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

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритм Кнута-Морриса-Пратта

Студент гр. 8303 _____ Гришин К. И.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2020

Цель работы

Изучить алгоритм Кнута-Морриса-Пратта для нахождения подстроки в строке, а также имплементировать его на языке C++.

Задание

1. Реализуйте алгоритм КМП и с его помощью для заданных шаблона ***P*** ($|P| \leq 15000$) и текста ***T*** ($|T| \leq 5000000$) найдите все вхождения P в T.

Вход:
Первая строка — P
Вторая строка — T

Выход:
индексы начал вхождения P в T, разделенных запятой, если P не входит в T, то вывести *-1*.

Sample input	Sample output
ab abab	0, 2

2. Заданы две строки ***A*** ($|A| \leq 5000000$) и ***B*** ($|B| \leq 5000000$).

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

Sample input	Sample output
defabc abcdef	3

Индивидуальное задание, вариант 1

Подготовка к распараллеливанию: работа по поиску разделяется на k равных частей, пригодных для обработки k потоками (при этом длина образца гораздо меньше длины строки поиска).

Префикс-функция и ее вычисление

Префикс-функция от строки S и позиции в i в ней — длина наибольшего префикса подстроки $S[1...i]$, который одновременно является суффиксом этой подстроки. То есть в начале подстроки $S[1...i]$ длиной i нужно найти такой префикс максимальной длины $k < i$, который был бы суффиксом данной подстроки $S[1...k] == S[(i-k+1)...i]$.

Префикс-функция каждого элемента строки вычисляется по-очереди, при этом префикс-функция первого элемента равна нулю. Допустим имеется строка $S[0...s]$ длины $s+1$, а также массив $\pi[0...s]$. Назовем текущий рассматриваемый символ i, а последний символ совпадающего префикса j. Тогда начнем с $j = 0$ и $i = 1$ (т.к. $\pi[0] = 0$):

если $s[i] \neq s[j]$, тогда, если $j = 0$, то $\pi[i] = 0$; i++;

иначе (т.е. $j \neq 0$) $j = \pi[j-1]$;

иначе (т.е. $s[i] \neq s[j]$) $\pi[i] = j+1$; i++; j++;

как только i стал равен $s+1$, вычисление префикс функции заканчивается.

Описание алгоритмов

Поиск подстроки P в строке S

Инициализируем алгоритм двумя индексами: $i = 0$ — текущий индекс строки, $j = 0$ — текущий индекс подстроки, а также префикс-функцией π строки P.

Пока i не равен длине S:

1. Сравниваются символы $S[i]$ и $P[j]$.
2. Если равны, то увеличить i и j на 1. При этом если j равен размеру P, то ***вхождение найдено*** и вычитанием из i длины P получаем индекс вхождения P в S. Если установить в j $\pi[j-1]$, то можно продолжать поиск.
3. Иначе если $j \neq 0$, то установить в j $\pi[j-1]$

Циклический сдвиг. Определить, является ли *A* циклическим сдвигом *B*

Назовем размер строк A и B — *size*.

Инициализируем алгоритм тремя индексами: *i = 0* — текущий индекс A, *j = 0* — текущий индекс B, *k = 0* — итератор, который дважды проходится по A, а также префикс-функцией π строки B.

Пока *k* не станет равен *длина(A) · 2 — 1*:

1. Получаем i из k, i = k mod size.
2. Сравниваются символы A[i] и B[j]
3. Если равны, то увеличить i и k на единицу. При этом если j равен size, то ***сдвиг найден*** со смещением k — size
4. Иначе если j ≠ 0, то установить в j π[j-1]

Разбиение входной строки *S* на *k* частей для параллельной обработки алгоритмом КМП с подстрокой P. k > 0;

Инициализация

1. Получим длину *part_length* одной части делением длины S на k нацело.
2. Если *part_length* оказалась меньше длины подстроки *S*, то разбиение на *k* частей непригодно, тогда в *part_length* заносим длину *P*, в *k* устанавливаем частное деления длины *S* на длину *P* нацело, такую ситуацию назовем *не критичной ошибкой*
3. Если после пункта 2 k стал равен 0, то разбиение невозможно вовсе, функция завершается с *критической ошибкой*
4. Получим остаток текста, после его целочисленного деления на части, назовем его *residual*
5. Получим половину длины P с округлением в большую сторону, назовем ее *connector*
6. Установим начало и конец текущей части в S, назовем их *token_begin = 0* — начало части, *token_end = part_length* — конец части, а также номер текущей части *i* = 0;

Основной ход

7. Если *residual* больше *0*, то нужно распределить остаток текста, для оставшихся частей. Для этого *residual* делится на оставшееся число частей (*k* - *i*) с округлением вверх, полученное число назовем *extra*. К *token_end* прибавим *extra*, а от *residual* отнимем.
8. ***Запомним строку*** *S*[*token_begin*, … , *token_end* + *connector*], а ***также ее индекс*** *token_begin* в основной строке. Важно: прибавлять *connector* нужно для всех частей, кроме последней!
9. Записать в *token_begin token_end*, а к *token_end* прибавить *part_length*
10. Если *i* < *k*, то вернуться к пункту 7.

Все полученные в пункте 8 строки пригодны для применения к ним алгоритма поиска подстроки *P*. А по ее полученному индексу восстанавливаются вхождения *P* в *S*.

Описание основных внешних функций

`vector<int> prefix(const string& image)` — функция получения префикс-функции строки. На вход получает образ строки, префикс-функцию которой нужно получить, на выходе массив значений префикс-функции каждого символа в строке *image* по порядку расположения символов.

`vector<int> KMPA(const string& haystack, const string& needle)` — функция получения всех индексов вхождения подстроки в строку. На вход получает строку *haystack* и строку *needle*. Возвращает массив индексов вхождения *needle* в *haystack* в порядке возрастания.

`vector<int> cyclicShift(const string& base, const string& shifted)` — функция получения смещения одной строки относительно другой. На вход получает строку *base* и строку *shifted*. Возвращает массив с одним элементом, в котором хранится смещение *shifted* относительно *base*.

Функция возвращает массив а не число для того, чтобы быть схожей с функцией поиска подстроки.

Описание дополнительных функций, классов, их методов

Класс TextPart:

Класс для хранения части разбитой строки. Объект класса создается во время разбиения подстроки на строки.

Хранит в себе строку *const string _str* и ее индекс *const size_t _index*.

Методы класса:

`private Конструктор(const string& str, const size_t index)` — устанавливает *str* и *index* в соответствующие поля класса.

`public Конструктор(const TextPart& other)` — конструктор копирования, создает объект подобный объекту *other*.

`const string& str() const` — метод получения строки.

`const size_t& index() const` — метод получения индекса строки.

Класс Splitter:

Содержит в себе *public* перечисление *Error*, где перечисляются возможные ошибки создания объекта класса.

Класс разбивающий строку на подстроки, если это возможно.

Хранит в себе массив подстрок *vector<TextPart> parts* и ошибку Error, которой был создан.

Методы класса:

`Конструктор(const string& text, size_t parts_count, size_t minimal_part_size)` — Разбивает строку на заданное число подстрок с учетом минимального размера подстроки. На вход получает строку *text*, количество частей разбиениея *parts_count* и минимальный размер одной части *minimal_part_size*. Если строку удалось разбить на заданное число частей, то части запоминаются в виде объектов TextPart в поле parts, а в поле *error* заносится *Error::absent*. Если число частей разбиения уменьшилось (из-за размера минимальной части), то полученное разбиение строки также запоминается в parts, но в error уже заносится *Error::some_parts_excluded* (не критичная ошибка). Если разбить

строку не получилось, массив parts остается пустым, а в error заносится *Error::cannot_be_splitted* (критическая ошибка).

`Error error() const` — метод получения ошибки, с которой завершилось разбиение текста.

`const string& getParts() const` — метод получения разбиения текста.

Утилитарные функции

`vector<int> distributeText(const vector<TextPart>& parts, const string needle)` — функция, применяющая алгоритм поиска подстроки в строке к каждой из строк распараллеленного текста. На вход получает массив *parts* объектов *TextPart* и подстроку *needle* для поиска. Возвращает массив индексов вхождения *needle* в текст, который состоит из частей в *parts*, в порядке возрастания.

Сложность алгоритма

Значение префикс-функции строки вычисляется последовательно для каждого символа этой строки, следовательно сложность вычисления префикс-функции $O(|P|)$. Во время нахождения всех вхождений P в T алгоритм проходится один раз по строке T, следовательно сложность основного хода алгоритма $O(|T|)$. В итоге, с учетом вычисления префикс функции, алгоритм имеет общую ***сложность по операциям*** $\boldsymbol{O(|P|+|T|)}$. Во время работы алгоритма требуется хранить значения префикс-функции, которым нужен массив размером |P|, следовательно, ***сложность по памяти*** $\boldsymbol{O(|P|)}$.

Хранение решения

Строки используют класс std::string стандартной библиотеки C++. Значения индексов и префикс-функции хранятся в динамических массивах std::vector<int> стандартной библиотеки C++. В индивидуальном задании используется оболочка TextPart над std::string, которая помимо строки, хранит индекс ее расположения в основной строке.

Использованные оптимизации

Вместо конкатенации подстроки и строки, префикс функция находится только для подстроки.

Тестирование

Логи каждого теста хранятся в директории ./tests/<program_name>/

KMP_search:

input	ab abab
output	0,2
input	lilia parallelilimikliliarlililialilikoplilkliliaklili
output	14,22,38
input	rare rawlrawlrarirarererorerarerararer
output	12,20,26
input	aaaa aaabaaabaaaabaaabaaaaabaaaaaaa
output	8,17,18,23,24,25,26
input	poof thisstringdoesnotcontainneedle
output	nil

cyclic_shift:

input	defabc abcdef
output	3
input	tsplitspli splitsplit
output	1
input	aaaaaaaaaaaaa aaaaaaaaaaaaa
output	0
input	strings have differrent size
output	shifted string and base string have different sizes nil

input	therearentsh iftedstrings
output	nil

split_text:

input	Something in a thirty-acre thermal thicket of thorns and thistles thumped and thundered threatening the three-D thoughts of Matthew the thug - although, theatrically, it was only the thirteen-thousand thistles and thorns through the underneath of his thigh that the thirty year old thug thought of that morning. 10 thou
output	Something in a thirty-acre thermal; 0 al thicket of thorns and thistles; 32 es thumped and thundered threaten; 63 ening the three-D thoughts of Mat; 94 atthew the thug - although, theat; 125 atrically, it was only the thirte; 156 teen-thousand thistles and thorns; 187 ns through the underneath of his ; 218 s thigh that the thirty year old ; 249 d thug thought of that morning.; 280 112,145,192,287
input	There was a fisherman named Fisher who fished for some fish in a fissure. Till a fish with a grin, pulled the fisherman in. Now they're fishing the fissure for Fisher. 8 fish
output	There was a fisherman n; 0 named Fisher who fishe; 21 hed for some fish in a ; 42 a fissure. Till a fish ; 63 h with a grin, pulled t; 84 the fisherman in. Now ; 105 w they're fishing the f; 126 fissure for Fisher.; 147 12,39,55,81,110,136

input	Luke Luck likes lakes. Luke's duck likes lakes. Luke Luck licks lakes. Luck's duck licks lakes. Duck takes licks in lakes Luke Luck likes. Luke Luck takes licks in lakes duck likes. And you should know, there is no needle :) 13 poof
output	Luke Luck likes lake; 0 kes. Luke's duck lik; 18 ikes lakes. Luke Luc; 36 uck licks lakes. Lu; 54 Luck's duck licks l; 71 lakes. Duck takes ; 88 s licks in lakes Lu; 105 Luke Luck likes. Lu; 122 Luke Luck takes lic; 139 icks in lakes duck ; 156 k likes. And you sh; 173 should know, there ; 190 e is no needle :); 207 nil

Стоит обратить внимание на вторую строчку логов.

input	Let's try to split this little haystack into big amount of parts, but neeeeeeeeeeeedle big enough too 15 neeeeeeeeeeeedle
output	Let's try to split this l; 0 t this little haystack in; 17 stack into big amount of ; 34 ount of parts, but neeeee; 51 t neeeeeeeeeeeedle big en; 68 e big enough too; 85 70
input	very little haystack 3 SHOCKINGLYLARGENEEDLE
output	splitter error. check logs
log	splitter initiated splitter error: minimal part bigger than whole text

Вывод

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Кнута-Морриса-Пратта для поиска подстроки в строке, результатом которого является набор индексов вхождения подстроки. Для работы алгоритма была также реализована префикс-функция. Помимо основного алгоритма, также реализован механизм распараллеливания строки, для запуска алгоритма сразу в нескольких местах. Сложность алгоритма линейна и пропорциональна сумме длин строки и подстроки $O(|P| + |T|)$.

ПРИЛОЖЕНИЕ А

Файл KMP_search.cpp

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

std::ofstream log_file;

auto prefix(const std::string& image){
    if(log_file.is_open()){
        log_file << "prefix function initiated for \"" << image
                 << "\" pattern" << std::endl;
    }

    std::vector<int> pi;
    pi.push_back(0);
    for(size_t i = 1, j = 0; i < image.size(); i++){
        if(image[i] != image[j]){
            if(j == 0){
                pi.push_back(0);
            }
            else{
                j = pi[j-1];
                i--;
            }
        }
        else{
            pi.push_back(j+1);
            j++;
        }
    }

    if(log_file.is_open()){
        log_file << "prefix function is:";
        for(auto i: pi)
            log_file << " " << i;
        log_file << std::endl << std::endl;
    }

    return pi;
}


void logger(std::ofstream &log,
            const std::string& haystack,
            const std::string& needle,
            size_t i, size_t j)
{
    std::string border;
    border.assign(haystack.size(), '-');
    log << border << std::endl;

    std::string car_top;
    std::string car_bot;
    car_top.assign(i, ' ');
    car_bot = car_top + "^";
    car_top += "V";
    log << car_top << std::endl;

    log << haystack << std::endl;

    std::string needle_str;
    needle_str.assign(i-j, ' ');
    needle_str += needle;
    log << needle_str << std::endl;
    log << car_bot << std::endl;
    log << border << std::endl;
}


auto KMPA(const std::string& haystack, const std::string& needle){
    auto pi = prefix(needle);
    std::vector<int> overlaps;
    if(needle.size() > haystack.size()){
        std::cout << "haystack bigger than needle" << std::endl;
        return overlaps;
    }

    if(log_file.is_open()){
        log_file << "initial state" << std::endl;
        logger(log_file, haystack, needle, 0, 0);
        log_file << std::endl;
    }

    for(size_t i = 0, j = 0; i < haystack.size(); i++){
        auto ti = i;
        auto tj = j;

        if(haystack[i] == needle[j]){
            j++;
            if(j == needle.size()){
                if(log_file.is_open())
                    log_file << "pattern carriage equal to pattern size\n"
                             << "it means, we found pattern in text" << std::endl;

                overlaps.push_back(i - j + 1);
                j = pi[j-1];
```

```
            }
            if(log_file.is_open())
                log_file << "characters match, move carriage" << std::endl;
        }
        else{
            if(j != 0){
                if(log_file.is_open())
                    log_file << "characters don't match, set pattern carriage on \n"
                             << "prefix function of previous symbol: " << pi[j-1] << std::endl;
                j = pi[j-1];
                i--;
            }
            else{
                if(log_file.is_open())
                    log_file << "mismatch on first symbol of pattern,\n"
                             << "move haystack carriage" << std::endl;
            }
        }

        if(log_file.is_open()){
            logger(log_file, haystack, needle, ti, tj);
            log_file << std::endl;
        }
    }
    if(log_file.is_open()){
        log_file << "can't move carriage, end of string" << std::endl;
    }
    return overlaps;
}


int main(int argc, char* argv[]){
    if(argc > 1){
        log_file.open(argv[1], std::ios::out | std::ios::trunc);
    }

    std::string haystack;
    std::string needle;

    std::cin >> needle;
    std::cin >> haystack;

    auto overlaps = KMPA(haystack, needle);

    for(auto it = overlaps.begin(); it != overlaps.end(); it++){
        if(it != overlaps.begin())
            std::cout << ",";
        std::cout << *it;
    }
    if(overlaps.empty())
        std::cout << "nil";
    std::cout << std::endl;

    log_file.close();

    return 0;
}
```

ПРИЛОЖЕНИЕ В
Файл cyclic_shift.cpp

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

std::ofstream log_file;

auto prefix(const std::string& image){
    if(log_file.is_open()){
        log_file << "prefix function initiated for \"" << image
                 << "\" pattern" << std::endl;
    }

    std::vector<int> pi;
    pi.push_back(0);
    for(size_t i = 1, j = 0; i < image.size(); i++){
        if(image[i] != image[j]){
            if(j == 0){
                pi.push_back(0);
            }
            else{
                j = pi[j-1];
                i--;
            }
        }
        else{
            pi.push_back(j+1);
            j++;
        }
    }

    if(log_file.is_open()){
        log_file << "prefix function is:";
        for(auto i: pi)
            log_file << " " << i;
        log_file << std::endl << std::endl;
    }

    return pi;
}


void logger(std::ofstream &log,
            const std::string& shifted,
            const std::string& base,
            size_t i, size_t j)
{
    std::string border;
    border.assign(shifted.size()*2-1, '-');
    log << border << std::endl;

    std::string car_top;
    std::string car_bot;
    car_top.assign(i, ' ');
    car_bot = car_top + "^";
    car_top += "V";
    log << car_top << std::endl;

    std::string haystack = shifted+shifted;
    haystack.pop_back();
    log << haystack << std::endl;

    std::string needle_str;
    needle_str.assign(i-j, ' ');
    needle_str += base;
    log << needle_str << std::endl;
    log << car_bot << std::endl;
    log << border << std::endl;
}


// moves base, until it becomes shifted
auto cyclicShift(const std::string& base, const std::string& shifted){
    std::vector<int> overlaps;
    auto size = base.size();
    if(size != shifted.size()){
        std::cout << "shifted string and base string have different sizes" << std::endl;
        return overlaps;
    }

    if(log_file.is_open()){
        log_file << "Imagine the haystack like concatenation of shifted string with itself, \n"
                 << "and the base string as the needle" << std::endl << std::endl;
    }

    auto pi = prefix(base);

    if(log_file.is_open()){
        log_file << "initial state" << std::endl;
        logger(log_file, shifted, base, 0, 0);
        log_file << std::endl;
    }

    for(size_t i = 0, j = 0; i < size*2-1; i++){
        auto ti = i;
        auto tj = j;
```

```
        if(shifted[i%size] == base[j]){
            j++;
            if(j == size){
                if(log_file.is_open()){
                    log_file << "charachers match,\n"
                             << "pattern carriage equal to pattern size\n"
                             << "lt means, we found the shift of base string" << std::endl;
                    logger(log_file, shifted, base, i, j-1);
                }

                overlaps.push_back(i - j + 1);
                return overlaps;
            }
            if(log_file.is_open())
                log_file << "characters match, move carriage" << std::endl;
        }
        else{
            if(j != 0){
                if(log_file.is_open())
                    log_file << "characters don't match, set pattern carriage on \n"
                             << "prefix function of previous symbol: " << pi[j-1] << std::endl;

                j = pi[j-1];
                i--;
            }
            else{
                if(log_file.is_open())
                    log_file << "mismatch on frist symbol of base string \n"
                             << "move shifted carriage" << std::endl;
            }
        }
        if(log_file.is_open()){
            logger(log_file, shifted, base, ti, tj);
            log_file << std::endl;
        }
    }
    if(log_file.is_open()){
        log_file << "can't move carriage, there is no base in shifted string" << std::endl;
    }
    return overlaps;
}


int main(int argc, char* argv[]){
    if(argc > 1){
        log_file.open(argv[1], std::ios::out | std::ios::trunc);
    }

    std::string shifted;
    std::string base;

    std::cin >> shifted;
    std::cin >> base;

    auto overlaps = cyclicShift(base, shifted);

    for(auto it = overlaps.begin(); it != overlaps.end(); it++){
        if(it != overlaps.begin())
            std::cout << ",";
        std::cout << *it;
    }
    if(overlaps.empty())
        std::cout << "nil";
    std::cout << std::endl;

    log_file.close();

    return 0;
}
```

ПРИЛОЖЕНИЕ С
Файл split_text.cpp

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

std::ofstream log_file;

auto prefix(const std::string& image){
    if(log_file.is_open()){
        log_file << "prefix function initiated for \"" << image
                 << "\" pattern" << std::endl;
    }

    std::vector<int> pi;
    pi.push_back(0);
    for(size_t i = 1, j = 0; i < image.size(); i++){
        if(image[i] != image[j]){
            if(j == 0){
                pi.push_back(0);
            }
            else{
                j = pi[j-1];
                i--;
            }
        }
        else{
            pi.push_back(j+1);
            j++;
        }
    }

    if(log_file.is_open()){
        log_file << "prefix function is:";
        for(auto i: pi)
            log_file << " " << i;
        log_file << std::endl << std::endl;
    }

    return pi;
}


void logger(std::ofstream &log,
            const std::string& haystack,
            const std::string& needle,
            size_t i, size_t j)
{
    std::string border;
    border.assign(haystack.size(), '-');
    log << border << std::endl;

    std::string car_top;
    std::string car_bot;
    car_top.assign(i, ' ');
    car_bot = car_top + "^";
    car_top += "V";
    log << car_top << std::endl;

    log << haystack << std::endl;

    std::string needle_str;
    needle_str.assign(i-j, ' ');
    needle_str += needle;
    log << needle_str << std::endl;
    log << car_bot << std::endl;
    log << border << std::endl;
}


auto KMPA(const std::string& haystack, const std::string& needle){
    auto pi = prefix(needle);
    std::vector<int> overlaps;
    if(needle.size() > haystack.size()){
        std::cout << "haystack bigger than needle" << std::endl;
        return overlaps;
    }

    if(log_file.is_open()){
        log_file << "initial state" << std::endl;
        logger(log_file, haystack, needle, 0, 0);
        log_file << std::endl;
    }

    for(size_t i = 0, j = 0; i < haystack.size(); i++){
        auto ti = i;
        auto tj = j;

        if(haystack[i] == needle[j]){
            j++;
            if(j == needle.size()){
                if(log_file.is_open())
                    log_file << "pattern carriage equal to pattern size\n"
                             << "it means, we found pattern in text" << std::endl;

                overlaps.push_back(i - j + 1);
                j = pi[j-1];
            }
```

```
            if(log_file.is_open())
                log_file << "characters match, move carriage" << std::endl;
        }
        else{
            if(j != 0){
                if(log_file.is_open())
                    log_file << "characters don't match, set pattern carriage on \n"
                             << "prefix function of previous symbol: " << pi[j-1] << std::endl;
                j = pi[j-1];
                i--;
            }
            else{
                if(log_file.is_open())
                    log_file << "mismatch on first symbol of pattern,\n"
                             << "move haystack carriage" << std::endl;
            }
        }

        if(log_file.is_open()){
            logger(log_file, haystack, needle, ti, tj);
            log_file << std::endl;
        }
    }
    if(log_file.is_open()){
        log_file << "can't move carriage, end of string" << std::endl;
    }
    return overlaps;
}


class TextPart{
    friend class Splitter;

    const std::string _str;
    const size_t _index;

    TextPart(const std::string& str, const size_t index):
        _str(str), _index(index){}

public:
    TextPart(const TextPart& other):
        _str(other._str), _index(other._index)
    {}

    const std::string& str() const
    { return _str; }

    const size_t& index() const
    { return _index; }

};

class Splitter
{
public:
    enum Error
    {
        absent,
        some_parts_excluded,
        cannot_be_splitted
    };

private:
    std::vector<TextPart> parts;
    Error _error;


public:
    Splitter(const std::string& text, size_t parts_count, size_t minimal_part_size = 0):
        _error(Error::absent)
    {
        if(log_file.is_open()){
            log_file << "splitter initiated" << std::endl;
        }

        // empty text cannot be splitted
        if(text.empty())
        {
            if(log_file.is_open()){
                log_file << "splitter error" << std::endl;
            }

            _error = Error::cannot_be_splitted;
            return;
        }

        // cannot split, if parts count more than characters in text
        if(text.size() < parts_count)
        {
            // weird error
            if(log_file.is_open()){
                log_file << "splitter error: too many parts" << std::endl;
            }

            _error = Error::cannot_be_splitted;
            return;
        }

        // size of minimal part cannot be smaller than 1
        if(minimal_part_size == 0) minimal_part_size = 1;

        size_t part_length = text.size()/parts_count;
```

```
        // if minimal part length smaller than minimum,
        // recalculate parts count using minimum
        if(minimal_part_size > part_length)
        {
            parts_count = text.size()/minimal_part_size;

            // if minimal part bigger than text, text cannot be splitted
            if(parts_count == 0)
            {
                if(log_file.is_open()){
                    log_file << "splitter error: minimal part bigger than whole text" << std::endl;
                }

                _error = Error::cannot_be_splitted;
                return;
            }

            if(log_file.is_open()){
                log_file << "splitter warning: minimal length not suitable for this parts quantity" << std::endl;
            }

            _error = Error::some_parts_excluded;
            part_length = minimal_part_size;
        }

        // need to restribute a residual to all other parts
        size_t residual = text.size()%part_length;

        // connector size. Size of similiar parts of neighbour tokens
        size_t connector = minimal_part_size/2;
        if(minimal_part_size%2) connector++;

        // there is only 1 part
        // so, text cannot be splitted
        if(parts_count == 1)
        {
            if(log_file.is_open()){
                log_file << "splitter error: there can't be only one part" << std::endl;
            }
            _error = Error::cannot_be_splitted;
            return;
        }

        if(log_file.is_open()){
            log_file << "The whole text divided into " << parts_count << ",\n"
                     << "with a remainder of " << residual << std::endl;
            log_file << "A remainder will be distributed among parts" << std::endl;
        }

        auto token_begin = text.begin();
        auto token_end = text.begin() + part_length;
        // get all whole tokens of text
        for(size_t i = 0; i < parts_count; i++)
        {
            std::string token;
            //auto token_begin = text.begin() + i*part_length;
            //auto token_end = text.begin() + (i+1)*part_length;

            if(residual > 0)
            {
                bool extra_residual = (residual%(parts_count - i))?1:0;
                size_t extra = residual/(parts_count - i) + extra_residual;

                if(log_file.is_open()){
                    log_file << "There is " << extra << " characters for " << i+1 << " part" << std::endl;
                }

                token_end += extra;
                residual -= extra;
            }

            token.assign(token_begin, token_end+((i != parts_count-1)?connector:0));
            if(log_file.is_open()){
                log_file << "Considering the overlay on the next part, \n"
                         << "We've got part " << i+1 << ":\n"
                         <<  token << "\n"
                         << "which starts at index " << token_begin - text.begin()
                         << " in the main text" << std::endl;
            }

            TextPart part(token, token_begin - text.begin());
            parts.emplace_back(part);

            token_begin = token_end;
            token_end += part_length;
        }
        if(log_file.is_open()){
            log_file << std::endl;
        }
    }

    Error error() const
    { return _error; }

    const auto& getParts() const
    { return parts; }
};


auto distributeText(const std::vector<TextPart>& parts, const std::string& needle){
    std::vector<int> overlaps;
    size_t i = 1;
    for(auto part: parts){
```

```
        if(log_file.is_open()){
            log_file << "Start KMP for " << i << " part:\n"
                     << part.str() << std::endl
                     << "which has offset: " << part.index() << std::endl;
        }

        auto private_overlaps = KMPA(part.str(), needle);

        if(log_file.is_open()){
            log_file << "positions of needle in this haystack (with global offset):";
        }

        for(auto it = private_overlaps.begin();
            it != private_overlaps.end(); it++)
        {
            if(log_file.is_open()){
                log_file << " " << *it;
            }

            *it += part.index();

            if(log_file.is_open()){
                log_file << "(" << *it << ")";
            }
        }

        if(log_file.is_open()){
            log_file << std::endl;
        }

        overlaps.insert(overlaps.end(), private_overlaps.begin(), private_overlaps.end());
    }
    if(log_file.is_open()){
        log_file << std::endl;
    }

    return overlaps;
}


int main(int argc, char* argv[]){
    if(argc > 1){
        log_file.open(argv[1], std::ios::out | std::ios::trunc);
    }

    std::string text;
    std::string pattern;
    size_t parts_count;
    size_t connector_size;

    std::getline(std::cin, text);
    std::cin >> parts_count;
    std::cin >> pattern;

    Splitter splitter(text, parts_count, pattern.size());
    if(splitter.error() == Splitter::cannot_be_splitted){
        std::cout << "splitter error. check logs" << std::endl;
        return 1;
    }

    for(auto& token: splitter.getParts()){
        std::cout << token.str() << "; " << token.index() << std::endl;
    }

    auto overlaps = distributeText(splitter.getParts(), pattern);

    for(auto o: overlaps){
        std::cout << o << std::endl;
    }

    return 0;
}
```

ПРИЛОЖЕНИЕ D
Makefile

```
CC = g++
CFLAGS = -g -std=c++17

all:
	@echo choose programm [KMP_search, cyclic_shift, split_text]

KMP_search: KMP_search.o
	$(CC) $(CFLAGS) KMP_search.o -o KMP_search

KMP_search.o: KMP_search.cpp
	$(CC) $(CFLAGS) -c KMP_search.cpp


cyclic_shift: cyclic_shift.o
	$(CC) $(CFLAGS) cyclic_shift.o -o cyclic_shift

cyclic_shift.o: cyclic_shift.cpp
	$(CC) $(CFLAGS) -c cyclic_shift.cpp

split_text: split_text.o
	$(CC) $(CFLAGS) split_text.o -o split_text

split_text.o: split_text.cpp
	$(CC) $(CFLAGS) -c split_text.cpp


clean:
	rm -rf *.o KMP_search cyclic_shift
```