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 Кнут-Моррис-Пратта

Студент гр. 3384 _____ Рудаков А.Л.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2025

Цель работы.

Написание алгоритма Кнута-Морриса-Пратта на языке программирования.

Задание.

Реализуйте алгоритм КМП и с его помощью для заданных шаблона P (|P|≤15000 и текста (|T|≤5000000) найдите все вхождения P в T.

Вход:

Первая строка - P

Вторая строка - T

Выход:

индексы начал вхождений P в T, разделенных запятой, если P не входит в T, то вывести −1

Sample Input:

ab

abab

Sample Output:

0,2

2. Заданы две строки A (|A|≤5000000) и B (|B|≤5000000).

Определить, является ли A циклическим сдвигом B (это значит, что A и B имеют одинаковую длину и A состоит из суффикса B, склеенного с префиксом B). Например, defabc является циклическим сдвигом abcdef.

Вход:

Первая строка - A

Вторая строка - B

Выход:

Если A вляется циклическим сдвигом B, индекс начала строки B в A, иначе вывести −1. Если возможно несколько сдвигов вывести первый индекс.

Sample Input:

defabc

abcdef

Sample Output:

3

Выполнение работы.

Для задания 1 разработан алгоритм, находящийся в фале KMP.cpp.

В нем реализован алгоритм Кнута-Морриса-Пратта, выводящий все индексы вхождения паттерна в текст.

Функция *void input(string &pattern, string &text)* считывает паттерн и текст из консоли.

Функция *void find_prefix_func(string& pattern, vector<int>& prefix_func)* выполняет поиск префиксной функции и заполняет массив префиксной функции значениями. Внутри происходит проход по паттерну, при этом на каждом индексе мы ищем первый нужный нам индекс повторяющегося префикса и суффикса, стоящий перед ним, после чего добавляем найденное значения в вектор.

Функция *vector<int> find_index(string& pattern, string& text)* выполняет поиск индексов вхождения паттерна в текст. Внутри мы проходим по тексту и паттерну слева направо, пока символы совпадают, при этом смотря на завершение паттерна. Если он завершился. Добавляем индекс начала вхождения, после чего продолжаем поиск со следующего индекса в тексте. Если символы не совпали, то мы двигаем паттерн на найденную заранее префиксную функцию для данного индекса в паттерне, после чего продолжаем поиск.

Таким образом если нам попался суффикс, который имеет повторение в паттерне в виде префикса, то мы сразу двигаем паттерн так, чтобы символы совпали, тем самым уменьшая количество операций относительно наивного поиска.

Функция *void output(vector<int>& finded_idx)* выводит массив найденных индексов, или -1, если индексы не были найдены.

Тесты представлены в табл. Б1.

Для задания 2 написан алгоритм, находящийся в файле second.cpp.

Он реализует поиск того, является ли первая строка циклическим сдвигом второй строки.

Функция *void input(string &pattern, string &text)* считывает паттерн и текст из консоли.

Функция *void find_prefix_func(string& pattern, vector<int>& prefix_func)* выполняет поиск префиксной функции и заполняет массив префиксной функции значениями. Внутри происходит проход по паттерну, при этом на каждом индексе мы ищем первый нужный нам индекс повторяющегося префикса и суффикса, стоящий перед ним, после чего добавляем найденное значения в вектор.

Функция int find_cyclic_shift(string& first, string& second) ищет индекс, с которого начинается циклический сдвиг первой строки относительно второй. Если строки не равны по длине, сразу возвращается -1. Далее происходит проход по второй строке, для поиска совпавшей с первой частью строки(индексы меняются также, как в первом задании). Далее вычисляется остаток (количество оставшихся с начала второй строки символов, которые должны быть конечными символами первой строки). После чего происходит проход по остатку (началу первой строки), сравнивая посимвольно с оставшимися символами первой строки. Если хотя бы один раз символ не совпадает, то сразу возвращается -1. Если все символы совпали, то возвращается индекс начала циклического сдвига.

Функция *void output(int idx)* выводит индекс.

Тесты представлены в табл. Б2.

Дополнительное задание.

Реализация дополнительного алгоритма - Алгоритм Бойера — Мура. Привести подробное тестирование, указав различные возможные случаи. Написать вычислительную сложность алгоритма

Выполнение дополнительного задания.

Для дополнительного задания написан алгоритм, находящийся в файле Boyer_Moore.cpp.

Он реализует поиск вхождений паттерна в строку, после чего выводит индексы вхождений.

Алгоритм работает на двух эвристиках: эвристика плохого символа и эвристика хорошего суффикса. Кроме этого, алгоритм проверяет текст слева направо, при этом сравнивая с паттерном справа налево.

Функция *void create_symbols_table(string& pattern, map<char, int>& bad_symbols)* реализует создание таблицы плохих символов. Внутри происходит итерация по паттерну, где для каждого символа, находящегося в строке, вычисляется сдвиг, равный длине паттерна – индекс в строке (начиная с 1).

Функция *void create_suffix_table(string& pattern, map<int, int>& good_suffix)* реализует создание таблицы хороших суффиксов. Для каждого суффикса строки ищется такой же фрагмент в строке, не равный ему самому. Если он находится, то в таблицу записывается сдвиг, равный длине паттерна – длина суффикса – индекс найденного места. Если вхождений не было найдено, то сдвиг становиться равен длине паттерна.

Функция *vector<int> do_boere_moore(string& pattern, string& text, map<int, int>& good_suffix, map<char, int>& bad_symbols)* реализует поиск паттерна в тексте, применяя обе эвристики. Внутри происходит итерация по тексту. Внутри нее сравниваются текущие символы текста и символы паттерна, начиная с его конца. Когда символы не равны или паттерн закончился, происходит проверка на то, был ли равен весь паттерн, если да, то индекс записывается в возвращаемый вектор индексов, индекс текста перемещается на длину паттерна

+ 1 вперед, индекс паттерна становиться на последний символ паттерна. Если же был использован не весь паттерн, то вычисляется значение, на которое можно сдвинуть индекс по эвристике плохого символа и по эвристике хорошего суффикса, после чего индекс текста сдвигается на максимальное из полученных значений, индекс паттерна начинает указывать на последний символ паттерна.

Использование данных двух эвристик помогает укоротить поиск паттерна в тексте, не сравнивая лишние символы, когда мы точно знаем, что при большем сдвиге ничего не пропустим.

Например, у нас есть:

текст aababcxfdedeaxardeabcxadebde

паттерн abcxadebde

Начиная сравнение с конца паттерна, мы получаем несовпадение на суффиксе bde (fde в тексте). Сдвиг по суффиксу de равен 3, но символа f в паттерне не было, следовательно сдвигаем на величину паттерна.

текст aababcxfdedeaxaedeabcxadebde

паттерн abcxadebde

Несовпадение на суффиксе bde (ade в тексте). Сдвиг по суффиксу de равен 3, сдвиг по букве e так же 3, поэтому сдвигаем на 3.

текст aababcxfdedeaxaedeabcxadebde

паттерн abcxadebde

Несовпадение на суффиксе e (c в тексте). Сдвиг по суффиксу равен 3, сдвиг по c равен 7. Сдвигаем на 7.

текст aababcxfdedeaxaedeabcxadebde

паттерн abcxadebde

Совпадение.

Тесты представлены в табл. Б3.

Для рассчета сложности будем считать, что длина текста n, длина паттерна m. Тогда в лучшем случае (когда в тексте и паттерне нет одинаковых символов) сложность будет равна O(n/m). В худшем случае (когда в текст будет

состоять из кусков, почти равных паттерну, но не равных полностью ему) сложность будет O(n*m). В среднем случае сложность равна O(n+m).

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

В ходе работы был реализован алгоритм Кнута-Морриса-Пратта для поиска заданного паттерна в заданном тексте, работающий на логике построения перед началом поиска таблицы префиксной функции для каждого символа подаваемого паттерна. После построения таблицы префиксной функции алгоритм итерируется по тексту быстрее, чем это делает алгоритм наивного поиска. Кроме этого, был реализован алгоритм, позволяющий понять, является ли одна поданная строка циклическим сдвигом другой поданной строки. Алгоритм основывается на алгоритме Кнут-Моррис-Пратта, он так же вычисляет таблицу префиксных функций для каждого символа первой заданной строки, после чего начинает поиск последней части второй строки, которая будет идентична началу первой строки, применяя при этом данную таблицу. После нахождения данной части происходит итерация по начальной части второй строки, сравнивая ее при этом с оставшейся частью первой строки, при несовпадении указывая, что циклический сдвиг не найден, или найден в противном случае.

Для решения дополнительной задачи был написан алгоритм Бойера-Мура, реализующий поиск паттерна в поданном тексте. За основу берутся две эвристики: эвристика хорошего суффикса и эвристика плохого символа. Обе эвристики помогают понять максимальный возможный сдвиг паттерна во время его поиска в тексте, не теряя при этом качества (не пропуская нужные значения). Алгоритму так же нужно вначале создать таблицу плохих символов и таблицу хороших суффиксов, для корректной работы. Сложность алгоритма в худшем

случае O(n*m), в среднем случае O(n+m), в лучшем случае O(n/m), где n – длина заданного текста, m – длина заданного паттерна.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: KMP.cpp

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

using namespace std;

void input(string &pattern, string &text) {
	cin >> pattern;
	cin >> text;
}

void find_prefix_func(string& pattern, vector<int>& prefix_func) {
	prefix_func.push_back(0);
	for (int cur_idx = 1; cur_idx < pattern.size(); cur_idx++) {
		int start_index = prefix_func[cur_idx - 1];
		while (start_index > 0 && pattern[start_index] != pattern[cur_idx]) {
			start_index = prefix_func[start_index - 1];
		}
		if (pattern[start_index] == pattern[cur_idx]) start_index++;
		prefix_func.push_back(start_index);
	}
}

vector<int> find_index(string& pattern, string& text) {
	vector<int> prefix_func;
	find_prefix_func(pattern, prefix_func);
	int current_idx = 0;
	vector <int> returned_idx;
	int i = 0;
	while (i < text.size()) {
		if (pattern[current_idx] == text[i]) {
```

```
                    current_idx++;
                    if (current_idx == pattern.size()) {
                        returned_idx.push_back(i - current_idx + 1);
                        current_idx = prefix_func[current_idx - 1];
                    }
                    i++;
                }
                else {
                    if (current_idx != 0) current_idx = prefix_func[current_idx - 1];
                    else i++;
                }
            }
            return returned_idx;
        }

        void output(vector<int>& finded_idx) {
            string out;
            for (int i : finded_idx) {
                out += to_string(i) + ",";
            }
            if (out.size() > 0) {
                out.pop_back();
                cout << out << '\n';
            }
            else {
                cout << "-1\n";
            }
        }


        int main() {
            string pattern, text;
            input(pattern, text);
            vector<int> finded_idx = find_index(pattern, text);
            output(finded_idx);
        }
```

Название файла: second.cpp

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

using namespace std;

void input(string& pattern, string& text) {
	cin >> pattern;
	cin >> text;
}

void find_prefix_func(string& pattern, vector<int>& prefix_func) {
	prefix_func.push_back(0);
	for (int cur_idx = 1; cur_idx < pattern.size(); cur_idx++) {
		int start_index = prefix_func[cur_idx - 1];
		while (start_index > 0 && pattern[start_index] != pattern[cur_idx]) {
			start_index = prefix_func[start_index - 1];
		}
		if (pattern[start_index] == pattern[cur_idx]) start_index++;
		prefix_func.push_back(start_index);
	}
}

int find_cyclic_shift(string& first, string& second) {
	if (first.size() != second.size()) return -1;
	vector<int> prefix_func;
	find_prefix_func(first, prefix_func);
	int current_idx = 0;
	int i = 0;
	while (i < second.size()) {
		if (first[current_idx] == second[i]) {
			current_idx++;
			i++;
		}
		else {
```

```
                    if (current_idx != 0) current_idx = prefix_func[current_idx - 1];
                    else i++;
                }
            }
            if (current_idx == i) return 0;
            int remains = i - current_idx;
            i = 0;
            while (i < remains) {
                if (first[current_idx] == second[i]) {
                    current_idx++;
                    i++;
                }
                else return -1;
            }
            return second.size() - remains;
        }

        void output(int idx) {
            cout << idx << '\n';
        }


        int main() {
            string first, second;
            input(first, second);
            int idx = find_cyclic_shift(first, second);
            output(idx);
        }
```

Название файла: Boyer_Moore.cpp

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

using namespace std;
```

```
	void input(string& pattern, string& text) {
		cin >> pattern;
		cin >> text;
	}

	void create_suffix_table(string& pattern, map<int, int>&
good_suffix) {
		for (int i = 1; i < pattern.size(); i++) {
			string suffix = pattern.substr(pattern.size() - i, i);
			good_suffix[i] = pattern.size();
			for (int j = pattern.size() - i - 1; j > -1; j--) {
				if (pattern.substr(j, i) == suffix) {
					good_suffix[i] -= (i + j);
					break;
				}
			}
		}
		good_suffix[pattern.size()] = pattern.size();
	}

	void create_symbols_table(string& pattern, map<char, int>&
bad_symbols) {
		for (int i = 0; i < pattern.size(); i++) {
			bad_symbols[pattern[i]] = pattern.size() - i - 1;
		}
	}

	vector<int> do_boere_moore(string& pattern, string& text, map<int,
int>& good_suffix, map<char, int>& bad_symbols) {
		int t_idx = pattern.size() - 1;
		int p_idx = pattern.size() - 1;
		vector<int> returned;
		if (pattern.size() <= text.size()) {
			while (t_idx < text.size()) {
				while (p_idx >= 0 && pattern[p_idx] == text[t_idx])
{
					p_idx--;
					t_idx--;
				}
```

```
                    if (p_idx == -1) {
                        returned.push_back(t_idx + 1);
                        t_idx += pattern.size() + 1;
                        p_idx = pattern.size() - 1;
                    }
                    else {
                        int good = (pattern.size() - p_idx - 1) +
good_suffix[pattern.size() - p_idx - 1];
                        int bad = 0;
                        if (bad_symbols.count(text[t_idx]) == 0) {
                            bad = pattern.size();
                        }
                        else {
                            bad = max(1, bad_symbols[text[t_idx]]);
                        }
                        int shift = max(good, bad);
                        t_idx += shift;
                        p_idx = pattern.size() - 1;
                    }
                }
            }
            return returned;
        }

        void output(vector<int>& out) {
            cout << "indexes:\n";
            for (int i : out) {
                cout << i << " ";
            }
            cout << "\n";
        }

        int main() {
            string pattern, text;
            input(pattern, text);
            map <int, int> good_suffix;
            create_suffix_table(pattern, good_suffix);
            map <char, int> bad_symbols;
            create_symbols_table(pattern, bad_symbols);
```

```
            vector<int> out = do_boere_moore(pattern, text, good_suffix,
bad_symbols);
            output(out);
      }
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Тесты для первого задания представлены в табл. Б1.

Таблица Б.1 - Примеры тестовых случаев задания 1.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	ab abab	0,2	Ok
2.	aaa aaaaaaaaaa	0,1,2,3,4,5,6,7	Ok
3.	aba abababab	0,2,4	Ok
4.	ab kldsjslkdslskmsksld	-1	Ok
5.	abc cbabbcacbbacabc	12	Ok
6.	abc a	-1	Ok
7.	abc abc	0	Ok

Тесты для второго задания представлены в табл. Б2.

Таблица Б.2 - Примеры тестовых случаев задания 2.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	abc abc	0	Ok
2.	abc bca	1	Ok
3.	abc cab	2	Ok
4.	aaaaaa aaaaaaa	-1	Ok
5.	aaaaaa aaaaa	-1	Ok
6.	aaaaa aaaaa	0	Ok

Тесты для дополнительного задания представлены в табл. Б3.

Таблица Б.3 - Примеры тестовых случаев дополнительного задания.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	abc abc	0	Ok
2.	ab acbacbababababcabasab	6,8,10,13,17	Ok
3.	aa aaaaaaaa	0,1,2,3,4,5,6	Ok
4.	abc jjslklksksbackskskj	-1	Ok
5.	abcxadebde aababcxfdedeaxaedeabc xadebde	18	Ok (тест из примера)