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. 8303 ____ Стукалев А.И.

Преподаватель ____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить алгоритм Ахо-Корасик и алгоритм поиска вхождений шаблонов с “джокерами” в строку. Написать программу, реализующую эти алгоритмы работы со строками.

Индивидуализация.

Вариант 5**.** Вычислить максимальное количество дуг, исходящих из одной вершины в боре; вырезать из строки поиска все найденные образцы и вывести остаток строки поиска.

Алгоритм Ахо-Корасик

Задание.

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=$\{p_1, \ldots, p_n\}$ $1 \le |p_i| \le 75$

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.

Пример входных данных

CCCA

1

CC

Пример выходных данных

1 1

2 1

Описание алгоритма.

Несколько строк поиска можно объединить в дерево поиска, так называемый бор (префиксное дерево). Бор является конечным автоматом, распознающим одну строку из m — но при условии, что начало строки известно.

Бор заполняется символами шаблонов, обрабатывая каждый символ по очереди и проверяя существует ли в боре вершина с переходом по текущему символу. Если в неё перехода нет, то вершина создаётся, добавляется в бор и в неё осуществляется переход по текущему символу. Если в неё переход есть, то просто совершается переход по текущему символу.

Алгоритм строит конечный автомат из бора с помощью добавления суффиксальных ссылок, которому затем передаёт строку поиска. Автомат получает по очереди все символы строки и переходит по соответствующим рёбрам. Если перехода по соответствующему символу нет, то осуществляется переход по текущему символу в вершину-сына и переход добавляется в автомат. Если такой переход также отсутствует, то переход осуществляется по хорошей суффиксальной ссылке. Если автомат пришёл в конечное состояние, соответствующая строка словаря присутствует в строке поиска, запоминается местоположение в тексте и номер подстроки.

Для нахождения суффиксальной ссылки для вершины, осуществляется переход в предка вершины, затем переход по суффиксальной ссылке предка и переход по текущему символу. Если предок не имеет суффиксальной ссылки, то для него она определяется аналогичным образом рекурсивно.

Для нахождения хорошей суффиксальной ссылки для вершины, необходимо найти ближайшую терминальную вершину. Если вершина по суффиксальной ссылке будет терминальной, то это и есть искомая вершина, иначе запускается рекурсивный поиск от этой вершины. При использование таких ссылок уменьшится и станет пропорционально количеству искомых

вхождений, оканчивающихся в этой позиции.

Сложность алгоритма по операциям:

Таблица переходов автомата хранится в контейнерах std::map<char, int> next_v, std::map<char, int>auto_movement[k]. Тогда сложность алгоритма по операциям будет равна O((M+N)*log k+t), где M – длина всех символов слов шаблонов, N – длина текста, в котором осуществляется поиск, k – размер алфавита, t – длина всех возможных вхождений всех строк-образцов.

Сложность алгоритма по памяти: O (M+N), где M – сумма всех длин подстрок, N – длина текста, т.к std::map не хранит лишних ячеек памяти вершин, поэтому память пропорциональна количеству вершин в боре.

Описание структур данных алгоритма Ахо-Карасик.

1.

```
struct BohrVrtx
{
   std::map<char, int> next_v; // контейнер, необходимый для переходов по рёбрам в боре.
   std::map<char, int> auto_movement; //контейнер, необходимый для переходов в автомате
   int number; // номер обработанной подстроки
   int parent; //индекс родителя
   int suff_link = -1; //индекс перехода по суффиксальной ссылке
   int goodsufflink = -1; //индекс перехода по хорошей суффиксальной ссылке
   int deep; // длина подстроки
   bool isEnd; //- логическая переменная, отвечающая за то, является ли вершина терминальной
   char parentsymb; // символ родителя
};
```

Структура для хранения информации о вершине.

2.

```
struct Result
{
   int position; - позиция подстроки в тексте
   int number; - номер подстроки.
};
```

Структура для хранения результата работы программы.

3.

class Bohr

{

std::vector <BohrVrtx> bohrTree; - бор

std::vector <Result> result; - контейнер для хранения результата

std::vector <int> forcutstr; - контейнер для удаления подстрок из текста

std::string text; - строка с текстом

int stringNum; - количество подстрок

std::vector <std::string> substrings; - контейнер для хранения подстрок

}

Класс, необходимы для работы алгоритма Ахо-Карасик.

Описание функций алгоритма Ахо-Карасик.

1.

Bohr()

Конструктор класса Bohr, в котором создаётся начальная пустая вершина и добавляется в бор.

2.

BohrVrtx make_bohr_vert(int number, int index, int symbol, bool flag)

Функция класса Bohr, создающая вершину для бора. Number – номер шаблона терминальной вершины, index – индекс родительской вершины, symbol, индекс символа родительской вершины, flag – является ли вершина терминальной. Данные параметры присваиваются к объекту типа BohrVertex. Функция возвращает данный объект

3.

void input_data()

Функция класса Bohr, считывающая текст и подстроки и заполняющая поля text, stringNum и substrings. Функция ничего не возвращает.

4.

void init_bohr()

Функция класса Bohr, добавляющая все подстроки в бор с помощью функции add_string_to_bohr(). Функция ничего не возвращает.

5.

void add_string_to_bohr(std::string str, int number)

Функция класса Bohr, добавляющая символы подстроки в бор. Str – подстрока, number – порядковый номер строки, т.е. какой по счёту данная строка была считана. Функция ничего не возвращает.

6.

int get_suff_link(int v_num)

Функция класса Bohr, для получения суффиксальной ссылки. V_num – индекс вершины, для которой осуществляется поиск по суффиксальной ссылке. Функция возвращает индекс вершины, доступной по суффиксальной ссылке, в контейнере со всеми вершинами автомата.

7.

int get_auto_move(int v_num, int symb)

Функция класса Bohr, для получения вершины и перехода в неё. V_num – индекс вершины из которой осуществляется переход, symb – символ по которому осуществляется переход.

8.

void Aho()

Функция класса Bohr, для осуществления поиска шаблонов в строке, ничего не возвращает

9.

void check_for_substring(int v_num, int letter_position)

Функция класса Bohr, проверяет наличие шаблона в тексте, заносит результат поиска в result. V_num – номер вершины, letter_position – номер текущего символа в тексте. Функция ничего не возвращает.

10.

void print_result()

Функция класса Bohr, выводит результат работы программы, строку с удалёнными из неё шаблонами, максимальное количество дуг, исходящих из

одной вершины и автомат, построенный вовремя работы программы. Функция ничего не возвращает.

11.

int get_good_suff_link(int v_num)

Функция класса Bohr, для получения хорошей суффиксальной ссылки. V_num – индекс вершины, для которой осуществляется поиск по хорошей суффиксальной ссылке. Функция возвращает индекс вершины, доступной по хорошей суффиксальной ссылке, в контейнере со всеми вершинами автомата.

12.

bool compare(Result a1, Result a2)

Функция-компаратор, необходима для сортирования результата. a1 – структура тип result, a2 – структура типа result. Возвращает true, если a2 находится в тексте раньше, чем a1, иначе false. В случае, когда позиции равны, проверяется номер подстроки. Функция возвращает true, если номер подстроки a2 больше чем a1, иначе false.

Тестирование.

1.

Входные данные:
CCCA
1
CC

Выходные данные:

Start add string to bohr CC, number of string: 0

Add vertex C on position 0
Added new vertex in tree: 1
Transition to vertex by symbol: C

Add vertex C on position 1
Added new vertex in tree: 2
Transition to vertex by symbol: C

New terminal vertex added: C

Search for substrings started
Current symbol: C(0)
Try to find transition from 0 by symbol C
Transition to vertex: 1 ,using symbol: C

Getting good suffix link from vertex: 1
Getting suffix link from vertex: 1
Suffix link leads to root!
Suffix link from vertex 1 is 0

Suffix link leads to root!
Good suffix link from vertex 1 is 0

Current symbol: C(1)
Try to find transition from 1 by symbol C
Transition to vertex: 2 ,using symbol: C

The entrance found, index = 1 and pattern is CC
Getting good suffix link from vertex: 2
Getting suffix link from vertex: 2
Getting suffix link from vertex: 1
Suffix link from vertex 1 is 0

Try to find transition from 0 by symbol C
Transition to vertex: 1 ,using symbol: C

Suffix link from vertex 2 is 1

Getting suffix link from vertex: 2
Suffix link from vertex 2 is 1

Getting good suffix link from vertex: 1
Good suffix link from vertex 1 is 0

Good suffix link from vertex 2 is 0

Current symbol: C(2)
Try to find transition from 2 by symbol C
Getting suffix link from vertex: 2
Suffix link from vertex 2 is 1

Try to find transition from 1 by symbol C
Transition to vertex: 2 ,using symbol: C

Go to vertex: 2 ,using symbol: C
Transition added
Transition to vertex: 2 ,using symbol: C

The entrance found, index = 2 and pattern is CC
Getting good suffix link from vertex: 2
Good suffix link from vertex 2 is 0

Current symbol: A(3)
Try to find transition from 2 by symbol A
Getting suffix link from vertex: 2
Suffix link from vertex 2 is 1

Try to find transition from 1 by symbol A
Getting suffix link from vertex: 1
Suffix link from vertex 1 is 0

Try to find transition from 0 by symbol A
Suffix link leads to root!
Go to vertex: 0 ,using symbol: A
Transition added
Go to vertex: 0 ,using symbol: A
Transition added
Transition to vertex: 0 ,using symbol: A

Go to vertex: 0 ,using symbol: A
Transition added

Transition to vertex: 0 ,using symbol: A

End of search

Result of programm:
1 1
2 1

Machine:
Ways from vertex 0:
    symbol: A, index: 0
    symbol: C, index: 1
Ways from vertex 1:
    symbol: A, index: 0
    symbol: C, index: 2
Ways from vertex 2:
    symbol: A, index: 0
    symbol: C, index: 2

Maximum number of outgoing arcs from one vertex of bor is: 1

Removing substrings from text:
Source text: CCCA
Edited text: A

2.

Номер теста	Входные данные	Выходные данные
1	NTAG 3 TAGT TAG T	2 2 2 3
2	NNTAGATNTAG 5 TAGAT AGAT GAT T TAG	3 1 3 4 3 5 4 2 5 3 7 4 9 4 9 5
3	ACAGAG 4 A CAGA GA AG	1 1 2 2 3 1 3 4 4 3 5 1 5 4
4	AACCGGTTNN	2 1

	4 ACC GGT TT CC	3 4 5 2 7 3

Алгоритм Ахо-Карасика с джокером .

Задание.

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 a*b*??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, $1 \le |T| \le 100000$)

Шаблон (P ,$1 \le |P| \le 40$)

Символ джокера

Выход:

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

Номера должны выводиться в порядке возрастания.

Пример выходных данных

```
ACTANCA
A$$A$
$
```

Пример выходных данных

```
1
```

Описание алгоритма.

В начале происходит считывание текста и строки с джокерами. Строка с джокерами разбивается на подстроки по джокерам, позиции подстрок в строке запоминаются. Затем данные подстроки-шаблоны заносятся в бор

Несколько строк поиска можно объединить в дерево поиска, так называемый бор (префиксное дерево). Бор является конечным автоматом, распознающим одну строку из m — но при условии, что начало строки известно. Бор заполняется символами шаблонов, обрабатывая каждый символ по очереди и проверяя существует ли в боре вершина с переходом по текущему символу. Если в неё перехода нет, то вершина создаётся, добавляется в бор и в неё осуществляется переход по текущему символу. Если в неё переход есть, то просто совершается переход по текущему символу.

Алгоритм строит конечный автомат из бора с помощью добавления суффиксальных ссылок, которому затем передаёт строку поиска. Автомат получает по очереди все символы строки и переходит по соответствующим рёбрам. Если перехода по соответствующему символу нет, то осуществляется переход по текущему символу в вершину-сына и переход добавляется в автомат. Если такой переход также отсутствует, то переход осуществляется по хорошей суффиксальной ссылке. Если автомат пришёл в конечное состояние, соответствующая подстрока-шаблон присутствует в строке поиска, увеличивается значение индекса (индекс вхождения подстроки-шаблона в строку минус индекс подстроки-шаблона в строке-шаблоне) в векторе совпадений шаблонов.

После окончания поиска происходит проверка вектора с вхождениями подстрок, если результате работы алгоритма в ячейке вектора найдётся число равное количеству всех подстрок исходной строки, значит в этом индексе строка входит в текст.

Для нахождения суффиксальной ссылки для вершины, осуществляется

переход в предка вершины, затем переход по суффиксальной ссылке предка и переход по текущему символу. Если предок не имеет суффиксальной ссылки, то для него она определяется аналогичным образом рекурсивно.

Сложность алгоритма по операциям:

Таблица переходов автомата хранится в контейнерах std::map<char, int> next_v, std::map<char, int>auto_movement[k]. Тогда сложность алгоритма по операциям будет равна O((M+N)*log k+t), где M – длина всех символов слов шаблонов, N – длина текста, в котором осуществляется поиск, k – размер алфавита, t – длина всех возможных вхождений всех строк-образцов.

Сложность алгоритма по памяти:

O (2M+2N +W), где M –длина всех символов слов шаблона, N – длина текста, в котором осуществляется поиск, W - количество подшаблонов. (длина строки с джокерами + длина текста + количество подшаблонов + массив подшаблонов + массив, в котором отмечается вхождения шаблонов в текст)

Описание структур данных алгоритма Ахо-Карасик с джокером.

1.

```
struct BohrVrtx
{
   std::map<char, int> next_v; // контейнер, необходимый для переходов по рёбрам в боре.
   std::map<char, int> auto_movement; // контейнер, необходимый для переходов в автомате
   std::vector<int> number; // контейнер с номер обработанной подстроки
   int parent; //индекс родителя
   int suff_link = -1; //индекс перехода по суффиксальной ссылке
   int goodsufflink = -1; - индекс перехода по хорошей суффиксальной ссылке
   int deep; // длина подстроки
   bool isEnd; //- логическая переменная, отвечающая за то, является ли вершина терминальной
   char parentsymb; // символ родителя
};
```

Структура для хранения информации о вершине.

2.

```
class Bohr
{
   std::vector <BohrVrtx> bohrTree; // бор
   std::string text; - исходный текст
   std::string stringwithjoker; - строка с джокером
   std::vector <int> match; //вектор для запоминания вхождения подстрок в текст
   std::vector <std::string> substrings; //  контейнер с подстроками
   int stringNum; // общее количество подстрок
   int currcount; // текущее количество обработанных подстрок
   char joker; - символ джокера
   std::vector <int> jokerpositions; //контейнер с позициями подстрок в исходной
строке
}
```

Класс, необходимы для работы алгоритма Ахо-Карасик.

Описание функций алгоритма Ахо-Карасик.

1.

Bohr()

Конструктор класса Bohr, в котором создаётся начальная пустая вершина и добавляется в бор.

2.

BohrVrtx make_bohr_vert(int parent, char symbol, bool flag)

Функция класса Bohr, создающая вершину для бора. parent– индекс родительской вершины, symbol - символ родительской вершины, flag – является ли вершина терминальной. Данные параметры присваиваются к объекту типа BohrVertex. Функция возвращает данный объект

3.

void input_data()

Функция класса Bohr, считывающая текст символ джокера и строку с джокерами и заполняющая поля text, joker и stringwithjoker. Функция ничего не возвращает.

4.

void processing_data()

Функция класса Bohr, которая делит строку с джокерами на строки без них, при этом запоминая их расположение в исходной строке. Функция ничего не возвращает.

5.

void init_bohr()

Функция класса Bohr, добавляющая все подстроки в бор с помощью функции add_string_to_bohr(). Функция ничего не возвращает.

6.

void add_string_to_bohr(std::string str)

Функция класса Bohr, добавляющая символы подстроки в бор. Str – подстрока. Функция ничего не возвращает.

7.

int get_suff_link(int v_num)

Функция класса Bohr, для получения суффиксальной ссылки. V_num – индекс вершины, для которой осуществляется поиск по суффиксальной ссылке. Функция возвращает индекс вершины, доступной по суффиксальной ссылке, в контейнере со всеми вершинами автомата.

8.

int get_auto_move(int v_num, int symb)

Функция класса Bohr, для получения вершины и перехода в неё. V_num – индекс вершины из которой осуществляется переход, symb – символ по которому осуществляется переход.

9.

void Aho()

Функция класса Bohr, для осуществления поиска шаблонов в строке, ничего не возвращает

10.

void check_for_substring(int v_num, int letter_position)

Функция класса Bohr, проверяет наличие шаблона в тексте, заносит результат поиска в result. V_num – номер вершины, letter_position – номер текущего символа в тексте. Функция ничего не возвращает.

11.

void print_result()

Функция класса Bohr, выводит результат работы программы, строку с удалёнными из неё шаблонами, максимальное количество дуг, исходящих из

одной вершины и автомат, построенный вовремя работы программы. Функция ничего не возвращает.

12.

int get_good_suff_link(int v_num)

Функция класса Bohr, для получения хорошей суффиксальной ссылки. V_num – индекс вершины, для которой осуществляется поиск по хорошей суффиксальной ссылке. Функция возвращает индекс вершины, доступной по хорошей суффиксальной ссылке, в контейнере со всеми вершинами автомата.

Тестирование.

1.

Входные данные:

ACTANCA
A$$A$
$

Выходные данные:
ACTANCA
A$$A$
$

String processing started
Symbol A(0) added to string number 1
Substring founded: A, beginning in 0 position
Symbol A(3) added to string number 2
Substring founded: A, beginning in 3 position
String processing ended

Start add string to bohr A, number of string: 0

Add vertex A on position 0
Added new vertex in tree: 1
Transition to vertex by symbol: A

New terminal vertex added: A

Start add string to bohr A, number of string: 1

Add vertex A on position 0
Transition to vertex by symbol: A

New terminal vertex added: A

Search for substrings started
Current symbol: A(0)
Try to find transition from 0 by symbol A
Transition to vertex: 1 ,using symbol: A

Substring founded on position: 0
Count of patterns on this index is 1 of of the necessary 2
Getting good suffix link from vertex: 1
Getting suffix link from vertex: 1
Suffix link leads to root!
Suffix link from vertex 1 is 0

Suffix link leads to root!
Good suffix link from vertex 1 is 0

Current symbol: C(1)
Try to find transition from 1 by symbol C
Getting suffix link from vertex: 1
Suffix link from vertex 1 is 0

Try to find transition from 0 by symbol C
Suffix link leads to root!
Go to vertex: 0 ,using symbol: C
Transition added
Go to vertex: 0 ,using symbol: C
Transition added
Transition to vertex: 0 ,using symbol: C

Current symbol: T(2)
Try to find transition from 0 by symbol T
Suffix link leads to root!
Go to vertex: 0 ,using symbol: T
Transition added
Current symbol: A(3)
Try to find transition from 0 by symbol A
Transition to vertex: 1 ,using symbol: A

Substring founded on position: 0
Count of patterns on this index is 2 of of the necessary 2
Getting good suffix link from vertex: 1
Good suffix link from vertex 1 is 0

Current symbol: N(4)
Try to find transition from 1 by symbol N
Getting suffix link from vertex: 1
Suffix link from vertex 1 is 0

Try to find transition from 0 by symbol N
Suffix link leads to root!
Go to vertex: 0 ,using symbol: N
Transition added
Go to vertex: 0 ,using symbol: N
Transition added

Transition to vertex: 0 ,using symbol: N

Current symbol: C(5)
Try to find transition from 0 by symbol C
Transition to vertex: 0 ,using symbol: C

Current symbol: A(6)
Try to find transition from 0 by symbol A
Transition to vertex: 1 ,using symbol: A

Getting good suffix link from vertex: 1
Good suffix link from vertex 1 is 0

End of search

The resulting array of substrings in the text
A C T A N C A
2 0 0 0 0 0 0
Result of programm:
Search for index equal to count of patterns started
2[0] == 2?
index is 1
0[1] == 2?
0[2] == 2?
0[3] == 2?
0[4] == 2?
0[5] == 2?
0[6] == 2?

Machine:
Ways from vertex 0:
    symbol: A, index: 1
    symbol: C, index: 0
    symbol: N, index: 0
    symbol: T, index: 0
Ways from vertex 1:
    symbol: C, index: 0
    symbol: N, index: 0

Maximum number of outgoing arcs from one vertex of bor is: 1

Removing substrings from text:
Source text: ACTANCA
Edited text: CA

2.

Номер теста	Входные данные	Выходные данные
1	ACTANCAACATAA A $	3 6 9 11
2	CACC XAXCX X	
3	ACCTTTNNNNGGGG $NN $	6 7 8
4	ACTANCACCAACTANCA A$$A$ $	1 4 7 11

Выводы.

В ходе выполнения лабораторной работы были получены навыки работы с алгоритмом Ахо-Корасик и алгоритмом поиска подстроки с “джокером”. Были написаны программы, реализующую эти алгоритмы работы со строками.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

АЛГОРИТМ АХО-КОРАСИК

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


struct BohrVrtx
{
   std::map<char, int> next_v; // контейнер, необходимый для переходов по рёбрам в боре.
   std::map<char, int> auto_movement; //контейнер, необходимый для переходов в автомате
   int number; // номер обработанной подстроки
   int parent; //индекс родителя
   int suff_link = -1; //индекс перехода по суффиксальной ссылке
   int goodsufflink = -1; //индекс перехода по хорошей суффиксальной ссылке
   int deep; // длина подстроки
   bool isEnd; //- логическая переменная, отвечающая за то, является ли вершина терминальной
   char parentsymb; // символ родителя
};

struct Result
{
   int position; //позиция подстроки в тексте
   int number; //номер подстроки
};

bool compare(Result a1, Result a2)
{
   if(a1.position == a2.position)
      return a1.number < a2.number;
   return a1.position < a2.position;
}

class Bohr
{
   std::vector <BohrVrtx> bohrTree; // бор
   std::vector <Result> result;
   std::string text;
   std::vector <std::string> substrings; //  контейнер с подстроками
   std::vector <int> forcutstr; //контейнер для удаления подстрок из текста
   int stringNum; // общее количество подстрок
   int currcount; // текущее количество обработанных подстрок
public:
   std::vector <int> jokerpositions;//контейнер с позициями подстрок в исходной строке
   Bohr():stringNum(0), currcount(0)
   {
      bohrTree.push_back(make_bohr_vert(-1,-1));
   }

   BohrVrtx make_bohr_vert( int parent, char symbol, bool flag)//создание вершины
   {
      BohrVrtx tmp;//инициализация параметров
      tmp.parent = parent;
      tmp.parentsymb = symbol;
      tmp.isEnd = flag;
      return  tmp;
   }

   BohrVrtx make_bohr_vert( int parent, int link)
   {
      BohrVrtx tmp;
      tmp.parent = parent;
      tmp.suff_link = link;
```

```
    tmp.isEnd = false;
    return  tmp;
}

void input_data()//ввод данных необходимых для работы алгоритма
{
    std::cin >> text;
    std::cin >> stringNum;
    std::string str;
    for(int i = 0; i < stringNum; i++)
    {
        std::cin >> str;
        substrings.push_back(str);
    }
    forcutstr.resize(text.size());

}

void init_bohr()
{
    for (int i = 0; i < stringNum; i++)
    {
        add_string_to_bohr(substrings[i]); //добавление строк поочередно в бор
    }
}

void add_string_to_bohr( std::string str)//добавление подстроки в бор
{
    std::cout << "\n-----------------------------------------------------\n";
    std::cout << "Start add string to bohr " << str << ", number of string: " << currcount << '\n';
    int index = 0;
    char currSymb = ' ';
    for (int i = 0; i < str.size(); i++)
    {
        currSymb= str[i];
        std::cout << "\nAdd vertex " << str[i] << " on position " << i << '\n';
        if (bohrTree[index].next_v.find(str[i]) == bohrTree[index].next_v.end())//если переход по символу
отсутсвует, происходит добавление новой вершины
        {
            bohrTree.push_back(make_bohr_vert(index,str[i],false));
            bohrTree[index].next_v[str[i]] = bohrTree.size() - 1;
            std::cout << "Added new vertex in tree: " <<  bohrTree[index].next_v[str[i]] << '\n';
        }
        index = bohrTree[index].next_v[str[i]];
        std::cout << "Transition to vertex by symbol: " << str[i] << "\n";
    }
    if(str.length() > 0)
    {
        bohrTree[index].deep = str.size();//запоминаем длину текущей строки

        bohrTree[index].number =++currcount;// запоминаие номера текущей строки
        bohrTree[index].isEnd = true;//помечаем терминальную вершину
        std::cout << "\nNew terminal vertex added: " << currSymb << "\n";
        std::cout << "-----------------------------------------------------\n";
    }

}

int get_suff_link(int index) //получение суффиксальной ссылки для вершины
{
    std::cout << "Getting suffix link from vertex: " << index <<'\n';
    if (bohrTree[index].suff_link == -1) //если суффиксальная ссылка отсутствует
    {
        if (index == 0 || bohrTree[index].parent == 0)// если вершина корень или родитель вершины корень
        {
            std::cout << "Suffix link leads to root!\n";
            bohrTree[index].suff_link = 0;
```

```
            std::cout << "Suffix link from vertex " << index << " is 0\n\n";
            return bohrTree[index].suff_link;

        }
        bohrTree[index].suff_link = get_auto_movement(get_suff_link(bohrTree[index].parent),
bohrTree[index].parentsymb);
    }
    std::cout << "Suffix link from vertex " << index << " is " << bohrTree[index].suff_link << "\n\n";
    return bohrTree[index].suff_link;
  }

  int get_good_suff_link(int v_num)// получение хорошей суффиксальной ссылки для вершины
    {
      std::cout << "Getting good suffix link from vertex: " << v_num <<'\n';
      if(bohrTree[v_num].goodsufflink == -1)//если хорошая суффиксальная ссылка отсутсвует
      {
        if(get_suff_link(v_num) == 0)//если вершина корень
        {
          bohrTree[v_num].goodsufflink = 0;
          std::cout << "Suffix link leads to root!\n";
        }
        else
        {
          int tmp = get_suff_link(v_num);
          if(bohrTree[tmp].isEnd)//если ближайшая терминальная вершина найдена
          {
            std::cout << "Nearest terminal vertex found: ";
            bohrTree[v_num].goodsufflink = tmp;
            std::cout << "good suffix link from vertex " << v_num << " is " << bohrTree[v_num].goodsufflink <<
"\n\n";
            return bohrTree[v_num].goodsufflink;
          }
          bohrTree[v_num].goodsufflink = get_good_suff_link(tmp);//рекурсинвый поиск
        }
      }
      std::cout << "Good suffix link from vertex " << v_num << " is " << bohrTree[v_num].goodsufflink << "\n\n";
      return bohrTree[v_num].goodsufflink;
    }

  int get_auto_movement(int index, char symb) //функция перехода из вершины по символу
  {
     std::cout << "Try to find transition from " << index << " by symbol " << symb << "\n";
    if(bohrTree[index].auto_movement.find(symb) == bohrTree[index].auto_movement.end()) // если путь в массиве
переходов найден не был
    {
      if (bohrTree[index].next_v.find(symb) != bohrTree[index].next_v.end())// если переход по текущему символу
существует в боре
      {
        bohrTree[index].auto_movement[symb] = bohrTree[index].next_v[symb];// добавляем этот переход
        //std::cout << "Go to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " <<  symb <<
"\n"
            //<< "Transition added\n";
      }
      else
      {
        if(index == 0)
        {
          std::cout << "Suffix link leads to root!\n";
          bohrTree[index].auto_movement[symb] = 0;
          std::cout << "Go to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " <<  symb
<< "\n"
              << "Transition added\n";
          return  bohrTree[index].auto_movement[symb];
        }
        bohrTree[index].auto_movement[symb] =  get_auto_movement(get_suff_link(index), symb);
        std::cout << "Go to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " <<  symb <<
"\n"
```

```
                << "Transition added\n";
            }
        }
        std::cout << "Transition to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " << symb <<
"\n\n";
        return bohrTree[index].auto_movement[symb];
    }

    void Aho()
    {
        std::cout << "\n-----------------------------------------------------\n";
        std::cout << "Search for substrings started\n";
        int curr = 0;
        for (int i = 0; i < text.size(); i++)
        {
            std::cout << "Current symbol: " << text[i] << '(' << i << ")\n";
            curr = get_auto_movement(curr, text[i]);
            check_for_substring(curr,i);

        }
        std::cout << "End of search\n";
        std::cout << "-----------------------------------------------------\n";
    }

    void check_for_substring(int v_num, int letter_position)
    {
        for (int tmp = v_num; tmp != 0; tmp = get_good_suff_link(tmp))
        {
            if (bohrTree[tmp].isEnd)//если вершина терминальная
            {
                Result buff;
                buff.position = letter_position + 2 - bohrTree[tmp].deep;
                buff.number = bohrTree[tmp].number;

                result.push_back(buff);
                std::cout << "The entrance found, index = " <<
                        letter_position + 2 - bohrTree[tmp].deep << " and pattern is " << substrings[bohrTree[tmp].number - 1]
<< "\n";
            }
        }
    }

    void print_result()

    {
        std::sort(result.begin(),result.end(), compare);
        std::cout << "Result of programm:\n";
        std::vector <int> stringforedit(text.size());
        for( int i = 0; i < result.size(); i++)
        {
            std::cout << result[i].position << " " << result[i].number << '\n';
        }
        std::cout << "\nMachine:\n";
        int maxcountoftrans = 0;
        int currcountoftrans;
        for (int i = 0; i < bohrTree.size(); i++)
        {
            auto iter = bohrTree[i].auto_movement.begin();
            std::cout << "Ways from vertex " << i << ":\n";
            currcountoftrans = bohrTree[i].next_v.size();
            for (int j = 0; j < bohrTree[i].auto_movement.size(); j++)//вывод автомата
            {
                std::cout << "\tsymbol: " << iter->first << ", index: " << iter->second << "\n";
                iter++;
            }
            if( currcountoftrans > maxcountoftrans)
                maxcountoftrans = currcountoftrans;
```

```
        }
        std::cout << "\nMaximum number of outgoing arcs from one vertex of bor is:  " << maxcountoftrans << '\n';
        std::cout << "\nRemoving substrings from text:\n";
        std::string editedtext = "";

        for(int i = 0; i < result.size(); i++)
        {
           for(int j = 0; j < substrings[result[i].number - 1].length(); j++)//отмечатеся вхождение подстроки в текст
           {
              forcutstr[result[i].position +j -1]++;
           }
        }
        for(int i = 0; i < forcutstr.size(); i++)
        {
           if(forcutstr[i] == 0)
              editedtext += text[i];
        }
        std::cout << "Source text: " << text << '\n';
        std::cout << "Edited text: " << editedtext << '\n';
    }



};


int main()
{

    Bohr* tmp = new Bohr();
    tmp->input_data();
    tmp->init_bohr();
    tmp->Aho();
    tmp->print_result();

    return 0;
}

/*
CCCA
1
CC

AACCGGTTNN
4
ACC
GGT
TT
CC

CACTANCA
A$$A$
$

CACTANCAAGAC
A$A$
$

ACTANCAACATAA
$A$
$

ACTANCA
A$$A$
$
```

*/

АЛГОРИТМ ПОИСКА ШАБЛОНА С “ДЖОКЕРОМ”

```
#include <iostream>
#include <string>
#include <vector>
#include <map>


struct BohrVrtx
{
   std::map<char, int> next_v; // массив, необходимый для переходов по рёбрам в боре.
   std::map<char, int> auto_movement; //массив, необходимый для переходов в автомате
   std::vector<int> number; // номер обработанной подстроки
   int parent; //индекс родителя
   int suff_link = -1; //индекс перехода по суффиксальной ссылке
   int goodsufflink = -1; //индекс перехода по хорошей суффиксальной ссылке
   int deep; // длина подстроки
   bool isEnd; //- логическая переменная, отвечающая за то, является ли вершина терминальной
   char parentsymb; // символ родителя
};

class Bohr
{
   std::vector <BohrVrtx> bohrTree; // бор
   std::string text;
   std::string stringwithjoker;
   std::vector <int> match; //вектор для запоминания вхождения подстрок в текст
   std::vector <std::string> substrings; //  контейнер с подстроками
   int stringNum; // общее количество подстрок
   int currcount; // текущее количество обработанных подстрок
   char joker;
public:
   std::vector <int> jokerpositions;//контейнер с позициями подстрок в исходной строке
   Bohr():stringNum(0), currcount(0)
   {
      bohrTree.push_back(make_bohr_vert(-1,-1));
   }

   BohrVrtx make_bohr_vert( int parent, char symbol, bool flag)//создание вершины
   {
      BohrVrtx tmp;//инициализация параметров
      tmp.parent = parent;
      tmp.parentsymb = symbol;
      tmp.isEnd = flag;
      return  tmp;
   }

   BohrVrtx make_bohr_vert( int parent, int link)
   {
      BohrVrtx tmp;
      tmp.parent = parent;
      tmp.suff_link = link;
      tmp.isEnd = false;
      return  tmp;
   }

   void input_data()//ввод данных необходимых для работы алгоритма
   {
      std::cin >> text;
      std::cin >> stringwithjoker;
      std::cin >> joker;
      match.resize(text.size());
   }
```

```
void init_bohr()
{
   for (int i = 0; i < stringNum; i++)
   {
      add_string_to_bohr(substrings[i]); //добавление строк поочередно в бор
   }
}

void processing_data()//обработка строки с джокерами
{
   std::cout << "\n-----------------------------------------------------\n";
   std::cout << "String processing started\n";
   std::string tmp = "";
   for (int i = 0; i < stringwithjoker.size(); i++)
   {
      if (stringwithjoker[i] == joker)//если текущий символ строки с джокерами - джокер
      {
         if (tmp.size() > 0) //исключение обработки пустых строк
         {
            substrings.push_back(tmp); //запоминание подстроки
            jokerpositions.push_back(i - tmp.size()); //запоминание позиции подстроки в строке с джокерами
            stringNum++;
            std::cout << "Substring founded: " << tmp << ", beginning in "
                  << jokerpositions.back() << " position\n";
            tmp = "";

         }
      }
      else
      {
         std::cout << "Symbol " << stringwithjoker[i] << '(' << i
               << ") added to string number " << stringNum + 1
               << '\n';
         tmp.push_back(stringwithjoker[i]);
         if (i == stringwithjoker.size() - 1)// случай, когда последний символ строки с джокерами
         {
            substrings.push_back(tmp); //запоминание подстроки
            jokerpositions.push_back(i - tmp.size() + 1); //запоминание позиции подстроки в строке с джокерами
            stringNum++;
            std::cout << "Substring founded: " << tmp << ", beginning in "
                  << jokerpositions.back() << " position\n";
         }
      }
   }
   std::cout << "String processing ended\n";
   std::cout << "-----------------------------------------------------\n";
}

void add_string_to_bohr( std::string str)//добавление подстроки в бор
{
   std::cout << "\n-----------------------------------------------------\n";
   std::cout << "Start add string to bohr " << str << ", number of string: " << currcount << '\n';
   int index = 0;
   char currSymb = ' ';
   for (int i = 0; i < str.size(); i++)
   {
      currSymb= str[i];
      std::cout << "\nAdd vertex " << str[i] << " on position " << i << '\n';
      if (bohrTree[index].next_v.find(str[i]) == bohrTree[index].next_v.end())//если переход по символу
отсутсвует, происходит добавление новой вершины
      {
         bohrTree.push_back(make_bohr_vert(index,str[i],false));
         bohrTree[index].next_v[str[i]] = bohrTree.size() - 1;
         std::cout << "Added new vertex in tree: " <<  bohrTree[index].next_v[str[i]] << '\n';
      }
      index = bohrTree[index].next_v[str[i]];
      std::cout << "Transition to vertex by symbol: " << str[i] << "\n";
```

```
        }
        if(str.length() > 0)
        {
            bohrTree[index].deep = str.size();//запоминаем длину текущей строки

            bohrTree[index].number.push_back(++currcount);// запоминаие номера текущей строки
            bohrTree[index].isEnd = true;//помечаем терминальную вершину
            std::cout << "\nNew terminal vertex added: " << currSymb << "\n";
            std::cout << "------------------------------------------------------\n";
        }

    }

    int get_suff_link(int index) //получение суффиксальной ссылки для вершины
    {
        std::cout << "Getting suffix link from vertex: " << index <<'\n';
        if (bohrTree[index].suff_link == -1) //если суффиксальная ссылка отсутствует
        {
            if (index == 0 || bohrTree[index].parent == 0)// если вершина корень или родитель вершины корень
            {
                std::cout << "Suffix link leads to root!\n";
                bohrTree[index].suff_link = 0;
                std::cout << "Suffix link from vertex " << index << " is 0\n\n";
                return bohrTree[index].suff_link;

            }
            bohrTree[index].suff_link = get_auto_movement(get_suff_link(bohrTree[index].parent),
bohrTree[index].parentsymb);
        }
        std::cout << "Suffix link from vertex " << index << " is " << bohrTree[index].suff_link << "\n\n";
        return bohrTree[index].suff_link;
    }

    int get_good_suff_link(int v_num)// получение хорошей суффиксальной ссылки для вершины
        {
            std::cout << "Getting good suffix link from vertex: " << v_num <<'\n';
            if(bohrTree[v_num].goodsufflink == -1)//если хорошая суффиксальная ссылка отсутсвует
            {
                if(get_suff_link(v_num) == 0)//если вершина корень
                {
                    bohrTree[v_num].goodsufflink = 0;
                    std::cout << "Suffix link leads to root!\n";
                }
                else
                {
                    int tmp = get_suff_link(v_num);
                    if(bohrTree[tmp].isEnd)//если ближайшая терминальная вершина найдена
                    {
                        std::cout << "Nearest terminal vertex found: ";
                        bohrTree[v_num].goodsufflink = tmp;
                        std::cout << "good suffix link from vertex " << v_num << " is " << bohrTree[v_num].goodsufflink <<
"\n\n";
                        return bohrTree[v_num].goodsufflink;
                    }
                    bohrTree[v_num].goodsufflink = get_good_suff_link(tmp);//рекурсивный поиск
                }
            }
            std::cout << "Good suffix link from vertex " << v_num << " is " << bohrTree[v_num].goodsufflink << "\n\n";
            return bohrTree[v_num].goodsufflink;
        }

    int get_auto_movement(int index, char symb) //функция перехода из вершины по символу
    {
         std::cout << "Try to find transition from " << index << " by symbol " << symb << "\n";
        if(bohrTree[index].auto_movement.find(symb) == bohrTree[index].auto_movement.end()) // если путь в массиве
переходов найден не был
        {
```

```
        if (bohrTree[index].next_v.find(symb) != bohrTree[index].next_v.end())// если переход по текущему символу существует в боре
        {
           bohrTree[index].auto_movement[symb] = bohrTree[index].next_v[symb];// добавляем этот переход
           //std::cout << "Go to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " <<  symb << "\n"
                 //<< "Transition added\n";
        }
        else
        {
           if(index == 0)
           {
              std::cout << "Suffix link leads to root!\n";
              bohrTree[index].auto_movement[symb] = 0;
              std::cout << "Go to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " <<  symb << "\n"
                    << "Transition added\n";
              return  bohrTree[index].auto_movement[symb];
           }
           bohrTree[index].auto_movement[symb] =  get_auto_movement(get_suff_link(index), symb);
           std::cout << "Go to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " <<  symb << "\n"
                 << "Transition added\n";
        }
     }
     std::cout << "Transition to vertex: " << bohrTree[index].auto_movement[symb] << " ,using symbol: " << symb << "\n\n";
     return bohrTree[index].auto_movement[symb];
  }

  void Aho()
  {
     std::cout << "\n-----------------------------------------------------\n";
     std::cout << "Search for substrings started\n";
     int curr = 0;
     for (int i = 0; i < text.size(); i++)
     {
        std::cout << "Current symbol: " << text[i] << '(' << i << ")\n";
        curr = get_auto_movement(curr, text[i]);
        check_for_substring(curr,i);

     }
     std::cout << "End of search\n";
     std::cout << "-----------------------------------------------------\n";
  }

  void check_for_substring(int v_num, int letter_position)
  {
     for (int tmp = v_num; tmp != 0; tmp = get_good_suff_link(tmp))
     {
        if (bohrTree[tmp].isEnd)//если вершина терминальная
        {
           for (int j = 0; j < bohrTree[tmp].number.size(); j++)
           {
              if (letter_position + 1 - jokerpositions[bohrTree[tmp].number[j] - 1] - bohrTree[tmp].deep >= 0 &&
                    letter_position + 1 - jokerpositions[bohrTree[tmp].number[j] - 1] - bohrTree[tmp].deep <= text.size() - stringwithjoker.length())
              {
                    match[letter_position + 1 - jokerpositions[bohrTree[tmp].number[j] - 1] - bohrTree[tmp].deep]++;
                    std::cout << "Substring founded on position: " << letter_position + 1 - jokerpositions[bohrTree[tmp].number[j] - 1] - bohrTree[tmp].deep << '\n';
                    std::cout << "Count of patterns on this index is " <<match[letter_position + 1 - jokerpositions[bohrTree[tmp].number[j] - 1] - bohrTree[tmp].deep];
                    std::cout << " of of the necessary " << stringNum << '\n';
              }
           }
        }
```

```
        }
    }

    void print_result()
    {
        std::cout << "The resulting array of substrings in the text\n";
        std::vector <int> stringforedit(text.size());
        for (int i = 0; i < text.size(); i++)
        {
            std::cout << text[i] << " ";
        }
        std::cout << '\n';
        for (int i = 0; i < match.size(); i++)
        {
            std::cout << match[i] << " ";
        }
        std::cout << "\nResult of programm:\nSearch for index equal to number of templates started\n";
        for (int i = 0; i < match.size(); i++)
        {
            std::cout << match[i] << "[" << i << "] == " << stringNum <<"?\n";
            if(match[i] == stringNum)
            {
                std::cout<< "index is " <<i + 1 << "\n";
                for(int j = 0; j < stringwithjoker.length(); j++)//помечаем все символы строки в тексте
                    stringforedit[i + j]++;
            }
        }
        std::cout << "\nMachine:\n";
        int maxcountoftrans = 0;
        int currcountoftrans;
        for (int i = 0; i < bohrTree.size(); i++)
        {
            auto iter = bohrTree[i].auto_movement.begin();
            std::cout << "Ways from vertex " << i << ":\n";
            currcountoftrans = bohrTree[i].next_v.size();
            for (int j = 0; j < bohrTree[i].auto_movement.size(); j++)//вывод автомата
            {
                std::cout << "\tsymbol: " << iter->first << ", index: " << iter->second << "\n";
                iter++;
            }
            if( currcountoftrans > maxcountoftrans)
                maxcountoftrans = currcountoftrans;

        }
        std::cout << "\nMaximum number of outgoing arcs from one vertex of bor is:  " << maxcountoftrans << '\n';
        std::cout << "\nRemoving substrings from text:\n";
        std::string editedtext = "";

        for(int i = 0; i < stringforedit.size(); i++)
        {
            if(stringforedit[i] == 0)//формируем обрезанную строку
                editedtext += text[i];
        }
        std::cout << "Source text: " << text << '\n';
        std::cout << "Edited text: " << editedtext << '\n';

    }



};


int main()
{

    Bohr* tmp = new Bohr();
```

```
    tmp->input_data();
    tmp->processing_data();
    if(tmp->jokerpositions.size() == 0)
    {
        std::cout << "A string cannot consist only of jokers!\n";
        return 0;
    }
    tmp->init_bohr();
    tmp->Aho();
    tmp->print_result();

    return 0;
}

/*
CCCA
1
CC

AACCGGTTNN
4
ACC
GGT
TT
CC

CACTANCA
A$$A$
$

CACTANCAAGAC
A$A$
$

ACTANCAACATAA
$A$
$

ACTANCA
A$$A$
$

 */
```