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»

Студентка гр.7304 _____ Каляева А.В.

Преподаватель _____ Филатов А.Ю

г. Санкт-Петербург

г. 2019

Цель работы:

Изучить алгоритм Ахо-Корасик поиска множества подстрок в строке, а так же реализовать данный алгоритм на языке программирования C++.

Задание:

1) Разработать программу, решающую задачу точного поиска набора образцов.
2) Используя реализацию точного множественного поиска, решить задачу точного поиска для одного образца с джокером.

Ход работы:

1) Был реализован алгоритм Ахо-Корасика для поиска множества подстрок в строке на языке программирования C++.

- **a.** Для корректной работы алгоритма КМП на первом шаге была реализована структура вершины бора. В структуре next_vertex – это массив вершин, в которые можно попасть из данной вершины; flag – переменная типа bool, которая определяет является ли вершина финальной для какого-либо шаблона; auto_move – массив переходов из одного состояния в другое; suff_link – переменная для хранения суффиксной ссылки; par – номер вершины-родителя; symbol – символ, по которому осуществляется переход от родителя.
- **b.** Далее была написана функция, которая вставляет строку в бор. Осуществляется проход по строке, которую необходимо вставить. Если проход по уже существующему бору невозможен, то создается новая вершина и добавляется в бор.
- **c.** После были реализованы две функции, которые строят конечный детерминированный автомат по данному бору. Первая функция реализует получение суффиксной ссылки для данной вершины. Вторая функция выполняет переход из одного состояния автомата в другое.
- **d.** Далее была реализована функция для поиска шаблона в исходном тексте.

2) Был реализован алгоритм, который используя реализацию точного множественного поиска, решит задачу точного поиска для одного образца с джокером.

- **a.** Идея работы алгоритма:

1. C — вектор длины T, инициализированный нулями.
2. $\mathbb{P} = \{P_1, P_2, \ldots, P_k\}$ — набор максимальных подстрок P без джокеров. $l_1, l_2, \ldots, l_k$ — начальные позиции этих подстрок в P. Для $P = ab??c?ab??$ $\mathbb{P} = \{ab, c, ab\}$ и $l_1 = 1$, $l_2 = 5$ и $l_3 = 7$
3. Алгоритмом Ахо-Корасик найти все вхождения P_i в T. Для каждого вхождения P_i в j-й позиции текста увеличить счётчик $C[j - l_i + 1]$ на единицу.
4. Вхождение P в T, начинающиеся в позиции p, имеется в том и только том случае, если $C(p) = k$.
5. Время поиска $O(km)$ из-за использования массива C, если k ограничено константой, не зависящей от $|P|$, то время поиска — линейно.

b. Для корректной работы программы структура вершины бора была модифицирована. Теперь одна вершина может хранить информацию о нескольких шаблонах, которые в ней заканчиваются.

Пример работы программы:

1) Входные данные:

```
CCCA
1
CC
```

Результат работы программы:

```
1 1
2 1
```

2) Входные данные:

```
ACT
A$
$
```

Результат работы программы:

```
1
```

Заключение:

В ходе выполнения лабораторной работы был изучен и реализован на языке программирования C++ алгоритм Ахо-Корасик для поиска множества подстрок в строке. Данный алгоритм делает точный поиск набора образцов в строке. В нем используются такие понятия, как бор, конечный детерминированный автомат, суффиксные ссылки. Если учитывать вычисления автомата и суффиксных. ссылок, то алгоритм работает O(M*k+N+t), где M=bohr.size(). Память — константные массивы размера k для каждой вершины бора, откуда и выливается оценка O(M*k).

Приложение А
Исходный код программы lr5_1.cpp

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

#define ALP 5

using namespace std;

struct bohr_vertex {
	int next_vertex[ALP];
	int path_num;
	bool flag;
	int suff_link;
	int auto_move[ALP];
	int par;
	char symbol;
	int suff_flink;
};

vector <bohr_vertex> bohr;
vector <string> pattern;

bohr_vertex make_bohr_vertex(int par, char symbol) {
	bohr_vertex vertex;
	memset(vertex.next_vertex, 255, sizeof(vertex.next_vertex));
	vertex.flag = false;
	vertex.suff_link = -1;
	memset(vertex.auto_move, 255, sizeof(vertex.auto_move));
	vertex.par = par;
	vertex.symbol = symbol;
	vertex.suff_flink = -1;
	return vertex;
}

void init_bohr() {
	bohr.push_back(make_bohr_vertex(-1, -1));
}

int find(char symbol) {
	int ch;
	switch (symbol)
	{
	case 'A':
		ch = 0;
		break;
	case 'C':
		ch = 1;
		break;
	case'G':
		ch = 2;
		break;
	case 'T':
		ch = 3;
		break;
	case 'N':
		ch = 4;
		break;
	default:
		break;
	}
	return ch;
```

```
}

void add_string_to_bohr(string s) {
	int num = 0;
	for (int i = 0; i < s.length(); i++) {
		char ch = find(s[i]);
		if (bohr[num].next_vertex[ch] == -1) {
			bohr.push_back(make_bohr_vertex(num, ch));
			bohr[num].next_vertex[ch] = bohr.size() - 1;
		}
		num = bohr[num].next_vertex[ch];
	}
	bohr[num].flag = true;
	pattern.push_back(s);
	bohr[num].path_num = pattern.size() - 1;
}

int get_auto_move(int v, char ch);

int get_suff_link(int v) {
	if (bohr[v].suff_link == -1) {
		if (v == 0 || bohr[v].par == 0) {
			bohr[v].suff_link = 0;
		}
		else {
			bohr[v].suff_link = get_auto_move(get_suff_link(bohr[v].par),
bohr[v].symbol);  //пройдем по суф.ссылке предка и запустим переход по символу.
		}
	}
	return bohr[v].suff_link;
}

int get_auto_move(int v, char ch) {				//вычисляемая функция переходов
	if (bohr[v].auto_move[ch] == -1) {
		if (bohr[v].next_vertex[ch] != -1) {			//если из текущей
вершины еть ребро с символом ch
			bohr[v].auto_move[ch] = bohr[v].next_vertex[ch];	//то идем по нему
		}
		else {
			if (v == 0) {
				bohr[v].auto_move[ch] = 0;
			}
			else {
				bohr[v].auto_move[ch] = get_auto_move(get_suff_link(v), ch);
//иначе перейдем по суффиксальной ссылке
			}
		}
	}
	return bohr[v].auto_move[ch];
}

int get_suff_flink(int v) {
	if (bohr[v].suff_flink == -1) {
		int u = get_suff_link(v);
		if (u == 0) {
			bohr[v].suff_flink = 0;
		}
		else {
			bohr[v].suff_flink = (bohr[u].flag) ? u : get_suff_flink(u);  //если
для вершины по суф.ссылке flag=true, то это искомая вершина, иначе рекурсия.
		}
	}
	return bohr[v].suff_flink;
}
```

```
void check(int v, int i) {
	for (int u = v; u != 0; u = get_suff_flink(u)) {
		if (bohr[u].flag) {
			cout << i - pattern[bohr[u].path_num].length() + 1 << " " <<
bohr[u].path_num +1<< endl;
		}
	}
}

void find_all_pos(string s) {
	int u = 0;
	for (int i = 0; i < s.length(); i++) {
		u = get_auto_move(u, find(s[i]));
		check(u, i + 1);
	}
}


int main() {
	string text;
	int n;
	init_bohr();
	cin >> text;
	cin >> n;
	for (int i = 0; i < n; i++) {
		string temp;
		cin >> temp;
		add_string_to_bohr(temp);
	}
	find_all_pos(text);
	return 0;
}
```

Приложение В
Исходный код программы lr5_2.cpp

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

#define ALP 5

using namespace std;

struct numbers {
	long long int index;
	int pattern_num;
};

struct bohr_vertex {
	int next_vertex[ALP];
	bool flag;
	int suff_link;
	int auto_move[ALP];
	int par;
	char symbol;
	int suff_flink;
	int pattern_num[40];
};

vector<numbers> num;
vector <bohr_vertex> bohr;
vector <string> pattern;

bohr_vertex make_bohr_vertex(int par, char symbol) {
	bohr_vertex vertex;
	memset(vertex.next_vertex, 255, sizeof(vertex.next_vertex));
	vertex.flag = false;
	vertex.suff_link = -1;
	memset(vertex.auto_move, 255, sizeof(vertex.auto_move));
	vertex.par = par;
	vertex.symbol = symbol;
	vertex.suff_flink = -1;
	memset(vertex.pattern_num, 255, sizeof(vertex.pattern_num));
	return vertex;
}

void init_bohr() {
	bohr.push_back(make_bohr_vertex(-1, -1));
}

int find(char symbol) {
	int ch;
	switch (symbol)
	{
	case 'A':
		ch = 0;
		break;
	case 'C':
		ch = 1;
		break;
	case'G':
		ch = 2;
		break;
	case 'T':
```

```
			ch = 3;
			break;
		case 'N':
			ch = 4;
			break;
		default:
			break;
		}
		return ch;
}

void add_string_to_bohr(string s) {
		int num = 0;
		for (int i = 0; i < s.length(); i++) {
			char ch = find(s[i]);
			if (bohr[num].next_vertex[ch] == -1) {
				bohr.push_back(make_bohr_vertex(num, ch));
				bohr[num].next_vertex[ch] = bohr.size() - 1;
			}
			num = bohr[num].next_vertex[ch];
		}
		bohr[num].flag = true;
		pattern.push_back(s);
		for (int i = 0; i < 40; i++) {
			if (bohr[num].pattern_num[i] == -1) {
				bohr[num].pattern_num[i] = pattern.size() - 1;
				break;
			}
		}
}

int get_auto_move(int v, char ch);

int get_suff_link(int v) {
		if (bohr[v].suff_link == -1) {
			if (v == 0 || bohr[v].par == 0) {
				bohr[v].suff_link = 0;
			}
			else {
				bohr[v].suff_link = get_auto_move(get_suff_link(bohr[v].par),
bohr[v].symbol);  //пройдем по суф.ссылке предка и запустим переход по символу.
			}
		}
		return bohr[v].suff_link;
}

int get_auto_move(int v, char ch) {                          //вычисляемая функция переходов
		if (bohr[v].auto_move[ch] == -1) {
			if (bohr[v].next_vertex[ch] != -1) {                  //если из текущей
вершины еть ребро с символом ch
				bohr[v].auto_move[ch] = bohr[v].next_vertex[ch];      //то идем по нему
			}
			else {
				if (v == 0) {
					bohr[v].auto_move[ch] = 0;
				}
				else {
					bohr[v].auto_move[ch] = get_auto_move(get_suff_link(v), ch);
//иначе перейдем по суффиксальной ссылке
				}
			}
		}
		return bohr[v].auto_move[ch];
}
```

```
int get_suff_flink(int v) {
	if (bohr[v].suff_flink == -1) {
		int u = get_suff_link(v);
		if (u == 0) {
			bohr[v].suff_flink = 0;
		}
		else {
			bohr[v].suff_flink = (bohr[u].flag) ? u : get_suff_flink(u);  //если
для вершины по суф.ссылке flag=true, то это искомая вершина, иначе рекурсия.
		}
	}
	return bohr[v].suff_flink;
}

void check(int v, int i) {
	struct numbers s;
	for (int u = v; u != 0; u = get_suff_flink(u)) {
		if (bohr[u].flag) {
			for (int j = 0; j < 40; j++) {
				if (bohr[u].pattern_num[j] != -1) {
					s.index = i - pattern[bohr[u].pattern_num[j]].length();
					s.pattern_num = bohr[u].pattern_num[j];
					num.push_back(s);
				}
				else
					break;
			}
		}
	}
}

void find_all_pos(string s) {
	int u = 0;
	for (int i = 0; i < s.length(); i++) {
		u = get_auto_move(u, find(s[i]));
		check(u, i + 1);
	}
}

int main() {
	vector<string> patterns;  //подстроки при делении по джокеру
	vector<int> patterns_pos; //позиции подстрок
	string text;
	string temp;
	char joker;
	string pat;
	cin >> text >> temp >> joker;
	init_bohr();
	for (int i = 0; i < temp.length(); i++) {
		if (temp[i] != joker) {
			patterns_pos.push_back(i + 1);
			for (int j = i; temp[j] != joker && j != temp.length(); j++) {
				pat += temp[j];
				i++;
			}
			add_string_to_bohr(pat);
			patterns.push_back(pat);
			pat.clear();
		}
	}
	find_all_pos(text);
	vector<int> c(text.length(), 0);
	for (int i = 0; i < num.size(); i++) {
```

```
            if (num[i].index < patterns_pos[num[i].pattern_num] - 1) continue;
            c[num[i].index - patterns_pos[num[i].pattern_num] + 1]++;
            if (c[num[i].index - patterns_pos[num[i].pattern_num] + 1] == patterns.size()
&&
                  num[i].index - patterns_pos[num[i].pattern_num] + 1 <= text.length() -
temp.length())
                  cout << num[i].index - patterns_pos[num[i].pattern_num] + 2 << endl;
      }
      return 0;
}
```