
            {
               sCurrP.push_back(c);
               if(it == key.end()-1)
               {
                  // save position of this substring
                  size_t sub_index = index + 2;
                  sub_index -= sCurrP.size();
                  subP_map[sCurrP].push_back(sub_index);
                  subP_count++;
               }
            }
            else
            {
               if(!sCurrP.empty())
               {
                  // we are on the position beyond string
                  // therefore we don't need to add one
                  size_t sub_index = index + 1;
                  sub_index -= sCurrP.size();
                  subP_map[sCurrP].push_back(sub_index);
                  subP_count++;
                  sCurrP.clear();
               }
            }
         }

         // put all substrings into aho-corasick machine
         for(auto& item: subP_map)
         {
            insert(item.first);
         }

         // add suffix links into trie
         turnIntoMachine();
      }


      std::vector<TrieNode*> WildcardSearch::getAllNodes()
      {
         std::vector<TrieNode*> nodes;
         if(root == nullptr)
            return nodes;

         // find all graph nodes using breadth first search
         std::queue<TrieNode*> q;
         q.push(root);

         while(!q.empty())
         {
            auto pCurrNode = q.front();
            q.pop();
            nodes.push_back(pCurrNode);
```

```
        for(auto child: pCurrNode->children)
        {
            // it is a tree, there can't be cycles,
            // so, there is no need to mark nodes
            q.push(child.second);
        }
    }
    return nodes;
}


void WildcardSearch::print()
{
    auto nodes = getAllNodes();
    for(auto node: nodes){
        std::cout << node->parent << " " << node << " "
                  << node->toParent << " " << suffix[node] << std::endl;
    }
}


std::vector<size_t> WildcardSearch::search(const std::string& text)
{
    std::vector<size_t> pattern_entries;
    if(!root) return pattern_entries;

    if(logger) log << "search initiated. Initial state is root" << std::endl;

    // subP_entries[i] - how many substrings of pattern
    // macthed on position i
    std::map<size_t, size_t> subP_entries;

    auto pCurrentState = root;
    for(auto it = text.begin(); it != text.end(); it++)
    {
        char c = *it;

        // calculate new state
        if(logger) log << "~~~calculate new state" << std::endl;
        while(true)
        {
            // if state has child with edge 'c', go in
            if(pCurrentState->children.find(c) != pCurrentState->children.end())
            {
                try
                {
                    pCurrentState = pCurrentState->children.at(c);
                }
                catch(std::out_of_range)
                {
                    // find failed
                    throw std::out_of_range("Wierd thing in <search>, when trying go to the child");
                }
                if(logger) log << "current state has child with \'" << c << "\' character. "
                               << "Its id is " << pCurrentState->id() << std::endl;
                break;
            }

            // otherwise we gonna go deeper to the root using suffix
            if(pCurrentState == root) break;

            try
            {
                pCurrentState = suffix.at(pCurrentState);
            }
            catch(std::out_of_range)
            {
                throw std::out_of_range("Can't access suffix link of vertex, "
                                        "check <search> function, "
                                        "it's actually broken");
            }

            if(logger) log << "There is no child with \'" << c << "\' character. "
                           << "Set state as suffix link. New state is " << pCurrentState->id() << std::endl;
        }
        if(logger) log << "~~~new state is " << pCurrentState->id() << std::endl << std::endl;


        auto pSuffixCrawl = pCurrentState;
        // check all suffixes of current state
        if(pCurrentState != root && logger)
            log << "new state is not root, we have to check node and its suffix path\n"
                << "there could be ends of patterns" << std::endl;
        while(pSuffixCrawl != root)
        {
            // when match
            if(pSuffixCrawl->end())
            {
                size_t index = (it - text.begin()) + 1 - pSuffixCrawl->getString().size();

                if(logger) log << "state " << pSuffixCrawl->id() << " is end of pattern. "
                               << "pattern found on position " << index << std::endl;

                index++;

                if(logger) log << "supposed full pattern positions: ";
                bool nowhere = 1;
                for(auto l: subP_map[pSuffixCrawl->getString()])
                {
                    if(l <= index)
                    {
                        nowhere = 0;
                        subP_entries[index - l]++;
                        if(logger) log << index - l << " ";
                    }
                }
                if(logger && nowhere) log << "nowhere";
                if(logger) log << std::endl;
            }

            try
            {
                pSuffixCrawl = suffix.at(pSuffixCrawl);
            }
            catch(std::out_of_range)
```

```
            {
                throw std::out_of_range("Can't access suffix link of vertex, "
                                        "check <search> function, "
                                        "it's actually broken");
            }
            if(logger) log << "go deeper on the suffix link. Now, state to check is "
                           << pSuffixCrawl->id() << std::endl;
        }
    }

    for(auto p: subP_entries)
    {
        if(p.second == subP_count)
        {
            if(p.first + pattern_size <= text.size())
                pattern_entries.push_back(p.first);
        }
    }
    return pattern_entries;
}


int main()
{
    std::string text;
    std::getline(std::cin, text);
    std::string pattern;
    std::cin >> pattern;
    char joker;
    std::cin >> joker;


    WildcardSearch w(1);
    w.setPattern(pattern, joker);
    auto pos = w.search(text);

    for(auto p: pos){
        std::cout << p+1 << std::endl;
    }

    return 0;
}
```