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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 0382 _____ Корсунов А.А.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2022

Цель работы.

Применить на практике знания о построение алгоритма Ахо-Корасик. Реализовать алгоритм Ахо-Корасик для поиска набора образцов в тексте и шаблонной подстроки.

Задание.

1) 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.

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

неопределённой 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,$1 \leq |T| \leq 100000$)

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.

Ход работы.

1. Был произведен анализ задания.

2. Был реализован алгоритм Ахо-Корасика:

1) Алгоритм для первого задания:

Объявляется алфавит входящих строк, где каждому символу соответствует свой уникальный «код» (номер от 0 до N-1, где N – количество символов в алфавите). Считываются входные данные согласно заданию. Для шаблонов строится Бор. Бор реализован в виде двух классов — Bor_Avto и Vertex, в первом объявлено поле Bor_vector, являющийся вектором, элементы которого являются объекты второго класса. Второй класс представляет из себя «вершину» в Боре, каждая такая вершина обладает следующими данными: информацию о вершине-родителе, информацию о символе на ребре, соединяющих вершину-родителя с этой вершиной, массив из 5 возможных детей (всего у каждой вершины может быть до 5 детей исходя из алфавита), индексы этого массива отвечают за каждый символ, который есть в алфавите

(например, первая ячейка отвечает за вершину, к которой есть ребро через символ <A>, а последняя за вершину, к которой есть ребро через символ <N>, т. к. алфавит имеет следующее соответствие: { {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4} }, в самих же этих ячейках хранится либо «-1» (если такой вершины нет), либо индекс в массиве всех вершин (если такая вершина есть), номер шаблона по счету, в который входит ребро, которое ведет к этой вершине, суффиксная ссылка: индекс в массиве всех вершин, если ссылка уже определена, «-1», если еще не определена, массив, в котором хранятся индексы вершин, в которые из данной вершины есть пусть исходя из суффиксных ссылок или потомков (если они были найдены), если в яейке «-1», то пути нет. После того как Бор был построен, запускается поиск шаблонов в тексте: поочередно считывается символ из текста, с помощью этого символа ищется вершина, которой ведет ребро с этим символом (если оно есть). Если такая вершина находится, то в Боре происходит переход в нее, на каждом шаге итерации у каждой вершины проверяется наличие пути (отмечен ли путь из текущей вершины в следующую доступную (на основе ребра с переданным символом) в массиве путей way, если отмечен — происходит переход по нему, если не отмечен, то проверяется, есть потомок с таким ребром у текущей вершины, если есть — отмечается в way и переходит по нему, если нет, то проверяется суффиксная ссылка у родителя текущей вершины, после чего ищется путь у вершины, к которой произошел переход через суффиксную ссылку с помощью ребра с переданным символом (в случае, если пути у него тоже нет, или ранее у родителя тоже не была отмечена суффиксная ссылка, то для них производится такой же поиск суффиксной ссылки (он будет происходит до тех пор, пока не найдется хоть у какого-то из родителей суффиксная ссылка), после чего суффиксная ссылка отмечается и происходит переход по ней). На этой же итерации после обработки каждого символа, проверяется вершина, в которой алгоритм остановился в конце итерации, проверяется она и ее суффиксная ссылка (а так же суффиксная ссылка суффиксной ссылки вплоть до пустой вершины (кроме нее самой)) на

терминальность, если такие вершины находятся, то они записываются. В самом конце с помощью этих вершин строится соответствие согласно заданию и выводится.

2) Алгоритм для второго задания:

Так же строится Бор + суффинксные ссылки (можно сказать, автомат), но уже из образцов, полученных выделением максимальных безджокерных подстрок из шаблонной подстроки, а не n переданных строк. Для каждого такого образца записывается смещение по которому образец находится в шаблонной строке. После чего выполняется поиск по тексту с использованием ранее созданного автомата, где при каждом обнаружении образца, ячейка предварительно созданного массива (заполненного нулями и длиной равной длины текста) инкрементируется по адресу, образованному разностью номера начального символа образца в тексте и смещения образца. Если у образца несколько смещений, то будет проходить инкрементация всех соответствующих ячеек массива.

3) Структура данных, которая представляет строки — string, Бор — собственных класс, базирующийся на структуре vector, алфавит — map.

4) Сложность алгоритмов по времени — O(ns+T+k), n – суммарная длина всех образцов (или же слов в словаре), s - размер алфавита, T – длина текста, в котором производится поиск, k - общая длина всех совпадений. В силу того, что во-первых, надо построить бор (ns), во-вторых, нужно пройти по всему тексту (T), в-третьих, в боре необходимо проходить по всем совпадениям (k)

Сложность алгоритмов по памяти — O(ns), т. к. нужно хранить Бор в виде индексного массива.

5) Функции и структуры данных:

*class Vertex - класс вершин

*int parent - родитель этой вершины

*char edge_symbol - символ на ребре от родителя до этой вершины

*int child[5] - массив, в котором хранятся для текущей вершины индексы потомков (если они есть), если в ячейке "-1" - такого потомка у вершины нет (значения потомка соотвует позиции в массиве согласно алфавиту в main)

*int I_of_p_i = 0 - номер шаблона, в который входит эта вершина

*bool flag = false - флаг терминальной вершины

*int suffix_link = -1 - суффиксная ссылка на текущую вершину

*int way[5] - массив, в котором хранятся индексы вершин, в которые есть путь исходя из суффиксных ссылок или потомков (если они были найдены), если в ячейке "-1", то пути нет

*class Bor_Avto

*std::vector<Vertex> Bor_vector - вектор с бором + суффиксные ссылки

*int Get_next_vertex(int, int) - получения индекса следущей вершины бора при поиске шаблонов в тексте

*int Get_suffix_link(int) - получение индекса следующей вершины через суффиксную ссылку

*void Result(std::vector <std::string>&, int, int, std::string, std::vector<std::pair<int, int>>&) - формирует результат поиска шаблона согласно заданию

*void Add_p_i_vertexes(std::map<char, int>, int, std::string) - добавление шаблонов в бор

*void Find_p_i_in_text(std::vector <std::string>, std::string, std::map<char, int>, std::vector<std::pair<int, int>>&) - основной метод поиска шаблонов в тексте

*std::vector <int> Templates(std::map<char, int>, std::stringstream&, char) — сопоставляет символы алфавита и его значения, строку и символ джокера, для разделения строки с джокером на подстроки для бора

*void Result_print(const std::vector<int>&, int, int) — проверяет совпадение

и выводит результат

б) Тестирование (1 программа):

а) Входные данные:

NTAG

3

TAGT

TAG

T

Выходные данные:

2 2

2 3

б) Входные данные:

NTAGAGA

4

TAGT

TAGA

GAGA

TAGG

Выходные данные:

2 2

4 3

в) Входные данные:

CAT

3

C

A

T

Выходные данные:

1 1

2 2

3 3

Рисунок 1 - Пример работы первой программы

7) Тестирование (2 программа):

а) Входные данные:

ACTANCA

A$$A$

$

Выходные данные:

1

б) Входные данные:

ACATN

CT

$

Выходные данные:

1

в) Входные данные:

CAT

A

$

Выходные данные:

1

Рисунок 2 - Пример работы второй программы

Выводы.

Были применины на практике знания о построение алгоритма Ахо-Корасик. Был реализовать алгоритм Ахо-Корасик для поиска набора образцов в тексте и шаблонной подстроки.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл KORASIK_TASK_1.cpp:

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

class Vertex //класс вершин
{
public:
	int parent; //родитель этой вершины
	char edge_symbol; //символ на ребре от родителя до этой вершины
	int child[5]; //массив, в котором хранятся для текущей вершины индексы потомков (если они есть), если в ячейке "-1" - такого потомка у вершины нет (значения потомка соотвует позиции в массиве согласно алфавиту в main)
	int I_of_p_i = 0; //номер шаблона, в который входит эта вершина
	bool flag = false; //флаг терминальной вершины
	int suffix_link = -1; //суффиксная ссылка на текущую вершину
	int way[5]; //массив, в котором хранятся индексы вершин, в которые есть путь исходя из суффиксных ссылок или потомков (если они были найдены), если в ячейке "-1", то пути нет
	Vertex(int parent, char symbol) : parent(parent), edge_symbol(symbol)
	{
		for (int i = 0; i < 5; i++)
		{
			child[i] = -1;
			way[i] = -1;
		}
	}
	Vertex() = default;
	~Vertex() = default;
};

class Bor_Avto
{
	std::vector<Vertex> Bor_vector; //вектор с бором + суффиксные ссылки
	int Get_next_vertex(int, int); //получения индекса следущей вершины бора при поиске шаблонов в тексте
	int Get_suffix_link(int); //получение индекса следующей вершины через
```

```
суффиксную ссылку
	void Result(std::vector <std::string>&, int, int, std::string, std::vector<std::pair<int, int>>&); //формирует результат поиска шаблона согласно заданию

public:
	Bor_Avto()
	{
		Bor_vector.push_back(Vertex(0, 0));
	}
	~Bor_Avto() = default;

	void Add_p_i_vertexes(std::map<char, int>, int, std::string); //добавление шаблонов в бор
	void Find_p_i_in_text(std::vector <std::string>, std::string, std::map<char, int>, std::vector<std::pair<int, int>>&);//основной метод поиска шаблонов в тексте
};

void Bor_Avto::Add_p_i_vertexes(std::map<char, int> alphabet, int index, std::string p_i)
{
	int n = 0; //номер вершины в боре

	for (int i = 0; i < p_i.size(); i++)
	{
		char symbol = p_i[i]; //в symbol кладется i-ый символ из переданной строки
		int s_edge = alphabet[symbol]; //в s_edge кладется номер этого символа согласно алфавиту
		if (Bor_vector[n].child[s_edge] == -1) //если у текущей вершины бора нет потомка с номер s_edge (s_edge соотвествует символу алфавита), то
		{
			Bor_vector.push_back(Vertex(n, s_edge)); //кладем в бор-вектор вершину с родителем 'n' (текущая вершина) и символом, которых их соединяет - s_edge
			Bor_vector[n].child[s_edge] = Bor_vector.size() - 1; //в массиве потомков для текущего вектора отмечается, что потомок стоит на <текущая длина бор-вектора - 1> месте в бор-векторе
		}
		n = Bor_vector[n].child[s_edge]; //в n кладется индекс потомка в бор-векторе
	}
```

```
	Bor_vector[n].flag = true; //последняя вершина (к которой ведет ребро с последний символом из переданной строки p_i) помечается терминальной
	Bor_vector[n].I_of_p_i = index; //эта же вершина теперь указывает на номер i-го шаблона (номер по счету переданной строки)
}

int Bor_Avto::Get_suffix_link(int vertex_index)
{
	if (Bor_vector[vertex_index].suffix_link == -1) //если суффикусной ссылки нет
	{
		if (vertex_index == 0 || Bor_vector[vertex_index].parent == 0) //если это пустая вершина или ее потомки
		{
			Bor_vector[vertex_index].suffix_link = 0; //отмечаем суффиксную ссылку дли них (на пустую вершину)
		}
		else //если это не пустая вершина
		{
			Bor_vector[vertex_index].suffix_link = Get_next_vertex(Get_suffix_link(Bor_vector[vertex_index].parent), Bor_vector[vertex_index].edge_symbol); //поиск суффиксной ссылки через предка
		} //этот метод будет запускаться для вершин, у которых родитель не пустой, для пустого родителя будет запускаться только get_suffix_link
	}
	return Bor_vector[vertex_index].suffix_link;
}

int Bor_Avto::Get_next_vertex(int vertex_index, int symbol_on_edge)
{
	if (Bor_vector[vertex_index].way[symbol_on_edge] == -1) //если путь для переданной вершины до вершины с ребром текущего символа не отмечен
	{
		if (Bor_vector[vertex_index].child[symbol_on_edge] != -1) //если для переданной вершины есть вершина-ребенок с текущим символом
		{
			Bor_vector[vertex_index].way[symbol_on_edge] = Bor_vector[vertex_index].child[symbol_on_edge]; //отметка пути (для текущей вершины отмечается путь до вершины-ребенка с ребром текущего символа)
		}
		else //если для переданной вершины нет вершины-ребенока с текущим символом
```

```
		{
			if (vertex_index == 0) //если это пустая вершина (корень)
			{
				Bor_vector[vertex_index].way[symbol_on_edge] = 0; //отметка пути (для текущей вершины отмечается, что пути до вершины с ребром текущего символа в боре нет, т.е. из пустой вершины тогда идет путь до нее самой же)
			}
			else //если это не пустая вершина (не корень)
			{
				Bor_vector[vertex_index].way[symbol_on_edge] = Get_next_vertex(Get_suffix_link(vertex_index), symbol_on_edge);
			}
		}
	}
	return Bor_vector[vertex_index].way[symbol_on_edge];
}

void Bor_Avto::Result(std::vector <std::string>& vector_P_I_STR, int vertex_index, int count_text, std::string text, std::vector<std::pair<int, int>> &output)
{
	for (int i = vertex_index; i != 0; i = Get_suffix_link(i)) //проход от текущей вершины по всем суффикс-ссылкам
	{
		if (Bor_vector[i].flag) //если текущая вершина терминальная
		{
			output.push_back(std::make_pair(count_text - vector_P_I_STR[Bor_vector[i].I_of_p_i].size() + 1, Bor_vector[i].I_of_p_i + 1));//записать ее
		}
	}
}

void Bor_Avto::Find_p_i_in_text(std::vector<std::string> vector_P_I_STR, std::string text, std::map<char, int> alphabet, std::vector<std::pair<int, int>> &output)
{
	int symbol_number = 0;

	for (int i = 0; i < text.size(); i++)
	{
		char symbol = text[i]; //i-ый символ из переданного текста
		int s_edge = alphabet[symbol]; //номер этого символа согласно алфавиту
```

```
            symbol_number = Get_next_vertex(symbol_number, s_edge);
            Result(vector_P_I_STR, symbol_number, i + 1, text, output);
        }
}

int main()
{
        std::string text; // текст
        int N; // количество шаблонов
        std::string P_I; // i-ый шаблон
        std::vector<std::string> vector_P_I; // вектор шаблонов
        std::vector<std::pair<int, int>> output; //вектор для вывода

        std::map<char, int> alphabet{ {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4} };
//объявляется алфавит (символы, которые встречаются в переданных тексте
и шаблонах, каждому из них ставится в соответсвие определенный
уникальный номер)
        Bor_Avto Bor;

        std::cin >> text >> N;

        for (int i = 0; i < N; i++) //создание бора и заполнение vector_P_I
переданными шаблонами
        {
            std::cin >> P_I;
            Bor.Add_p_i_vertexes(alphabet, i, P_I);
            vector_P_I.push_back(P_I);
        }

        Bor.Find_p_i_in_text(vector_P_I, text, alphabet, output);

        sort(output.begin(), output.end()); //сортировка ouput для вывода согласно
заданию

        for (int i = 0; i < output.size(); i++)
        {
            std::cout << output[i].first << ' ' << output[i].second << '\n';
        }

        return 0;
}
```

Файл KORASIK_TASK_2.cpp:

```
#include <iostream>
#include <vector>
#include <cstring>
#include <map>
#include <algorithm>
#include <sstream>

class Vertex //класс вершин
{
public:
	int parent; //родитель этой вершины
	char edge_symbol; //символ на ребре от родителя до этой вершины
	int child[5]; //массив, в котором хранятся для текущей вершины индексы потомков (если они есть), если в ячейке "-1" - такого потомка у вершины нет (значения потомка соотвует позиции в массиве согласно алфавиту в main)
	std::vector<int> str_number; //вектор с позициями строк
	bool flag = false; //флаг терминальной вершины
	int suffix_link = -1; //суффинксная ссылка на текущую вершину
	int way[5]; //массив, в котором хранятся индексы вершин, в которые есть путь исходя из суффиксных ссылок или потомков (если они были найдены), если в ячейке "-1", то пути нет
	Vertex(int parent, char symbol) : parent(parent), edge_symbol(symbol)
	{
		for (int i = 0; i < 5; i++)
		{
			child[i] = -1;
			way[i] = -1;
		}
		str_number.resize(0);
	}
	Vertex() = default;
	~Vertex() = default;
};

class Bor_Avto
{
	std::vector<Vertex> Bor_vector; //вектор с бором + суффиксные ссылки
	int Get_next_vertex(int, int); //получения индекса следущей вершины бора при поиске шаблонов в тексте
	int Get_suffix_link(int); //получение индекса следующей вершины через суффиксную ссылку
	void Result(int, int, std::vector <int>&, std::vector<int>);
	std::vector<std::string> template_strings; //вектор строк
```

```
    void Add_p_i_vertexes(std::map<char, int>, std::string); //добавление строки в бор

public:
    Bor_Avto()
    {
        Bor_vector.push_back(Vertex(0, 0));
    }
    ~Bor_Avto() = default;

    void Find_p_i_in_text(std::map<char, int>, std::string&, std::vector <int>&, const std::vector <int>&);
    std::vector <int> Templates(std::map<char, int>, std::stringstream&, char);
    void Result_print(const std::vector<int>&, int, int);
};

void Bor_Avto::Add_p_i_vertexes(std::map<char, int> alphabet, std::string str)
{
    int n = 0; //номер вершины в боре

    for (int i = 0; i < str.size(); i++) {
        char symbol = str[i]; //в symbol кладется i-ый символ из переданной строки
        int s_edge = alphabet[symbol]; //в s_edge кладется номер этого символа согласно алфавиту
        if (Bor_vector[n].child[s_edge] == -1) //если у текущей вершины бора нет потомка с номер s_edge (s_edge соотвествует символу алфавита), то
        {
            Bor_vector.push_back(Vertex(n, s_edge)); //кладем в бор-вектор вершину с родителем 'n' (текущая вершина) и символом, которых их соединяет - s_edge
            Bor_vector[n].child[s_edge] = Bor_vector.size() - 1; //в массиве потомков для текущего вектора отмечается, что потомок стоит на <текущая длина бор-вектора - 1> месте в бор-векторе
        }
        n = Bor_vector[n].child[s_edge]; //в n кладется индекс потомка в бор-векторе
    }
    Bor_vector[n].flag = true; //последняя вершина (к которой ведет ребро с последний символом из переданной строки p_i) помечается терминальной
    template_strings.push_back(str); //добавление этой строки в вектор строк
    Bor_vector[n].str_number.push_back(template_strings.size() - 1); //отметка этой строки в боре
```

```
}

int Bor_Avto::Get_suffix_link(int vertex_index)
{
	if (Bor_vector[vertex_index].suffix_link == -1) //если суффикусной ссылки нет
	{
		if (vertex_index == 0 || Bor_vector[vertex_index].parent == 0) //если это пустая вершина или ее потомки
		{
			Bor_vector[vertex_index].suffix_link = 0; //отмечаем суффиксную ссылку дли них (на пустую вершину)
		}
		else //если это не пустая вершина
		{
			Bor_vector[vertex_index].suffix_link = Get_next_vertex(Get_suffix_link(Bor_vector[vertex_index].parent), Bor_vector[vertex_index].edge_symbol); //поиск суффиксной ссылки через предка
		} //этот метод будет запускаться для вершин, у которых родитель не пустой, для пустого родителя будет запускаться только get_suffix_link
	}
	return Bor_vector[vertex_index].suffix_link;
}

int Bor_Avto::Get_next_vertex(int vertex_index, int symbol_on_edge)
{
	if (Bor_vector[vertex_index].way[symbol_on_edge] == -1) //если путь для переданной вершины до вершины с ребром текущего символа не отмечен
	{
		if (Bor_vector[vertex_index].child[symbol_on_edge] != -1) //если для переданной вершины есть вершина-ребенок с текущим символом
		{
			Bor_vector[vertex_index].way[symbol_on_edge] = Bor_vector[vertex_index].child[symbol_on_edge]; //отметка пути (для текущей вершины отмечается путь до вершины-ребенка с ребром текущего символа)
		}
		else //если для переданной вершины нет вершины-ребенока с текущим символом
		{
			if (vertex_index == 0) //если это пустая вершина (корень)
			{
				Bor_vector[vertex_index].way[symbol_on_edge] = 0;
```

```
//отметка пути (для текущей вершины отмечается, что пути до вершины с ребром текущего символа в боре нет, т.е. из пустой вершины тогда идет путь до нее самой же)
                        }
                        else //если это не пустая вершина (не корень)
                        {
                                Bor_vector[vertex_index].way[symbol_on_edge] = Get_next_vertex(Get_suffix_link(vertex_index), symbol_on_edge);
                        }
                }
        }
        return Bor_vector[vertex_index].way[symbol_on_edge];
}

void Bor_Avto::Result(int vertex_index, int i, std::vector<int>& additional_array, std::vector<int> templates_length)
{
        for (int u = vertex_index; u != 0; u = Get_suffix_link(u)) //проход от текущей вершины по всем суффикс-ссылкам
        {
                if (Bor_vector[u].flag) //если текущая вершина терминальная
                {
                        for (const auto& j : Bor_vector[u].str_number)
                        {
                                if ((i - templates_length[j] < additional_array.size()))
                                {
                                        additional_array[i - templates_length[j]]++;
                                }
                        }
                }
        }
}

std::vector<int> Bor_Avto::Templates(std::map<char, int> alphabet, std::stringstream& template_string, char joker)
{
        std::vector<int> templates_length;
        int length = 0;
        std::string storage;
        while (getline(template_string, storage, joker))
        {
                if (storage.size() > 0)
                {
                        length += storage.size();
```

```
                templates_length.push_back(length);
                Add_p_i_vertexes(alphabet, storage);
            }
            length++;
        }
        return templates_length;
}

void Bor_Avto::Result_print(const std::vector<int>& additional_array, int text_size,
int length)
{
        for (int i = 0; i < text_size; i++)
        {
            if ((additional_array[i] == template_strings.size()) && (i + length <=
text_size))
            {
                std::cout << i + 1 << "\n";
            }
        }
}

void Bor_Avto::Find_p_i_in_text(std::map<char, int> alphabet, std::string& text,
std::vector <int>& additional_array, const std::vector <int>& templates_length)
{
        int u = 0;
        int lenght = text.length();

        for (int i = 0; i < lenght; i++)
        {
            char symbol = text[i]; //i-ый символ из переданного текста
            int s_edge = alphabet[symbol]; //номер этого символа согласно
алфавиту
            u = Get_next_vertex(u, s_edge);
            Result(u, i + 1, additional_array, templates_length);
        }
}

int main()
{
        std::string text; //текст
        std::string P; //шаблон
        char joker; //джокер

        std::map<char, int> alphabet{ {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N', 4} };
```

```
//объявляется алфавит (символы, которые встречаются в переданных тексте
и шаблонах, каждому из них ставится в соответсвие определенный
уникальный номер)
        Bor_Avto Bor;

        std::cin >> text >> P >> joker;

        std::stringstream str_stream(P);
        std::vector<int> templates_length = Bor.Templates(alphabet, str_stream,
joker);
        std::vector<int> additional_array(text.size(), 0);

        Bor.Find_p_i_in_text(alphabet, text, additional_array, templates_length);
        Bor.Result_print(additional_array, text.size(), P.size());

        return 0;
}
```