



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технический университет имени
Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по рубежному контролю №1
по курсу «Анализ Алгоритмов»

Студент группы ИУ7-54Б

(Подпись, дата)

Спирин М.П.

(Фамилия И.О.)

Преподаватель

(Подпись, дата)

Волкова Л. Л.

(Фамилия И.О.)

Преподаватель

(Подпись, дата)

Строганов Ю. В.

(Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение	3
1 Аналитическая часть	4
1.1 Многопоточность	4
1.2 Поиск подстроки	5
1.3 Методы поиска опечаток	6
1.3.1 Расстояние Левенштейна	6
1.3.2 Расстояние Дамерау-Левенштейна	7
2 Конструкторская часть	9
2.1 Требования к программному обеспечению	9
2.2 Разработка алгоритмов	9
3 Технологическая часть	18
3.1 Средства реализации	18
3.2 Сведения о файлах программы	19
4 Исследовательская часть	27
4.1 Демонстрация работы программы	27
Заключение	28
Список использованных источников	29

Введение

По мере развития вычислительных систем программисты столкнулись с необходимостью производить параллельную обработку данных для улучшения отзывчивости системы и ускорения производимых вычислений. Благодаря развитию процессоров стало возможным использовать один процессор для выполнения нескольких параллельных операций, что дало начало термину "Многопоточность".

Целью данного рубежного контроля является написание ПО, выполняющего поиск всех вхождений подстроки в файл с учетом возможных опечаток, возможное количество которых зависит от длины строки.

Для поставленной цели необходимо выполнить следующие задачи.

- 1) Описать основы распараллеливания вычислений и методы нахождения опечаток в строке.
- 2) Разработать и реализовать многопоточную версию алгоритма поиска всех вхождений подстроки в файл с учетом возможных опечаток.
- 3) Определить средства программной реализации.

1 Аналитическая часть

В данном разделе будет представлена информация о многопоточности и реализуемом алгоритме нахождения всех вхождений подстроки в файле.

1.1 Многопоточность

Многопоточность (англ. *multithreading*) — это способность центрального процессора (ЦП) обеспечивать одновременное выполнение нескольких потоков в рамках использования ресурсов одного процессора [1]. Поток — последовательность инструкций, которая может исполняться параллельно с другими потоками одного и того же породившего их процесса.

Процессом называют программу в ходе своего выполнения [2]. Таким образом, когда запускается программа или приложение, создается процесс. Один процесс может состоять из одного и более потоков. Таким образом, поток является сегментом процесса, сущностью, которая выполняет задачи, стоящие перед исполняемым приложением. Процесс завершается, когда все его потоки заканчивают работу. Каждый поток в операционной системе является задачей, которую должен выполнить процессор. Сейчас на практике большинство процессоров умеют выполнять несколько задач на одном ядре, создавая дополнительные, виртуальные ядра, или же имеют несколько физических ядер. Такие процессоры называются многоядерными.

При написании программы, использующей несколько потоков, следует учесть, что не всегда есть выигрыш от использования нескольких потоков. Необходимо создавать потоки для независимых по данным и выполнять их параллельно, тем самым сокращая общее время выполнения процесса.

Одной из проблем, появляющихся при использовании потоков, является проблема совместного доступа к информации. Фундаментальным ограничением является запрет на запись из двух и более потоков в одну ячейку памяти одновременно.

Из этого следует, что необходим примитив синхронизации обращения к данным — так называемый мьютекс (англ. *mutex* — *mutual exclusion*). Он

может быть захвачен для работы в монопольном режиме или освобожден, чтобы позволить следующему потоку его захватить. Так, если 2 потока одновременно пытаются захватить мьютекс, успевает только один, а другой будет ждать освобождения.

Набор инструкций, выполняемых между захватом и освобождением мьютекса, называется *критической секцией*. Поскольку в то время, пока мьютекс захвачен, остальные потоки, требующие выполнения критической секции для доступа к одним и тем же данным, ждут освобождения мьютекса для его последующего захвата, требуется разрабатывать программное обеспечение таким образом, чтобы критическая секция была минимальной по объему.

1.2 Поиск подстроки

Поиск подстроки в строке — одна из простейших задач поиска информации. Применяется в виде встроенной функции в текстовых редакторах, СУБД, поисковых машинах, языках программирования и т.д.

В данной лабораторной работе проводится распараллеливание алгоритма поиска всех вхождений искомой подстроки в исходном файле с последующей записью номеров строки и столбца первого символа найденного вхождения в результирующий файл.

Так как последовательное чтение из файла быстрее, чем случайное, в параллельной реализации алгоритма будет использоваться 1 читающий поток [1]. Кроме того, параллельное чтение с жесткого диска возможно только при наличии нескольких головок для чтения, что на практике далеко не всегда доступно.

Считанные строки будут равномерно распределены между несколькими потоками, так как их обработка не зависит от других строк. Именно за счет распараллеливания обработки строк возможно получить какой-то выигрыш в данном алгоритме.

Поскольку одновременная запись в файл из множества потоков невозможна, для записи в файл будет также использоваться только 1 поток.

1.3 Методы поиска опечаток

В качестве количества опечаток будет рассматриваться значение редакционного расстояния — метрика, измеряющая разность между двумя последовательностями символов. Для поиска редакционного расстояния будут рассматриваться алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна.

1.3.1 Расстояние Левенштейна

Расстояние Левенштейна — это минимальное количество редакторских операций вставки (I, от англ. insert), замены (R, от англ. replace) и удаления (D, от англ. delete), необходимых для преобразования одной строки в другую. Стоимость зависит от вида операции:

- 1) $w(a, b)$ — цена замены символа a на b ;
- 2) $w(\lambda, b)$ — цена вставки символа b ;
- 3) $w(a, \lambda)$ — цена удаления символа a .

Будем считать стоимость каждой вышеизложенной операции равной 1:

- $w(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;
- $w(\lambda, b) = 1$;
- $w(a, \lambda) = 1$.

Введем понятие совпадения символов — M (от англ. match). Его стоимость будет равна 0, то есть $w(a, a) = 0$.

Введем в рассмотрение функцию $D(i, j)$, значением которой является редакционное расстояние между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M и N соответственно рассчитывается по рекуррентной формуле:

$$D(i, j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min\{D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1) + m(S_1[i], S_2[j])\}, & i > 0, j > 0 \end{cases} \quad (1.1)$$

где сравнение символов строк S_1 и S_2 рассчитывается по формуле:

$$m(a, b) = \begin{cases} 0 & \text{если } a = b, \\ 1 & \text{иначе.} \end{cases} \quad (1.2)$$

1.3.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна, названное в честь ученых Фредерика Дамерау и Владимир Левенштейна, — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к трем базовым операциям добавляется операция транспозиции T (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычислено по рекуррентной формуле:

$$D(i, j) = \begin{cases} 0, & i = 0, j = 0, \\ i, & j = 0, i > 0, \\ j, & i = 0, j > 0, \\ \min\{D(i, j-1) + 1, & \text{если } i > 1, j > 1, \\ D(i-1, j) + 1, & S_1[i] = S_2[j-1], \\ D(i-1, j-1) + m(S_1[i], S_2[j]), & S_1[i-1] = S_2[j], \\ D(i-2, j-2) + 1\}, & (1.3) \\ \min\{D(i, j-1) + 1, & \text{иначе.} \\ D(i-1, j) + 1, \\ D(i-1, j-1) + m(S_1[i], S_2[j])\}, \end{cases}$$

Вывод

В данном разделе была представлена информация о многопоточности и методах поиска количества опечаток.

Так как расстояние Дамерау-Левенштейна учитывает транспозицию, оно описывает большее количество возможных ошибок, возникающих при написании текста, из-за для оценки количества опечаток будет использоваться именно алгоритм поиска расстояния Дамерау-Левенштейна.

2 Конструкторская часть

В данном разделе разработаны схемы реализаций алгоритма поиска всех вхождений подстроки в файле с учетом опечаток.

2.1 Требования к программному обеспечению

К программе предъявлены ряд требований:

- должен присутствовать интерфейс для выбора действий;
- должна работать с нативными потоками;

2.2 Разработка алгоритмов

На рисунках 2.1 – 2.7 приведены схемы многопоточной реализации алгоритма поиска подстроки в файле с учетом опечаток.

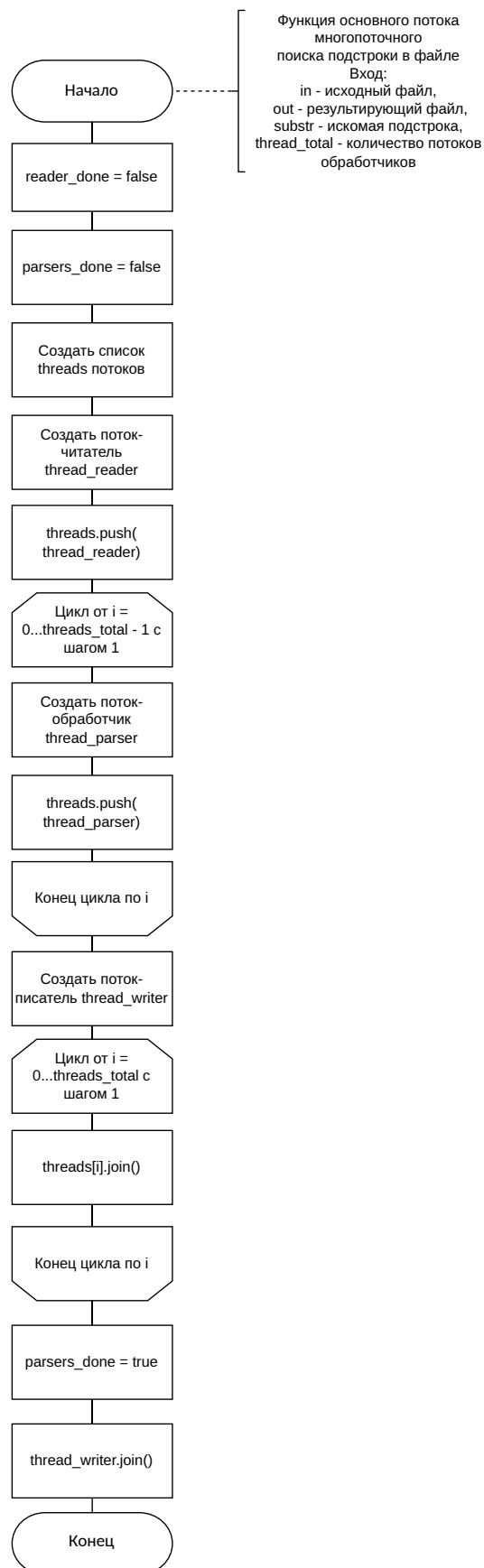


Рисунок 2.1 – Схема алгоритма работы основного потока, запускающего вспомогательные потоки.

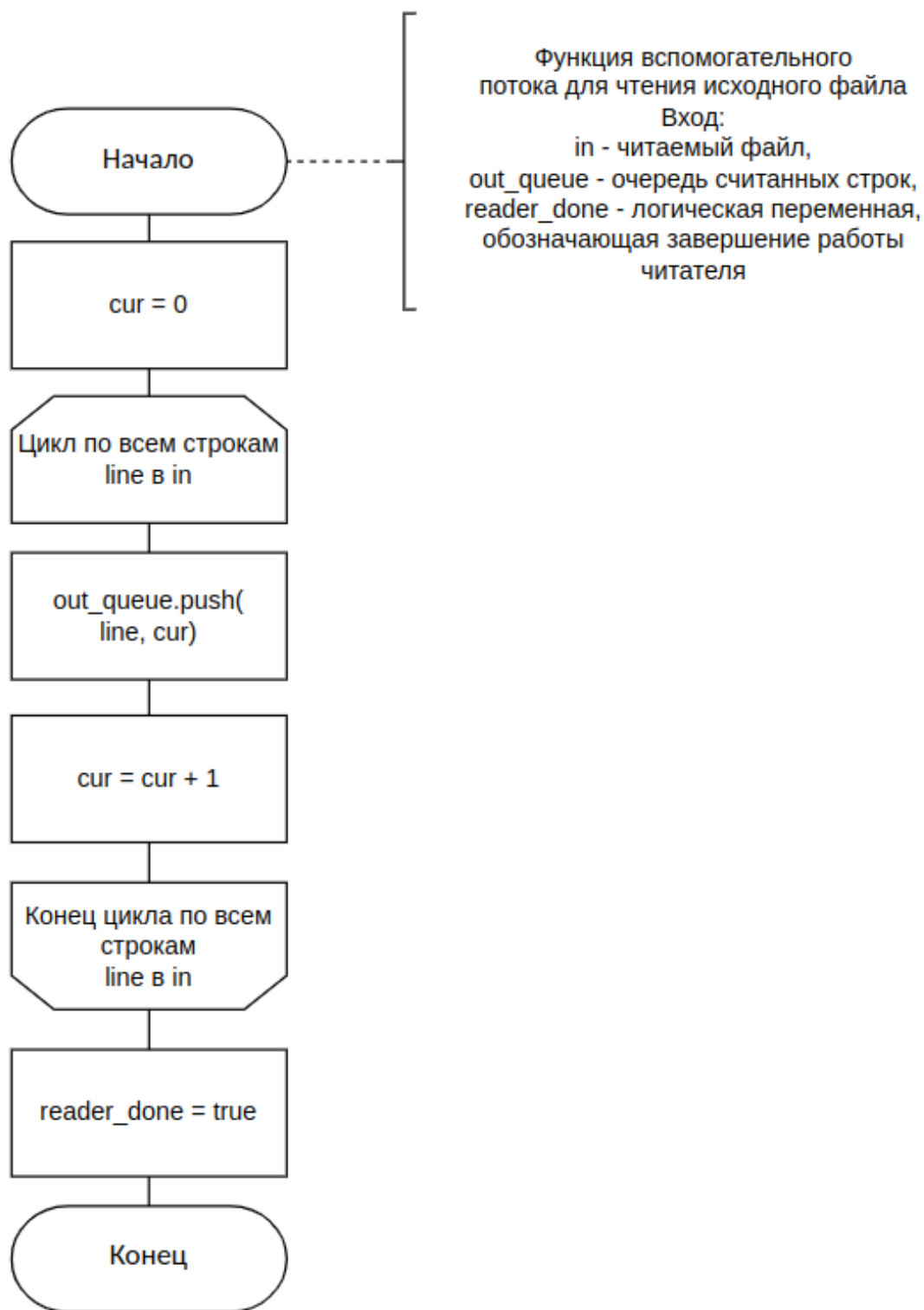


Рисунок 2.2 – Схема алгоритма работы потока, выполняющего чтение из файла.

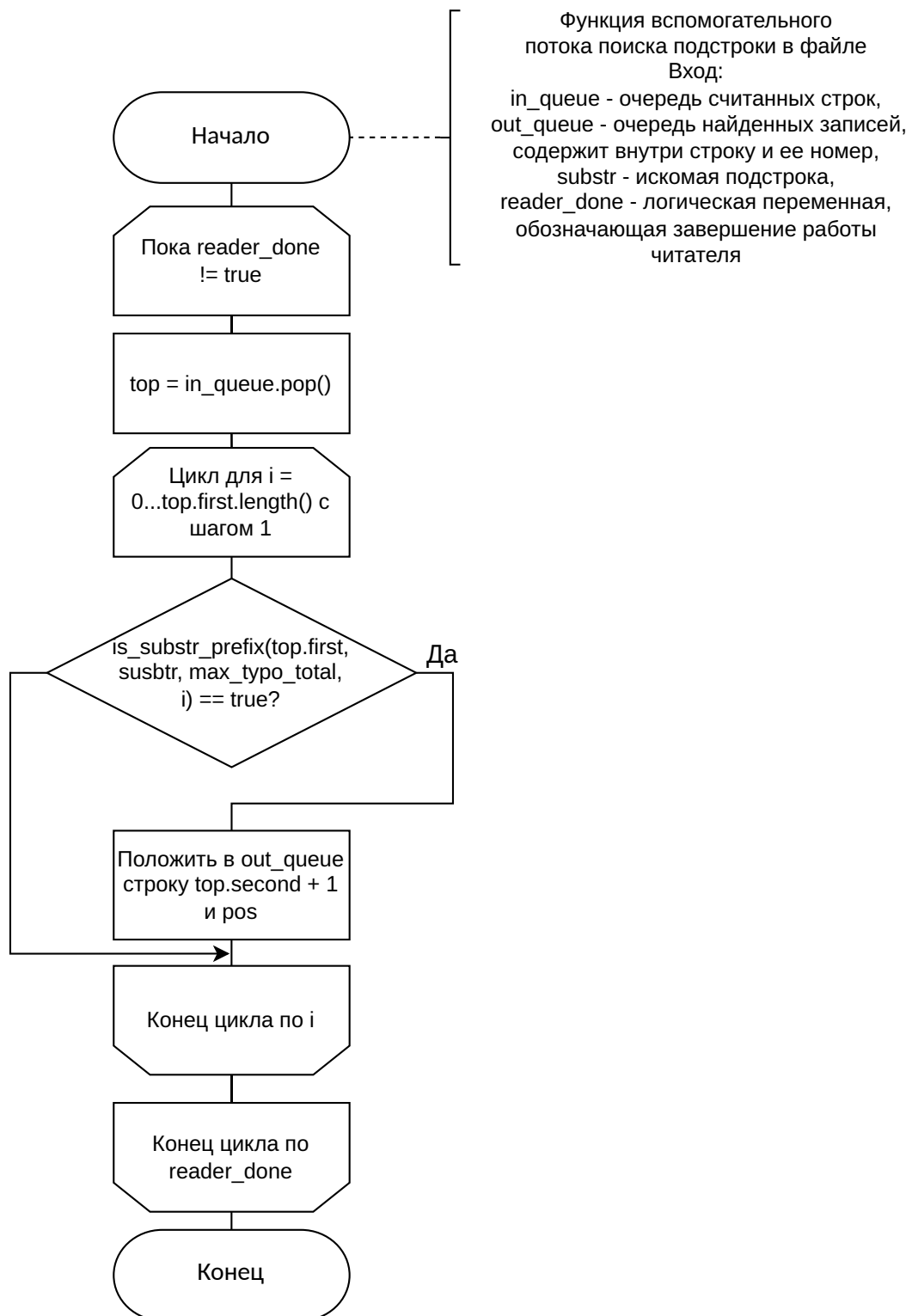


Рисунок 2.3 – Схема алгоритма работы потока, обрабатывающего считанные строки.

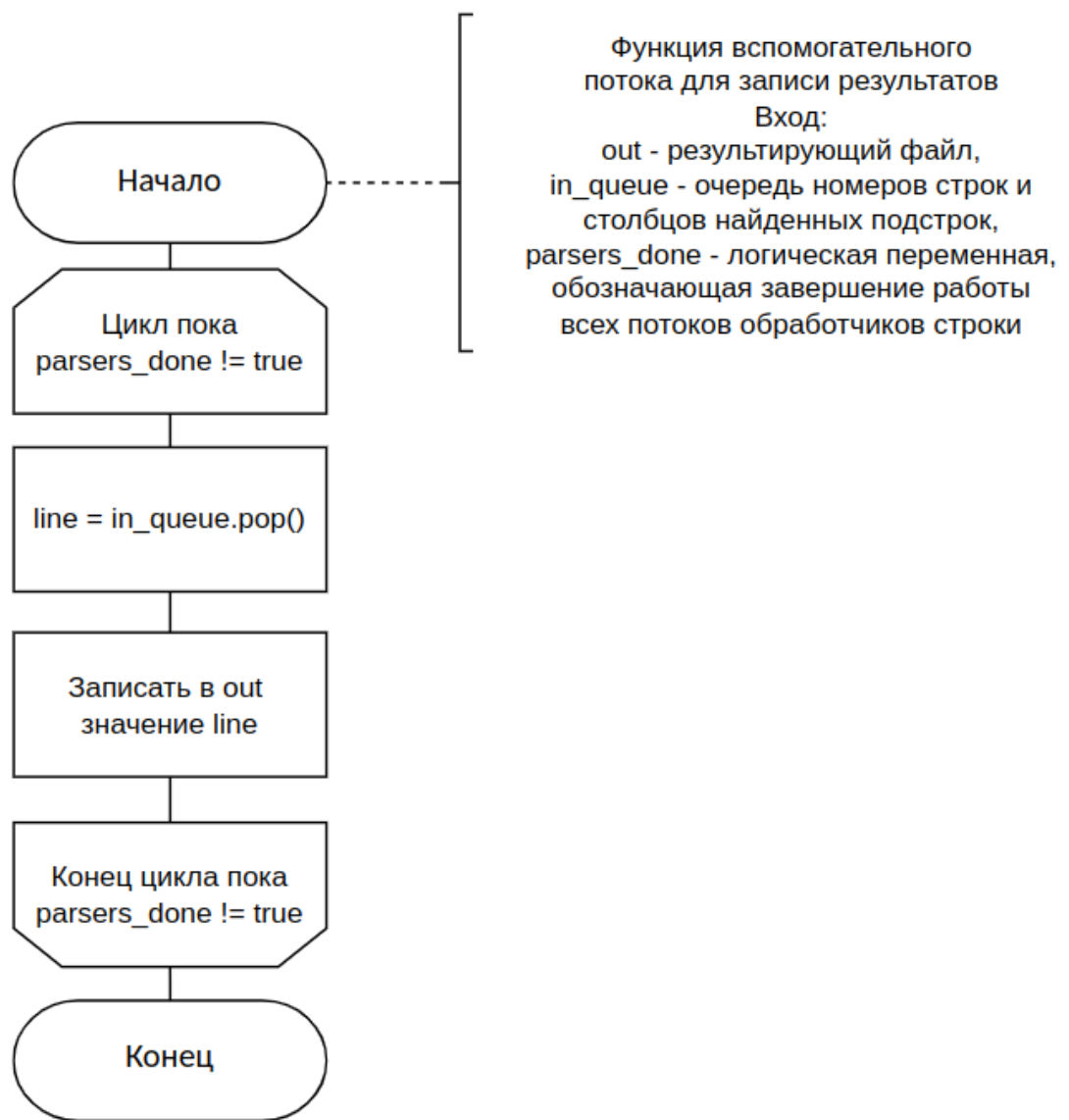


Рисунок 2.4 – Схема алгоритма работы потока, записывающего результаты в файл.

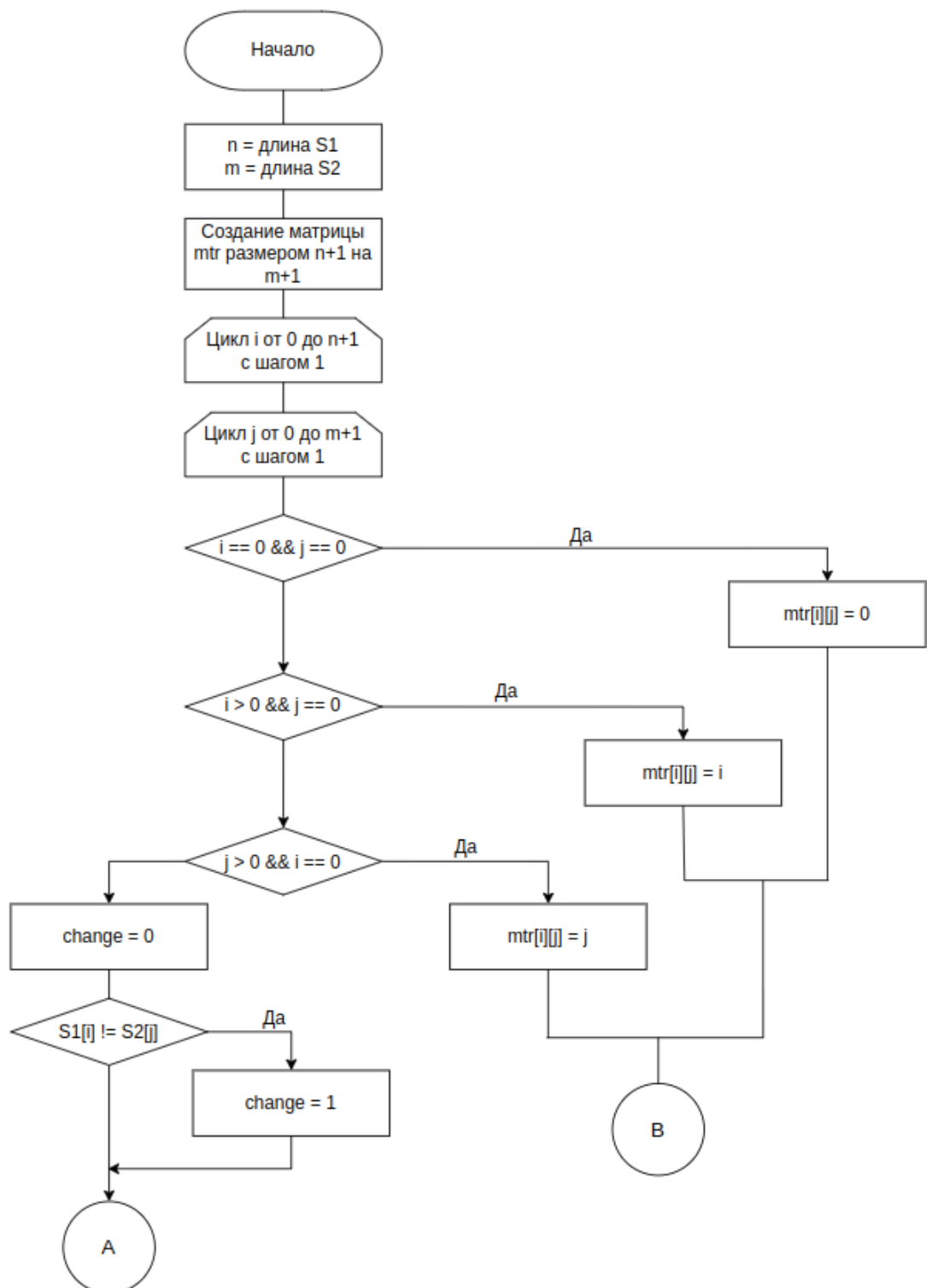


Рисунок 2.5 – Схема 1 нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

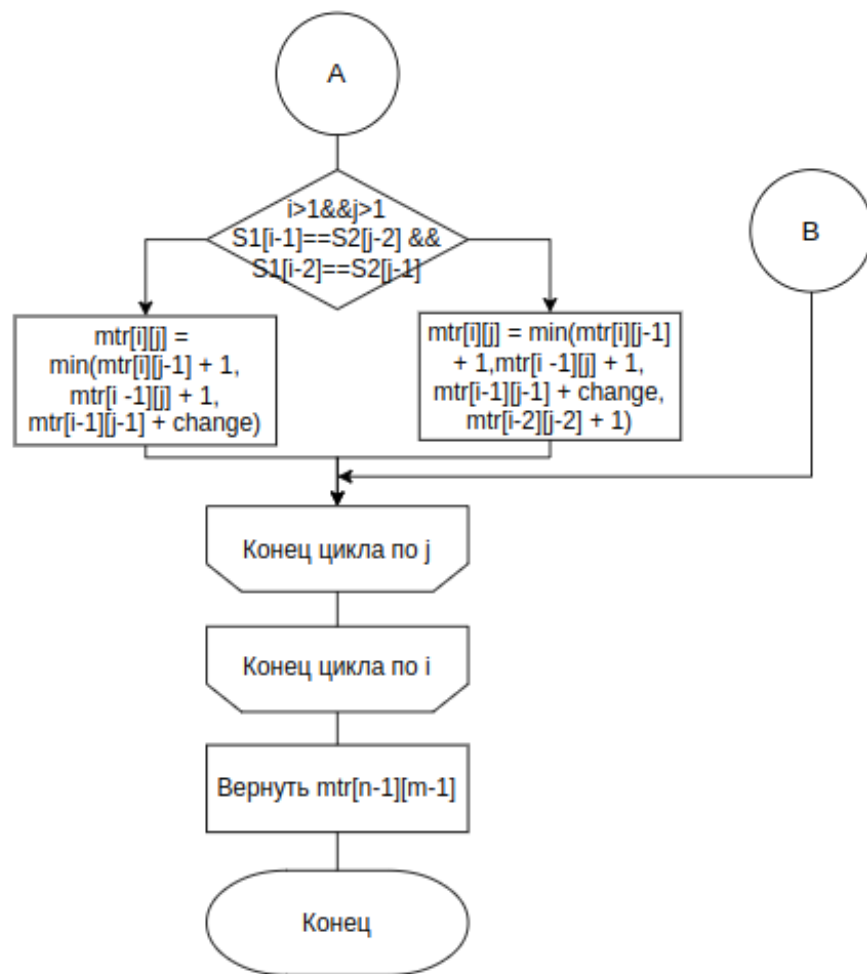


Рисунок 2.6 – Схема 2 итерационного алгоритма нахождения расстояния Дамерау-Левенштейна

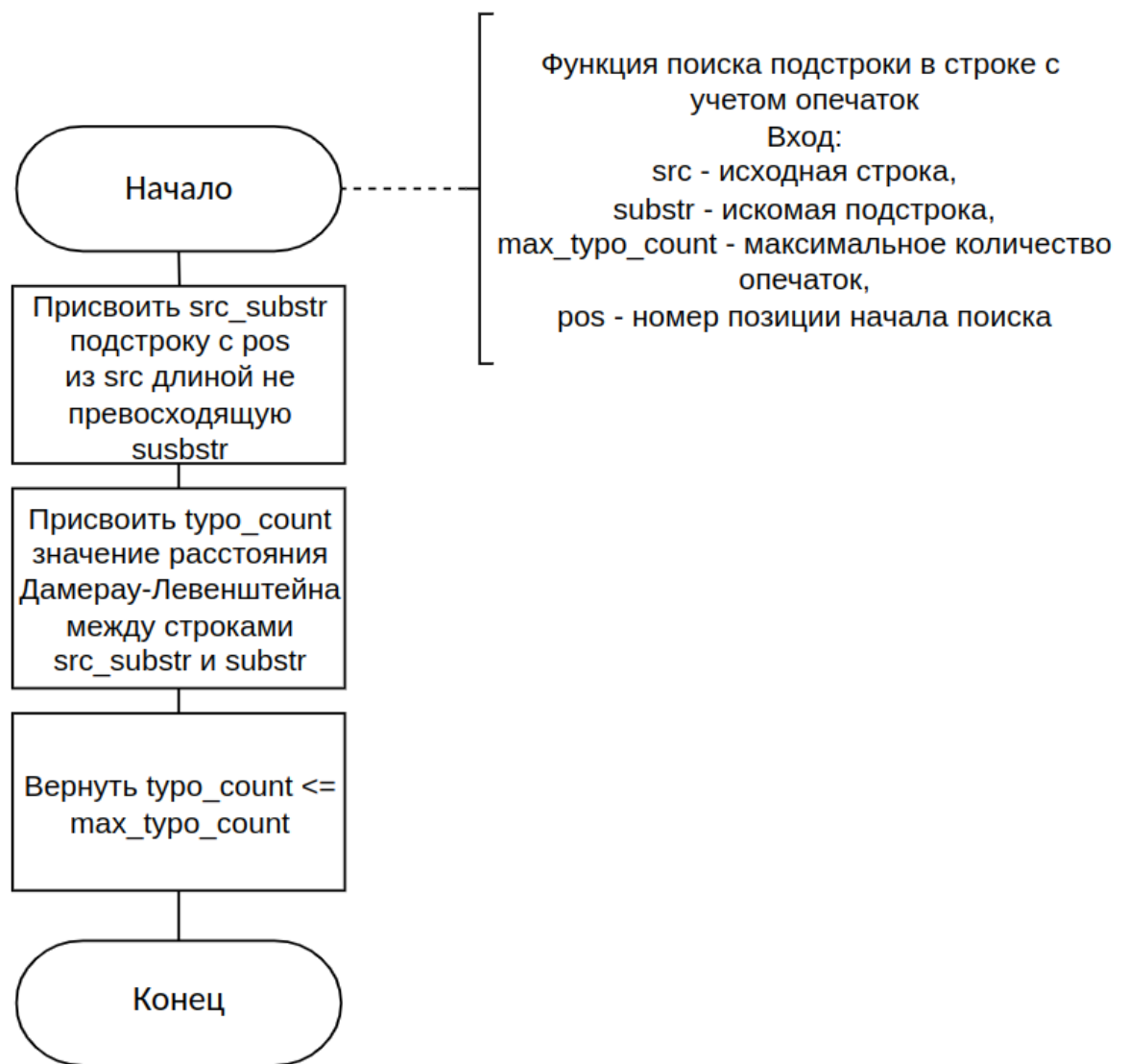


Рисунок 2.7 – Схема функции нахождения подстроки с учетом опечаток

Вывод

В данном разделе разработаны схемы реализаций рассматриваемого алгоритма.

3 Технологическая часть

В данном разделе приведены средства реализации программного обеспечения и листинги кода реализаций алгоритма.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык *C++* [3]. Данный выбор обусловлен наличием у языка встроенной библиотеки измерения процессорного времени и соответствием с выдвинутыми требованиям.

Для измерения времени использовалась функция *clock()* из библиотеки *ctime* [4].

3.2 Сведения о файлах программы

Данная программа разбита на следующие файлы:

- `main.cpp` — файл, содержащий точку входа в программу, в нем происходит общение с пользователем и вызов алгоритмов;
- `src/threads.cpp` — файл содержит функции, реализующие алгоритм поиска подстроки в файле;
- `src/file.cpp` — файл содержит функции для работы с файлами;
- `src/cpu_time.cpp` — файл содержит функции, измеряющие время работы написанных функций.

В листингах 3.1 – 3.6 приведены реализации алгоритма поиска подстроки в файле.

Листинг 3.1 – Функция работы основного потока при многопоточной реализации

```
1 int thread_substring_replace(std::string &in_file , std::string
  &out_file , std::string &substr , size_t thread_total)
2 {
3     reader_done = false;
4     parsers_done = false;
5     std::ifstream in(in_file , std::ios::in);
6     if (!in.is_open())
7     {
8         return 1;
9     }
10    std::ofstream out(out_file);
11    if (!out.is_open())
12    {
13        return 1;
14    }
15    std::string line;
16    ThreadQueue<std::string> out_queue;
17    ThreadQueue<std::pair<std::string , int>> in_queue;
18    std::vector<std::thread> threads;
19    std::thread thread_read(read_thread , std::ref(in) ,
        std::ref(in_queue));
20    threads.push_back(std::move(thread_read));
21    for (int i = 0; i < thread_total; i++)
22    {
23        std::thread thread_obj(thread_substr , std::ref(substr) ,
            std::ref(out_queue) , std::ref(in_queue));
24        threads.push_back(std::move(thread_obj));
25    }
26    std::thread thread_write(write_thread , std::ref(out) ,
        std::ref(out_queue));
27    for (int i = 0; i < threads.size(); i++)
28    {
29        threads[i].join();
30    }
31    {
32        std::lock_guard<std::mutex> _(parsers_done_mutex);
33        parsers_done = true;
34    }
35    thread_write.join();
```

```
36     in.close();  
37     out.close();  
38     return 0;  
39 }
```

Листинг 3.2 – Функция работы потока-читателя

```
1 void read_thread(std::ifstream &in ,  
    ThreadQueue<std::pair<std::string , int>> &out_queue)  
2 {  
3     if (!in.is_open())  
4     {  
5         return;  
6     }  
7  
8     std::string line;  
9     int cur = 0;  
10    while (getline(in , line))  
11    {  
12        out_queue.push(std::pair<std::string , int>(line , cur));  
13        cur++;  
14    }  
15    std::unique_lock<std::shared_mutex> r(reader_done_mutex);  
16    reader_done = true;  
17    return;  
18 }
```

Листинг 3.3 – Функция работы потока-писателя

```
1 void write_thread(std::ofstream &out, ThreadQueue<std::string>
  &in_queue)
2 {
3     if (!out.is_open())
4     {
5         return;
6     }
7
8     std::string line;
9     while (true)
10    {
11        if (in_queue.empty())
12        {
13            std::lock_guard<std::mutex> _(parsers_done_mutex);
14            if (parsers_done)
15                return;
16            continue;
17        }
18        line = in_queue.pop();
19        out << line;
20
21    }
22    return;
23 }
```

Листинг 3.4 – Функция работы потока

```

1 void thread_substr(std::string &substr,
  ThreadQueue<std::string> &out_queue,
  ThreadQueue<std::pair<std::string, int>> &in_queue)
2 {
3     std::pair<std::string, int> top;
4     int max_typo_total = calculate_k(substr.length());
5
6     while (true)
7     {
8         try
9         {
10             top = in_queue.pop();
11         }
12         catch (...)
13         {
14             std::shared_lock<std::shared_mutex>
15                 r(reader_done_mutex);
16             if (reader_done)
17             {
18                 return;
19             }
20             continue;
21         }
22         for (int i = 0; i < top.first.length(); i++)
23         {
24             if (is_substr_prefix(top.first, substr,
25                                 max_typo_total, i))
26             {
27                 std::ostringstream stream;
28                 stream << top.second + 1 << " " << i << "\n";
29                 out_queue.push(stream.str());
30             }
31         }
32 }

```

Листинг 3.5 – Функция поиска подстроки

```

1 bool is_substr_prefix(std::string &src, std::string &substr,
  int max_typo_count, size_t pos)

```



```

2 {
3     std::string src_substr = src.substr(pos,
4         std::min(src.length() - pos, substr.length()));
5     int typo_count = damerau_levenshtein_matrix(src_substr,
6         substr);
7     if (typo_count > max_typo_count)
8     {
9         return false;
10    }
11    return true;
12 }

```

Листинг 3.6 – Функция поиска расстояния Дамерау-Левенштейна

```

1 int damerau_levenshtein_matrix(std::string &first_word,
2     std::string &second_word)
3 {
4     int first_length = first_word.length();
5     int second_length = second_word.length();
6
7     int **matrix = alloc_matrix(first_length + 1, second_length
8         + 1);
9
10    for (int i = 0; i < first_length + 1; i++)
11    {
12        for (int j = 0; j < second_length + 1; j++)
13        {
14            if (i == 0 || j == 0)
15            {
16                matrix[i][j] = abs(i - j);
17            }
18            else
19            {
20                int equal = 1;
21                if (first_word[i - 1] == second_word[j - 1])
22                    equal = 0;
23                matrix[i][j] = std::min({matrix[i - 1][j] + 1,
24                    matrix[i][j - 1] + 1,
25                    matrix[i - 1][j - 1] +
26                        equal});
27                if (i > 1 && j > 1 && first_word[i - 2] ==

```

```

26         second_word[j - 1] &&
27         first_word[i - 1] == second_word[j - 2])
28     {
29         matrix[i][j] = std::min(matrix[i][j],
30                                 matrix[i - 2][j -
31                                     2] + 1);
32     }
33 }
34 int res = matrix[first_length][second_length];
35 free_matrix(matrix, first_length + 1);
36
37 return res;
38 }

```

Вывод

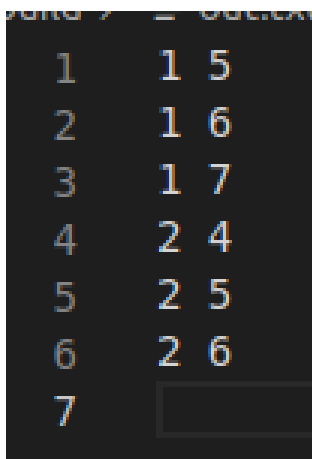
В данном разделе были рассмотрены средства реализации, а также представлен листинг реализации алгоритма поиска подстроки в файле с учетом опечаток.

4 Исследовательская часть

В данном разделе приведены примеры работы реализации алгоритма поиска подстроки в файле с учетом опечаток.

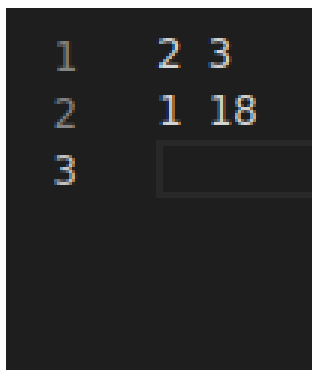
4.1 Демонстрация работы программы

В исходном файле лежит запись "Hello world, it's me!\nI am wordle."



1	1	5
2	1	6
3	1	7
4	2	4
5	2	5
6	2	6
7		

Рисунок 4.1 – Пример №1, поиск подстроки word



1	2	3
2	1	18
3		

Рисунок 4.2 – Пример №2, поиск подстроки m

Заключение

Цель, поставленная в начале работы, была достигнута. Кроме того были достигнуты все поставленные задачи.

- 1) Описаны основы распараллеливания вычислений и методы нахождения опечаток в строке.
- 2) Разработаны и реализованы многопоточную версию алгоритма поиск всех вхождений подстроки в файл с учетом возможных опечаток.
- 3) Определены средства программной реализации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Stoltzfus Justin. Multithreading. — Режим доступа: <https://www.techopedia.com/definition/24297/multithreading-computer-architecture> (дата обращения: 28.01.2023).
2. У. Ричард Стивенс Стивен А. Раго. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018. — С. 994.
3. Документация по Microsoft C++ [Электронный ресурс]. — Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017> (дата обращения: 25.09.2023).
4. C library function clock() [Электронный ресурс]. — Режим доступа: https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm (дата обращения: 25.09.2023).