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. 7383 _____ Рудоман В.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,$1 \leq |T| \leq 100000$).

Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p1,\ldots,pn\}$ $1 \leq |pi| \leq 75$

Все строки содержат символы из алфавита $\{A,C,G,T,N\}$

Выход:

Все вхождения образцов из P в T.

Каждое 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:

CCCA

1

CC

Sample Output:

1 1

2 1

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

В шаблоне встречается специальный символ, именуемого 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 шаблоне входит хотя бы один символ не джокер, те шаблоны вида ??? недопустимы.

Все строки содержат символы из алфавита $\{A,C,G,T,N\}$

Вход:

Текст (T,$1 \leq |T| \leq 100000$)

Шаблон (P,$1 \leq |P| \leq 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACT

A$

$

Sample Output:

1

Вариант №3. Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

Описание алгоритма.

Основной идеей алгоритма Ахо-Корасик является построение конечного автомата по требуемым для нахождения шаблонам и дальнейшая работа с ним. Такой автомат получается добавлением суффиксных ссылок в бор.

Построение суффиксных ссылок:

Для каждой из вершин (от родителя к сыновьям вглубь, а не вширь) вызывается функция получения суффиксной ссылки. Если для текущей вершины суффиксная ссылка еще не вычислена и вершина является либо корнем, либо сыном корневой вершины, то суффиксная ссылка – корень дерева. Если же ссылка не вычислена, а вершина отлична от вышеперечисленных, то для получения суффиксной ссылки необходимо вызвать функцию переходов – `goTo(getSuffLink(v->parent), v->ch)`. Рассмотрим на примере. Пусть требуется построить суффиксную ссылку для выделенной вершины.

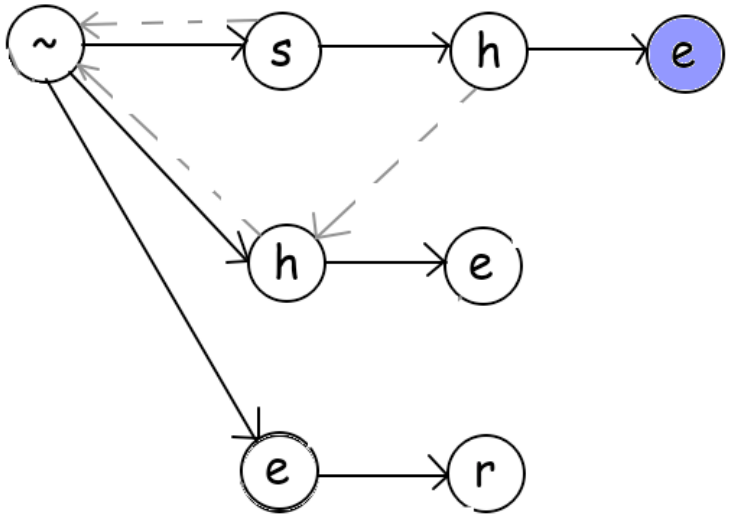

Рисунок 1 – Пример построения суффиксной ссылки.

Суффиксная ссылка показывает, в какую вершину осуществляется переход в случае отсутствия подаваемого на вход символа в сыновьих состояниях. В данном случае мы обработали некоторую подстроку вида *she,* после чего получили произвольный символ. Чтобы не пропустить ни одного вхождения любой другой строки из набора шаблонов, требуется осуществить переход в вершину, путь до которой из корня является наибольшим суффиксом

строки *she*. В данном примере, суффиксной ссылкой фиолетовой вершины будет будет красная вершина.

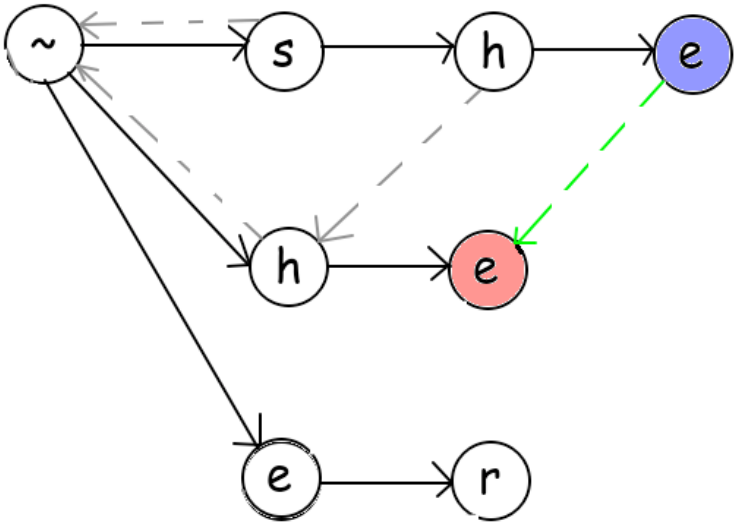

Рисунок 2 – Пример построения суффиксной ссылки.

Бор представляет собой подвешенное дерево с символами на рёбрах (вершинах), хранит наборы строк. Строится путём посимвольного добавления строк-шаблонов, начиная с корня для каждого нового шаблона. Также будем ставить отметки, означающие, что вершина является терминальной, т.е. последней для данного шаблона. Пример бора далее в тесте.

После этого каждая вершина в боре дополняется суффиксными ссылками, т.е. ссылками в вершины, строка в которых является максимальным суффиксом для строки из этой вершины. Суффиксная ссылка корня ведёт в корень.

Для использования преобразованного бора в качестве конечного автомата определим функцию перехода из одного состояния в другое при подаче очередного символа. При наличии сына с символом, равным поданному символу, автомат переходит в эту вершину. Если такого сына нет и вершина корень, то остаётся на месте, иначе функция рекурсивно вызывается для суффиксной ссылки текущей вершины.

После окончания всех нужных построений скармливаем функции перехода посимвольно строку для поиска и, если автомат приходит в

терминальную вершину, выводим позицию нахождения шаблона в тексте и пытаемся вывести входящие в этот шаблон другие шаблоны.

В конце работы (после окончания символов на вход) получим все позиции вхождения шаблонов в текст.

При поиске шаблона-маски он разбивается на безмасочные строки, по которым затем строится автомат Ахо-Корасик и выполняется вышеописанный процесс поиска. Результаты поиска записываются в массив, длиной в строку, в которой производится поиск, таким образом, что в каждой ячейке будет находиться количество шаблонов, начинающихся оттуда с учётом смещения внутри шаблона-маски. В конце работы выводим индексы ячеек, в которых хранится число безмасочных шаблонов. Получаем позиции вхождения шаблона с маской в нашу строку для поиска.

Описание классов и функций.

```
struct Vertex { //представление узла дерева
    char ch; // символ в узле
   //конечная ли вершина (для какого-либо шаблона), номер шаблона, длина шаблона
    std::tuple <bool, size_t, size_t> label;
    Vertex* parent; // родительский узел
    Vertex* suffLink; //суффиксная ссылка
    std::list <Vertex*> children; // список сыновей
  //обычный конструктор
   Vertex(Vertex*  p,  char  ch)  :  ch(ch),  label({false,  0,  0}),  parent(p),
suffLink(nullptr){};
};
```

class Bor содержит единственное поле Vertex* head

Методы класса Bor:

public:

Bor() : head(new Vertex(nullptr, '~'))

Аргументы: отсутствуют

Назначение: стандартный конструктор.

void addString(std::string const &str)

Аргументы: строка, которую необходимо добавить в дерево

Возвращаемое значение:отсутствует

Назначение: добавляет шаблон в дерево

```
void processText(std::string const &str)
```

Аргументы: строка, в которой осуществляется множественный поиск шаблонов

Возвращаемое значение: отсутствует

Назначение: основная функция для алгоритма Ахо-Корасик.

```
void setSuffixLinks()
```

Аргументы: отсутствуют

Возвращаемое значение: отсутствует

Назначение: устанавливает суффиксные ссылки для алгоритма Ахо-Корасик. Они гарантируют, что ни одно вхождение шаблона не будет пропущено.

```
void printF()
```

Аргументы: отсутствуют

Возвращаемое значение: отсутствует

Назначение: точка входа в рекурсивую функцию вывода дерева

```
void inDeep(Vertex* v)
```

Аргументы: вершина-родитель v

Возвращаемое значение: отсутствуют

Назначение: рекурсивная функция, вызывающая для каждого элемента дерева функцию getSuffLink().

```
Vertex* getSuffLink(Vertex* v)
```

Аргументы: вершина v, для которой нужно получить суффиксную ссылку

Возвращаемое значение: вершина, в которую перейдет автомат, при первом несовпадении обрабатываемого символа и имеющихся шаблонов.

Назначение: возвращает суффиксную ссылку для вершины v: вершину в которую перейдет автомат, при первом несовпадении обрабатываемого символа и имеющихся шаблонов. Наличие суффиксных ссылок гарантирует, что ни одно вхождение не будет пропущено.

```
Vertex* goTo(Vertex* v, char ch)   // determines where we go from vertex 'v' having char 'ch'
```

Аргументы: вершина v, откуда осуществлется переход по символу ch.

Возвращаемое значение: узел, в который произойдет переход из вершины v, при обрабатываемом символе ch.

Назначение: функция-переходов отвечает за перемещение по дереву шаблонов. Если из вершины v есть сын с символом ch, то возвращается указатель на этого сына, иначе возвращаемое значение определяется суффиксными ссылками.

```
Vertex* addCharacterInVertex(char newCh, Vertex* vertex)
```

Аргументы: символ ch, вершина vertex

Возвращаемое значение: указатель на вершину, с символом ch.

Назначение: добавляет (при необходимости) символ ch в вершину, являющуюся сыном вершины vertex.

```
void printSubF(Vertex* v, const std::string& prefix) // nice output, don't bother
```

Аргументы: вершина-родитель v, префикс-строка prefix для отображения структуры дерева

Возвращаемое значение: отсутствуют

Назначение: осуществляет вывод дерева на консоль. (рекурсивная функция)

Сложность алгоритма.

O(N + Z), где N –длина текста, Z -- число вхождений шаблонов в текст. (без учета построения дерева, которое выполняется за O(L), где L – суммарная длина всех шаблонов.)

Как узнать по текущему состоянию v, имеется ли совпадение с какими-то строками из набора? Во-первых, понятно, что если мы стоим в листе, то имеется совпадение с тем образцом, который в боре оканчивается в вершине v. Однако это далеко не единственный возможный случай достижения совпадения: если мы, двигаясь по суффиксным ссылкам, мы можем достигнуть одной или нескольких помеченных вершин, то совпадение также будет, но уже для образцов, оканчивающихся в этих состояниях.

Таким образом, если в каждой помеченной вершине хранить номер образца, оканчивающегося в ней (или список номеров, если допускаются повторяющиеся образцы), то мы можем для текущего состояния за O (n) найти номера всех образцов, для которых достигнуто совпадение, просто пройдя по суффиксным ссылкам от текущей вершины до корня. Однако это недостаточно эффективно. Можно заметить, что движение по суффиксным ссылкам можно соптимизировать, предварительно посчитав для каждой вершины ближайшую к ней помеченную вершину, достижимую по суффиксным ссылкам (это называется "функцией выхода"). Эту величину можно считать ленивой динамикой за линейное время. Тогда для текущей вершины мы сможем за O (1) находить следующую в суффиксном пути помеченную вершину, т.е. следующее совпадение. Тем самым, на каждое совпадение будет тратиться O (1) действий, и в сумме получится асимптотика **O(N + Z).**

Сложность по памяти составит **O(M)**, где **M** – суммарная длина всех шаблонов, поскольку в худшем случае в дереве придется хранить каждую вершину отдельно.

Тестирование.

а)Тестирование основного алгоритма Ахо-Корасик

Пунктирные стрелки – суффиксные ссылки.

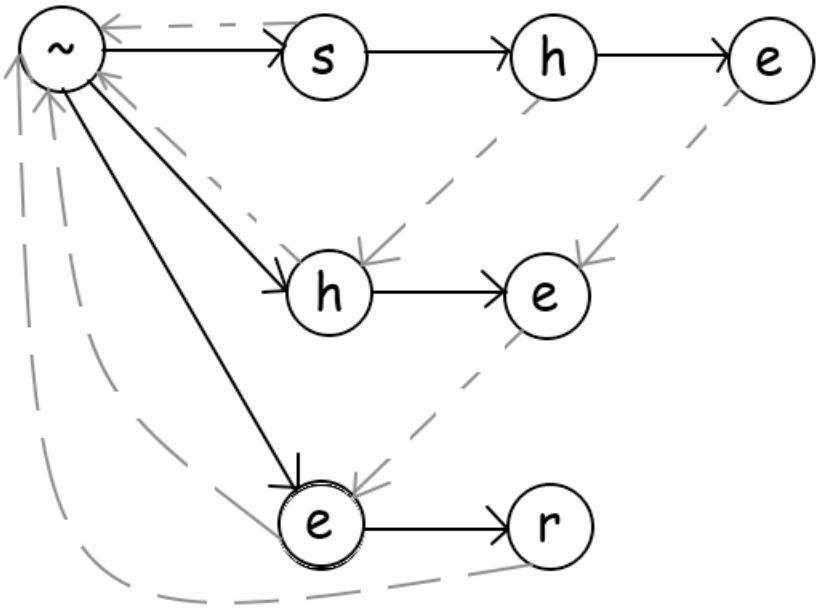

Рисунок 3 – Граф к тесту 1

```
ushers
3
he
she
er

Builded bor: ( • stands for terminal vertex)
~
├── h
│   └── e•
├── s
│   └── h
│       └── e•
└── e
    └── r•


Строим суффиксные ссылки

Получаем суф ссылку для вершины ~
Ссылки нет
Вершина - голова или её сын
Наша ссылка: ~

Получаем суф ссылку для вершины h
Ссылки нет
Вершина - голова или её сын
Наша ссылка: ~

Получаем суф ссылку для вершины e
Ссылки нет
Вызовём функцию goTo с аргументами: h и e

Получаем суф ссылку для вершины h
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и e. Сравниваем s и e. Сравниваем e и e. Совпадение! Возвращаем e
Наша ссылка: e

Получаем суф ссылку для вершины s
Ссылки нет
Вершина - голова или её сын
Наша ссылка: ~

Получаем суф ссылку для вершины h
Ссылки нет
Вызовём функцию goTo с аргументами: s и h

Получаем суф ссылку для вершины s
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и h. Совпадение! Возвращаем h
Наша ссылка: h

Получаем суф ссылку для вершины e
```

Рисунок 4 – Тест 1

```
Ссылки нет
Вызовём функцию goTo с аргументами: h и e

Получаем суф ссылку для вершины h
Наша ссылка: h
goTo. Просмотрим сыновей переданной вершины h
Сравниваем e и e. Совпадение! Возвращаем e
Наша ссылка: e

Получаем суф ссылку для вершины e
Ссылки нет
Вершина - голова или её сын
Наша ссылка: ~

Получаем суф ссылку для вершины r
Ссылки нет
Вызовём функцию goTo с аргументами: e и r

Получаем суф ссылку для вершины e
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и r. Сравниваем s и r. Сравниваем e и r. Совпадений не найдено!
Суф ссылка родителя - голова -> возвращаем голову!
Наша ссылка: ~

Суффиксные ссылки построены! Начинаем обработку текста.
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и u. Сравниваем s и u. Сравниваем e и u. Совпадений не найдено!
Суф ссылка родителя - голова -> возвращаем голову!
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и s. Сравниваем s и s. Совпадение! Возвращаем s

Получаем суф ссылку для вершины s
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины s
Сравниваем h и h. Совпадение! Возвращаем h

Получаем суф ссылку для вершины h
Наша ссылка: h

Получаем суф ссылку для вершины h
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины h
Сравниваем e и e. Совпадение! Возвращаем e
Вывод для stepik:
2 2

Получаем суф ссылку для вершины e
Наша ссылка: e
Вывод для stepik:
3 1

Получаем суф ссылку для вершины e
Наша ссылка: e
```

Рисунок 5 – Тест 1

```
Наша ссылка: e

Получаем суф ссылку для вершины e
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины e
Совпадений не найдено!
Вызов goTo из goTo, аргументы: getsufflink(e) и e

Получаем суф ссылку для вершины e
Наша ссылка: e
goTo. Просмотрим сыновей переданной вершины e
Совпадений не найдено!
Вызов goTo из goTo, аргументы: getsufflink(e) и e

Получаем суф ссылку для вершины e
Наша ссылка: e
goTo. Просмотрим сыновей переданной вершины e
Сравниваем r и r. Совпадение! Возвращаем r
Вывод для stepik:
4 3
Max suffix link: 3
Max finite link: 2
```

Рисунок 6 – Тест 1

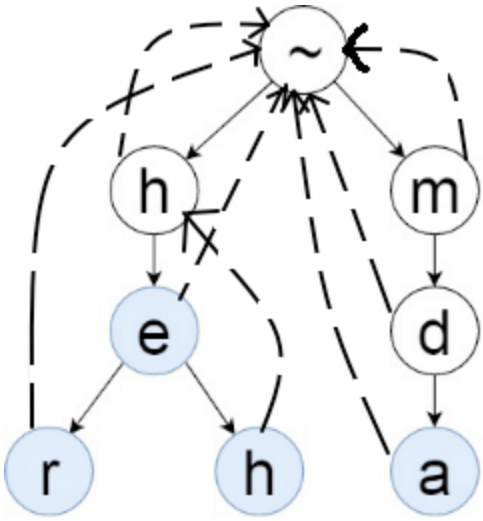

Рисунок 7 – Граф к тесту 2

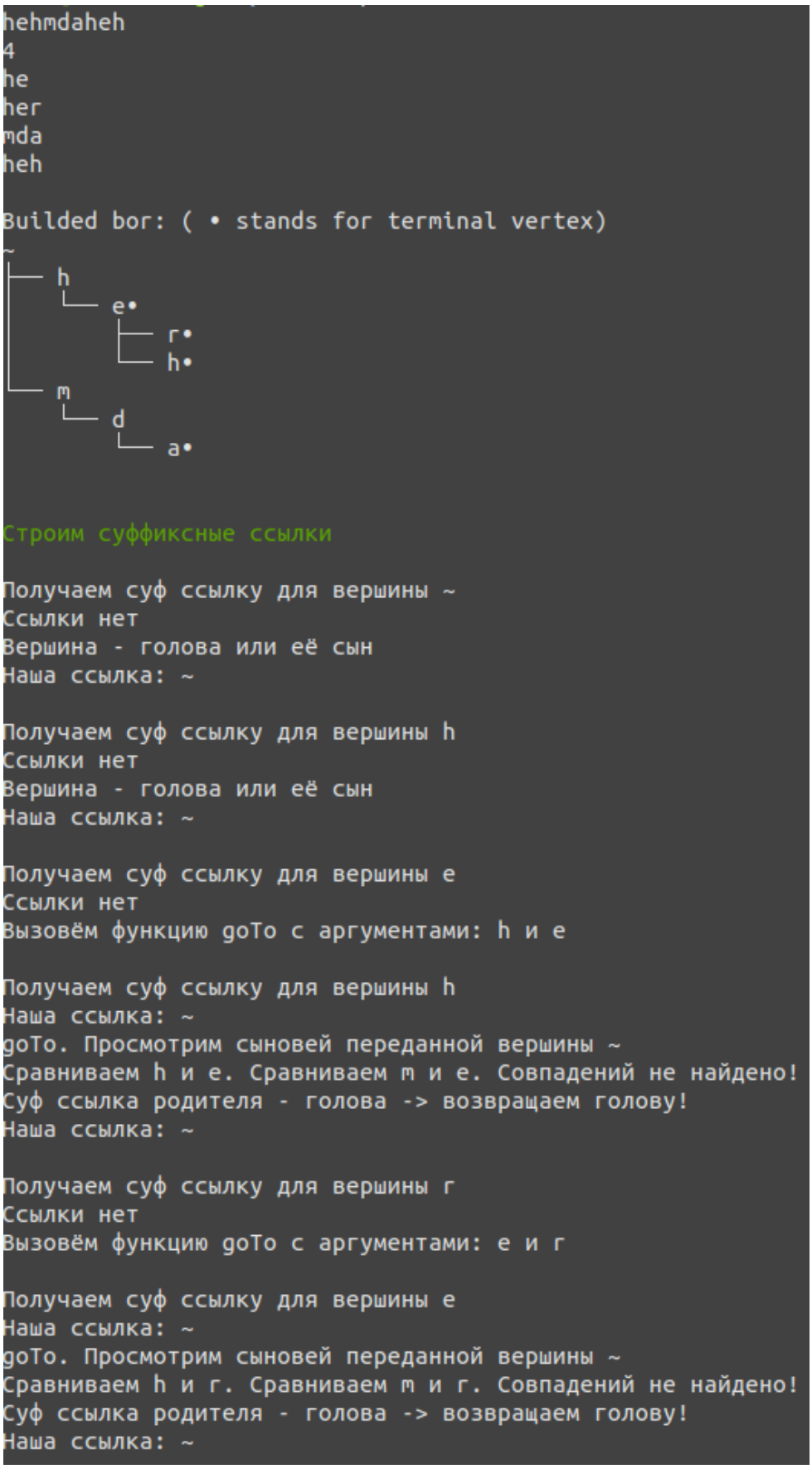

Рисунок 8 – Тест 2

```
Получаем суф ссылку для вершины h
Ссылки нет
Вызовём функцию goTo с аргументами: e и h

Получаем суф ссылку для вершины e
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и h. Совпадение! Возвращаем h
Наша ссылка: h

Получаем суф ссылку для вершины m
Ссылки нет
Вершина - голова или её сын
Наша ссылка: ~

Получаем суф ссылку для вершины d
Ссылки нет
Вызовём функцию goTo с аргументами: m и d

Получаем суф ссылку для вершины m
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и d. Сравниваем m и d. Совпадений не найдено!
Суф ссылка родителя - голова -> возвращаем голову!
Наша ссылка: ~

Получаем суф ссылку для вершины a
Ссылки нет
Вызовём функцию goTo с аргументами: d и a

Получаем суф ссылку для вершины d
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и a. Сравниваем m и a. Совпадений не найдено!
Суф ссылка родителя - голова -> возвращаем голову!
Наша ссылка: ~

Суффиксные ссылки построены! Начинаем обработку текста.
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и h. Совпадение! Возвращаем h

Получаем суф ссылку для вершины h
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины h
Сравниваем e и e. Совпадение! Возвращаем e
Вывод для stepik:
1 1

Получаем суф ссылку для вершины e
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины e
Сравниваем r и h. Сравниваем h и h. Совпадение! Возвращаем h
Вывод для stepik:
1 4
```

Рисунок 9 – Тест 2

```
Получаем суф ссылку для вершины h
Наша ссылка: h

Получаем суф ссылку для вершины h
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины h
Совпадений не найдено!
Вызов goTo из goTo, аргументы: getsufflink(h) и h

Получаем суф ссылку для вершины h
Наша ссылка: h
goTo. Просмотрим сыновей переданной вершины h
Сравниваем e и m. Совпадений не найдено!
Вызов goTo из goTo, аргументы: getsufflink(h) и h

Получаем суф ссылку для вершины h
Наша ссылка: ~
goTo. Просмотрим сыновей переданной верш
Сравниваем h и m. Сравниваем m и m. Совп

Получаем суф ссылку для вершины m
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины m
Сравниваем d и d. Совпадение! Возвращаем d

Получаем суф ссылку для вершины d
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины d
Сравниваем a и a. Совпадение! Возвращаем a
Вывод для stepik:
4 3

Получаем суф ссылку для вершины a
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины a
Совпадений не найдено!
Вызов goTo из goTo, аргументы: getsufflink(a) и a

Получаем суф ссылку для вершины a
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины ~
Сравниваем h и h. Совпадение! Возвращаем h

Получаем суф ссылку для вершины h
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины h
Сравниваем e и e. Совпадение! Возвращаем e
Вывод для stepik:
7 1

Получаем суф ссылку для вершины e
Наша ссылка: ~
goTo. Просмотрим сыновей переданной вершины e
Сравниваем r и h. Сравниваем h и h. Совпадение! Возвращаем h
Вывод для stepik:
```

```
Вывод для stepik:
7 4

Max suffix link: 2
Max finite link: 1
```

исунок 10 – Тест 2

исунок 11 – Тест 2

)Тестирование с джокером.

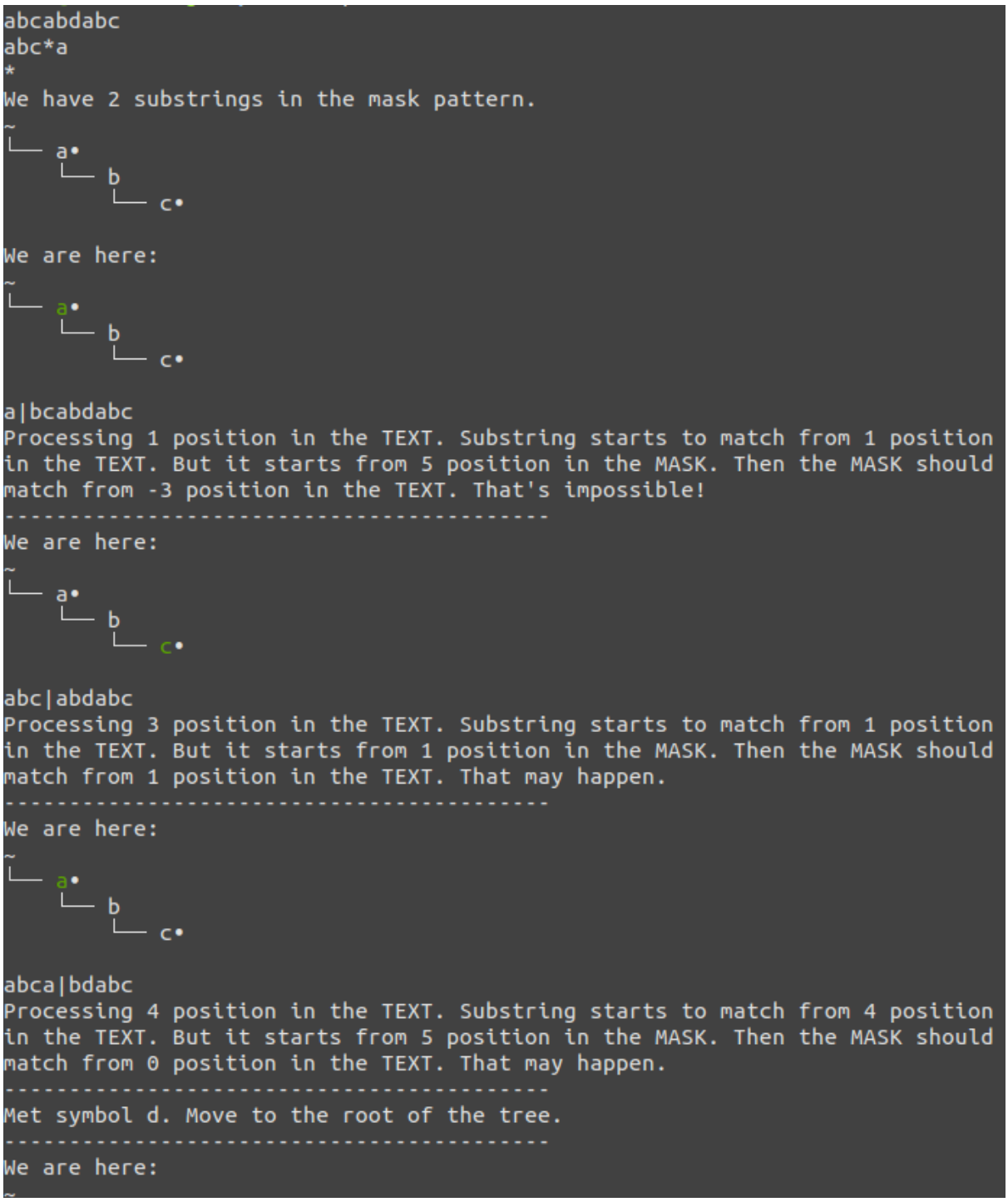

Рисунок 12 – Тест 3

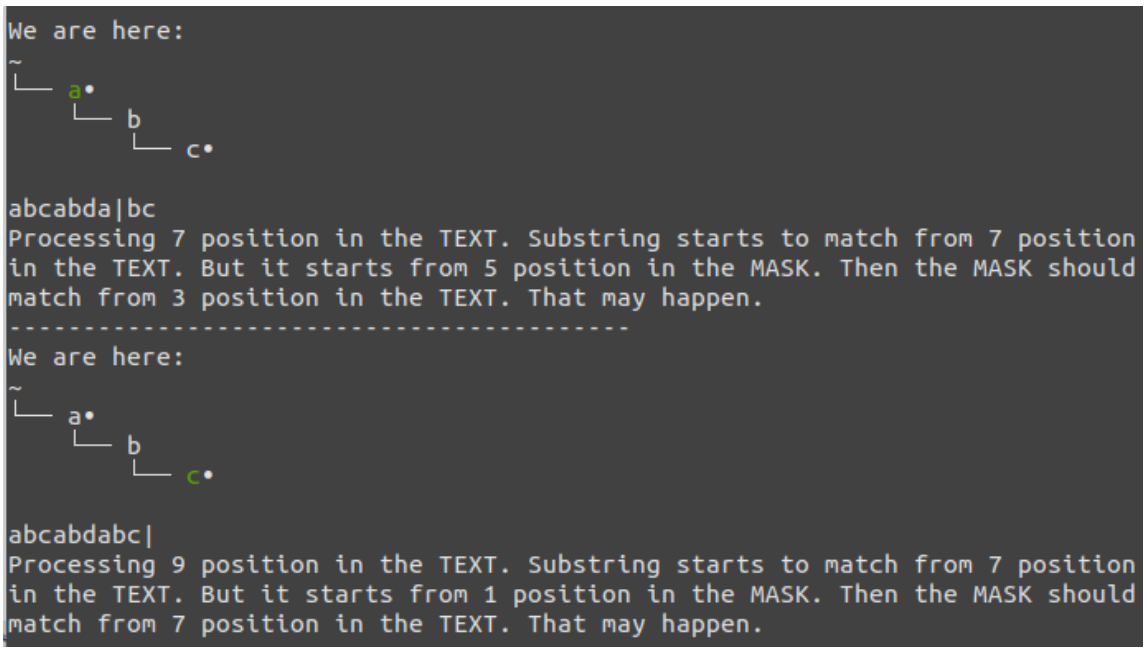

Рисунок 13 – Тест 3

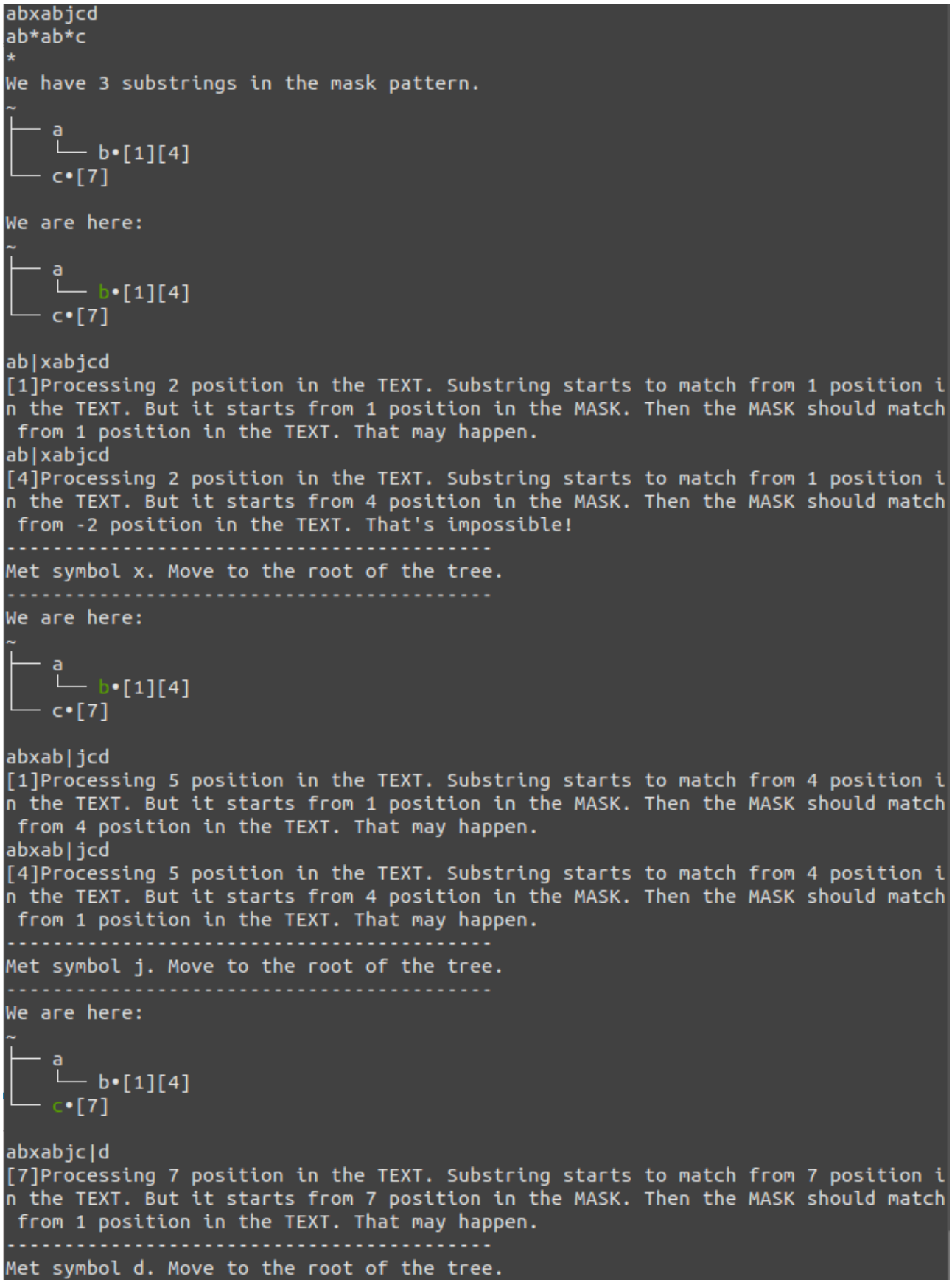

Рисунок 14 – Тест 4

Вывод.

В процессе выполнения лабораторной работы был изучен и реализован на практике алгоритм Ахо-Корасик для нахождения множества подстрок в строке. Сложность исследованного алгоритма составила **O(N + Z).**

Приложение А

Исходный код для программы 1

```
#include <iostream>
#include <list>
#include <algorithm>
#include <cstring>
#include <iomanip>

bool test = false;

struct Vertex {
    char ch;
    std::tuple <bool, size_t, size_t> label; // 1 - is_terminal_vertex?, 2 - patter_num, 3 - dist_from_root
    Vertex* parent;
    Vertex* suffLink;
    std::list <Vertex*> children;

    Vertex(Vertex* p, char ch) : ch(ch), label({false, 0, 0}), parent(p), suffLink(nullptr){};
};

class Bor {
    Vertex* head;
public:
    Bor() : head(new Vertex(nullptr, '~')) {} ;

    void addString(std::string const &str){ // adding patterns in bor
        static int number = 1;
        auto ptr = head;                    // starting to move from head
        for (auto const &elem : str)
            ptr = addCharacterInVertex(elem, ptr);
        ptr->label = { true, number, str.size() };
        number++;
    }

    void processText(std::string const &str){
        Vertex* cur  = head;
```

```
            for (size_t i = 0; i < str.size(); i++){    // going through the text
                cur = goTo(cur, str[i]);
                for (Vertex* ptr = cur; ptr != head; ptr = getSuffLink(ptr))
                    if (std::get<0>(ptr->label)){       // if vertex is termial (have
pattern ended)
                        if(test) std::cout << "\033[32mВывод для stepik: \033[0m" <<
std::endl;
                        std::cout << i - std::get<2>(ptr->label) + 2 << " " <<
std::get<1>(ptr->label) << std::endl;
                    }
            }
        }

        void setSuffixLinks(){
            inDeep(head);
        }

        void printF(){
            std::cout << head->ch << std::endl;
            if(!head->children.empty()){
                printSubF(head, "");
                std::cout << std::endl;
            }
        }

    private:
        void inDeep(Vertex* v){
            v->suffLink = getSuffLink(v);
            for (auto const &el : v->children)
                inDeep(el);
        }

        Vertex* getSuffLink(Vertex* v){
            if (test) std::cout << "\nПолучаем суф ссылку для вершины " << v->ch <<
"\n";
            if (v->suffLink == nullptr){
                if (test) std::cout << "Ссылки нет\n";
                if (v == head || v->parent == head){
                    v->suffLink = head;
                    if (test) std::cout << "Вершина - голова или её сын\n";
                }
```

```
                else {
                    if (test) std::cout << "Вызовём функцию goTo с аргументами: " <<
v->parent->ch << " и " << v->ch << "\n";
                    v->suffLink = goTo(getSuffLink(v->parent), v->ch);
                }
            }
            if (test) std::cout << "Наша ссылка: " << v->suffLink->ch << std::endl;
            return v->suffLink;
        }

        Vertex* goTo(Vertex* v, char ch){   // determines where we go from vertex 'v'
having char 'ch'
            if (test) std::cout << "goTo. Просмотрим сыновей переданной вершины " <<
v->ch << std::endl;
            for (auto const &elem : v->children){
                if (test) std::cout << "Сравниваем " << elem->ch << " и " << ch <<
std::endl;
                if (elem->ch == ch){
                    if (test) std::cout << "Совпадение! Возвращаем " << elem->ch <<
std::endl;
                    return elem;
                }
            }
            if (test) std::cout << "Совпадений не найдено!\n";
            if (v == head){
                if (test) std::cout << "Суф ссылка родителя - голова -> возвращаем
голову!\n";
                return head;
            }
            if (test) std::cout << "Вызов getlink из goTo, аргументы: getsufflink("
<< v->ch << ") и " << v->ch << "\n";
            return goTo(getSuffLink(v), ch);
        }

        Vertex* addCharacterInVertex(char newCh, Vertex* vertex){
            for (auto &el: vertex->children)
                if (el->ch == newCh)           // found the character
                    return el;
            Vertex* newElem = new Vertex(vertex, newCh); // not found -> adding a new
node
            vertex->children.push_back(newElem);
```

```
            return newElem;
        }

        void printSubF(Vertex* v, const std::string& prefix){ // nice output, don't
bother
            auto tmp = v->children.begin();
            while(1){
                std::cout << prefix;
                if(++tmp != v->children.end()){
                    --tmp;
                    std::cout << "├── ";
                    std::string newPrefix = prefix + "│   ";
                    std::cout << (*tmp)->ch;
                    if(std::get<0>((*tmp)->label))
                        std::cout << "•";
                    std::cout << std::endl;
                    if(!(*tmp)->children.empty())
                        printSubF(*tmp, newPrefix);
                }
                else{
                    --tmp;
                    std::cout <<  "└── " << (*tmp)->ch;
                    if(std::get<0>((*tmp)->label))
                        std::cout << "•";
                    std::cout << std::endl;
                    if(!(*tmp)->children.empty()){
                        printSubF(*tmp, prefix + "    ");
                    }
                    return;
                }
                ++tmp;
            }
        }

    };

    int main(int argc, char** argv) {
       if (argc == 2 && !strcmp(argv[1], "-test\0"))
             test = true;
        Bor bor;
        std::string HayStack;
```

```
            std::string pattern;
            int numberOfPatterns;
            std::cin >> HayStack >> numberOfPatterns;
            for (int i = 0; i < numberOfPatterns; i++){
                std::cin >> pattern;
                bor.addString(pattern);
            }
            bor.setSuffixLinks();
            if (test) {
                std::cout << "\nBuilded bor: ( • stands for terminal vertex)" <<
std::endl;
                bor.printF();
                std::cout << std::endl;
            }

            bor.processText(HayStack);
            return 0;
        }
```

Приложение Б

Исходный код для программы 2

```
#include <iostream>
#include <list>
#include <algorithm>
#include <iomanip>
#include <vector>

struct Vertex {
    char ch;
    std::tuple < bool, size_t, size_t > label; // 1 - is_terminal_vertex?, 2 -
patter_num, 3 - dist_from_root
    Vertex* parent;
    Vertex* suffLink;
    std::list <Vertex*> children;
    std::vector <int> positionsInOriginalText;

    Vertex(Vertex* p, char ch) : ch(ch), label({false, 0, 0}), parent(p),
suffLink(nullptr){};

};

class Bor {
    Vertex* head;
public:
    Bor() : head(new Vertex(nullptr, '~')) {} ;
    void addString(std::string const &str, int positionInOriginalText){
        static int number = 1;
        auto ptr = head;
        for (auto const &elem : str)
            ptr = addCharacterInVertex(elem, ptr);
        ptr->label = { true, number, str.size() };
        ptr->positionsInOriginalText.push_back(positionInOriginalText);
        number++;
    }

    void processText(std::string const &str, int numberOfSubStrings){
        Vertex* cur  = head;
        std::vector <int> helpArray(str.size());

        for (size_t i = 0; i < str.size(); i++){
            cur = getLink(cur, str[i]);
            for (Vertex* ptr = cur; ptr != head; ptr = getSuffLink(ptr))
                if (std::get<0>(ptr->label)){
                    int position = i - std::get<2>(ptr->label) + 1 + 1;
                    int index = std::get<1>(ptr->label);

                    for (auto const startPos : ptr->positionsInOriginalText){
                        int arr_index = position;
                        arr_index -= startPos;
                        arr_index += 1;
                        if (arr_index >= 0) helpArray[arr_index]++;
                    }
                }
        }
        for (long int i = 0; i < helpArray.size(); i++){
            if (helpArray[i] == numberOfSubStrings)
```

```
                std::cout << i << std::endl;
            }
        }

        void setSuffixLinks(){
            inDeep(head);
        }

        void printF(){
            std::cout << head->ch << std::endl;
            if(!head->children.empty()){
                printSubF(head, "");
                std::cout << std::endl;
            }
        }

    private:
        void inDeep(Vertex* v, size_t indent = 0){
            v->suffLink = getSuffLink(v);
            for (auto const &el : v->children)
                inDeep(el, indent+1);
        }

        Vertex* getSuffLink(Vertex* v){
            if (v->suffLink == nullptr){
                if (v == head || v->parent == head)
                    v->suffLink = head;
                else
                    v->suffLink = getLink(getSuffLink(v->parent), v->ch);
            }
            return v->suffLink;
        }

        Vertex* getLink(Vertex* v, char ch){
            for (auto const &elem : v->children)
                if (elem->ch == ch)
                    return elem;
            if (v == head)
                return head;
            return getLink(getSuffLink(v), ch);
        }

        Vertex* addCharacterInVertex(char newCh, Vertex* vertex){
            for (auto &el: vertex->children)
                if (el->ch == newCh)
                    return el;
            Vertex* newElem = new Vertex(vertex, newCh);
            vertex->children.push_back(newElem);
            return newElem;
        }

        void printSubF(Vertex* v, const std::string& prefix){ // nice output, don't bother
            auto tmp = v->children.begin();
            while(1){
                std::cout << prefix;
                if(++tmp != v->children.end()){
                    --tmp;
                    std::cout << "├── ";
                    std::string newPrefix = prefix + "│   ";
                    std::cout << (*tmp)->ch;
                    if(std::get<0>((*tmp)->label))
                        std::cout << "•";
```

```
                std::cout << std::endl;
                if(!(*tmp)->children.empty())
                    printSubF(*tmp, newPrefix);
            }
            else{
                --tmp;
                std::cout <<  "└── " << (*tmp)->ch;
                if(std::get<0>((*tmp)->label))
                    std::cout << "•";
                std::cout << std::endl;
                if(!(*tmp)->children.empty()){
                    printSubF(*tmp, prefix + "    ");
                }
                return;
            }
            ++tmp;
        }
    }
};

std::vector<std::pair<std::string, size_t>> parse(std::string str, char joker){
    std::vector <std::pair<std::string, size_t>> ret;
    int curPos = 1;         //absolute position while 'pos' is relative
    for (int pos = str.find(joker); pos != std::string::npos ; pos = str.find(joker)){
//searching for joker
        std::string found = str.substr(0, pos);
        if (found.size() == 0){     // found at the beginning
            str = str.substr(pos+1);
            curPos++;
            continue;
        }
        pos++;
        str = str.substr(pos);   // moving forward
        ret.push_back({found, curPos});
        curPos += pos;
    }
    if (str.size() != 0){
        ret.push_back({str, curPos});
    }
    return ret;
}

int main() {
    Bor bor;
    std::string HayStack;
    std::string pattern;
    char joker;
    std::cin >> HayStack >> pattern >> joker;

    auto withoutMask = parse(pattern, joker);

    for (auto const &el : withoutMask)
        bor.addString(el.first, el.second);
    bor.setSuffixLinks();

    bor.printF();

    bor.processText(HayStack, withoutMask.size());

    return 0;
}
```