



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 лабораторной работе №5

По курсу: «Анализ алгоритмов»

Тема: «Конвейерная обработка данных»

Студент:

Пронин А. С.

Группа:

ИУ7-52Б

Преподаватель:

Волкова Л. Л.

Оценка:

Москва

2021

Содержание

Введение	3
1 Аналитический раздел	4
1.1 Решаемая конвейером задача	4
1.2 Вывод	5
2 Конструкторский раздел	6
2.1 Схемы алгоритмов	6
2.2 Вывод	12
3 Технологический раздел	13
3.1 Требование к ПО	13
3.2 Выбор инструментов	13
3.3 Реализация алгоритмов	14
3.4 Тестирование	17
3.5 Вывод	18
4 Исследовательский раздел	19
4.1 Технические характеристики	19
4.2 Результат работы программы	19
4.3 Вывод	21
Заключение	22
Список использованных источников	23

Введение

Цель работы – изучить муравьиный алгоритм на материале решения задачи коммивояжёра.

Задачи работы:

- описать методы решения;
- описать реализацию, реализовать метод;
- выбрать класс данных, составить набор данных;
- провести параметризацию метода на основании муравьиного алгоритма для выбранного класса данных;
- провести сравнительный анализ двух методов;
- дать рекомендации о применимости метода решения задачи коммивояжера на основе муравьиного алгоритма.

1 Аналитический раздел

Задача коммивояжёра — одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город.

Задача коммивояжёра относится к числу транс вычислительных: уже при относительно небольшом числе городов (66 и более) она не может быть решена методом перебора вариантов никакими теоретически мыслимыми компьютерами за время, меньшее нескольких миллиардов лет.

Муравьиный алгоритм — один из эффективных полиномиальных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, ищущих пути от колонии к источнику питания, и представляет собой метаэвристическую оптимизацию.

STOPPED HERE!!!

1.1 Решаемая конвейером задача

В данной лабораторной работе выбран следующий алгоритм для реализации: поиск наибольшего полиндрома в строке. Данный алгоритм можно разбить на 3 этапа:

- разбить строку на слова;
- найти все полиндромы в полученных словах;
- найти наибольший полиндром из всех;

1.2 Вывод

В данном разделе была рассмотрена концепция конвейера и выбран алгоритм для реализации.

2 Конструкторский раздел

В данном разделе представлены схемы алгоритмов работы главного потока и конвейера.

2.1 Схемы алгоритмов

Схема алгоритма работы главного потока представлена на рисунке 2.1. Схема алгоритма работы ленты конвейера представлена на рисунке 2.2. Здесь представлен общий вид схемы, а под обработкой заявки понимаются разные алгоритмы. Лента 1 выполняет алгоритм, представленный на рис. 2.3, лента 2 – алгоритм, представленный на рис. 2.4, лента 3 – алгоритм, представленный на рис. 2.5.

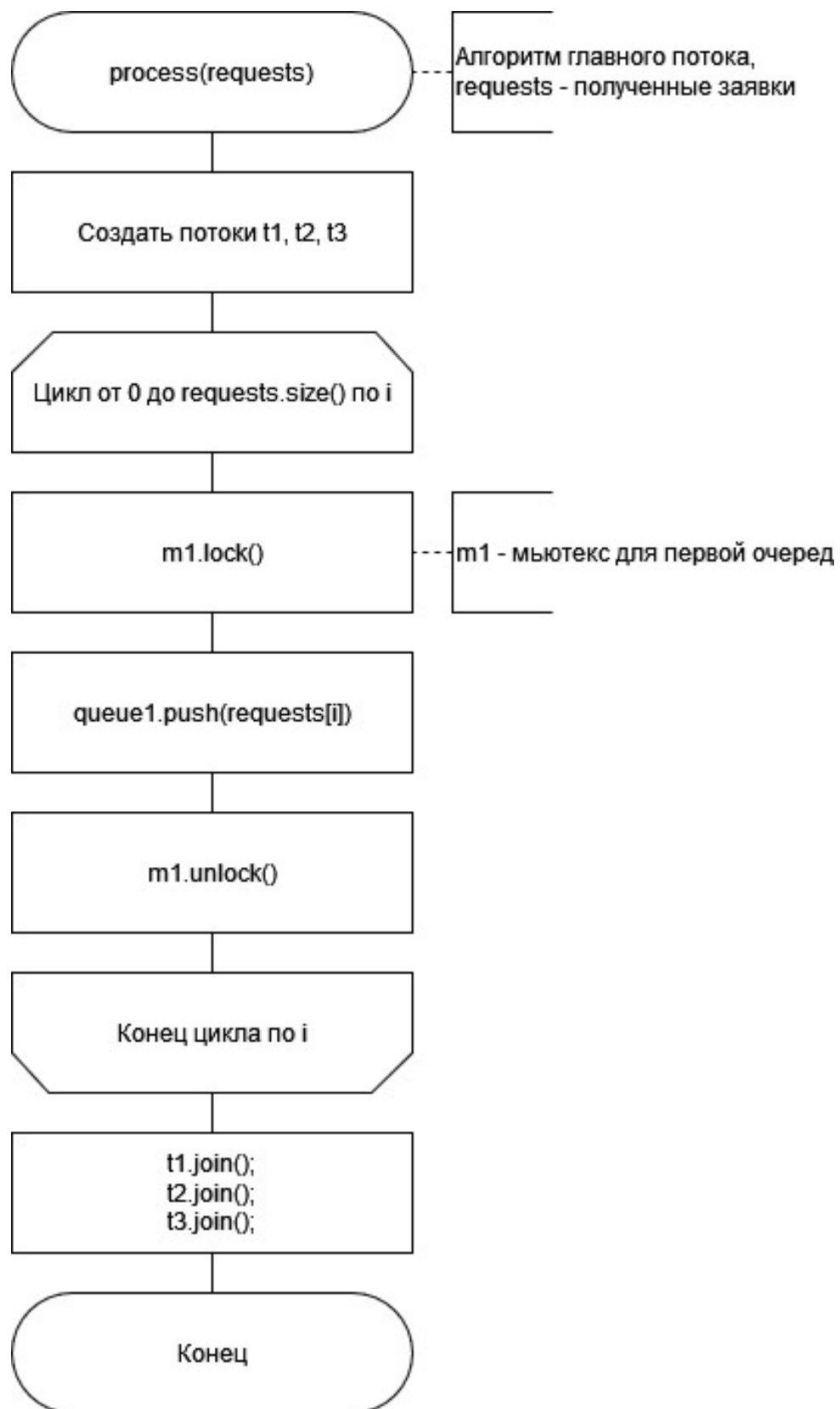


Рис. 2.1: Схема алгоритма работы главного потока

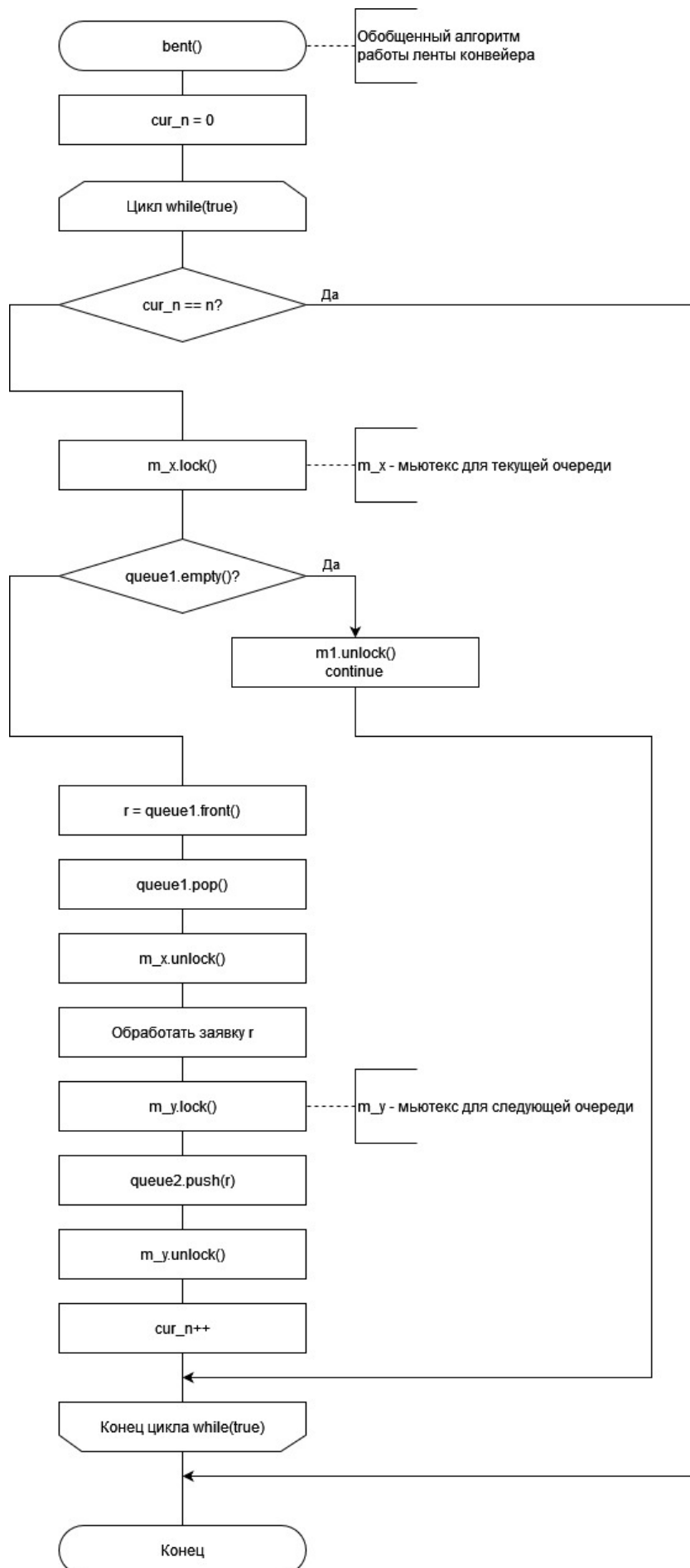


Рис. 2.2: Обобщенная схема алгоритма работы ленты конвейера

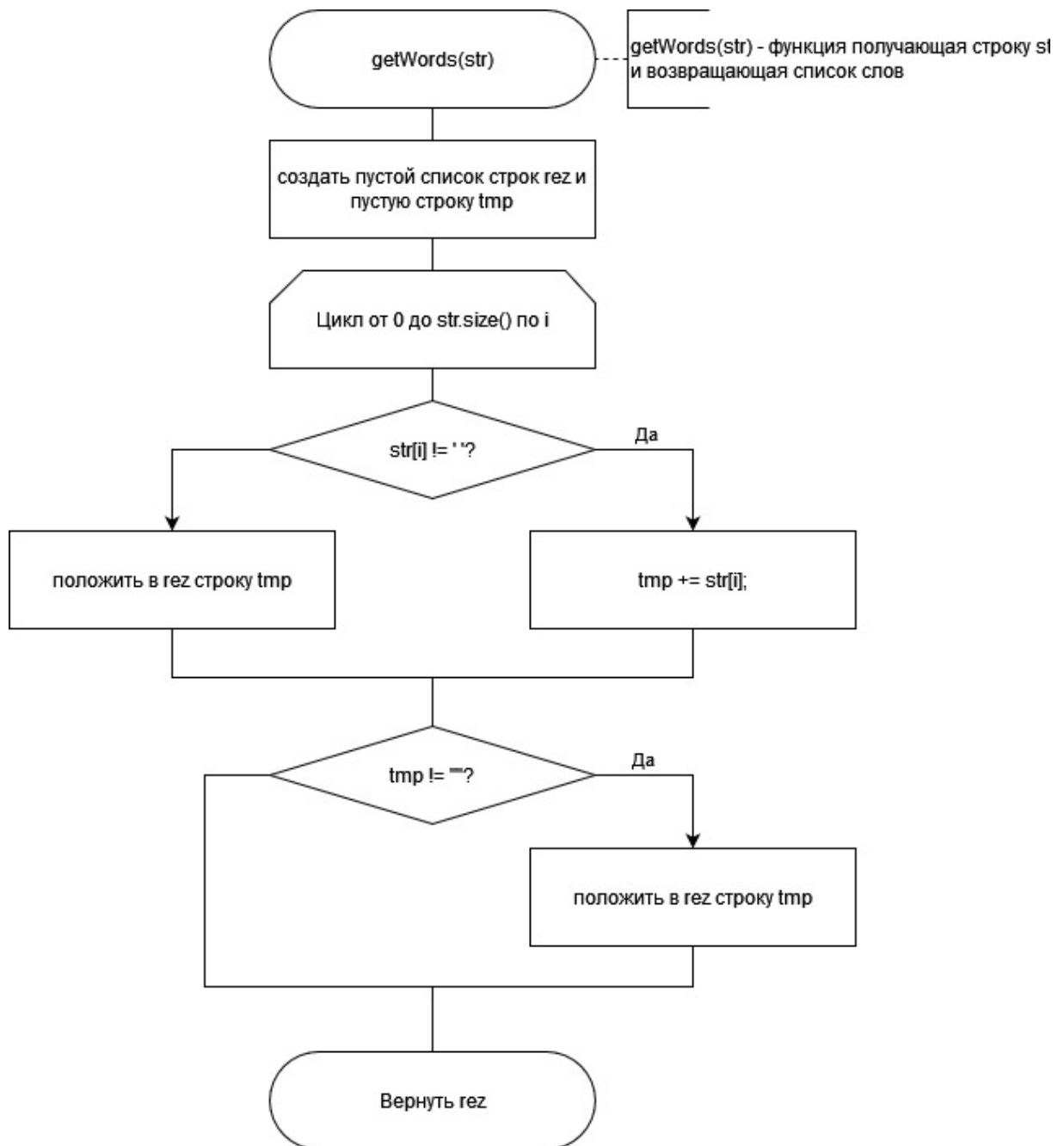


Рис. 2.3: Схема алгоритма разбиения строки на слова

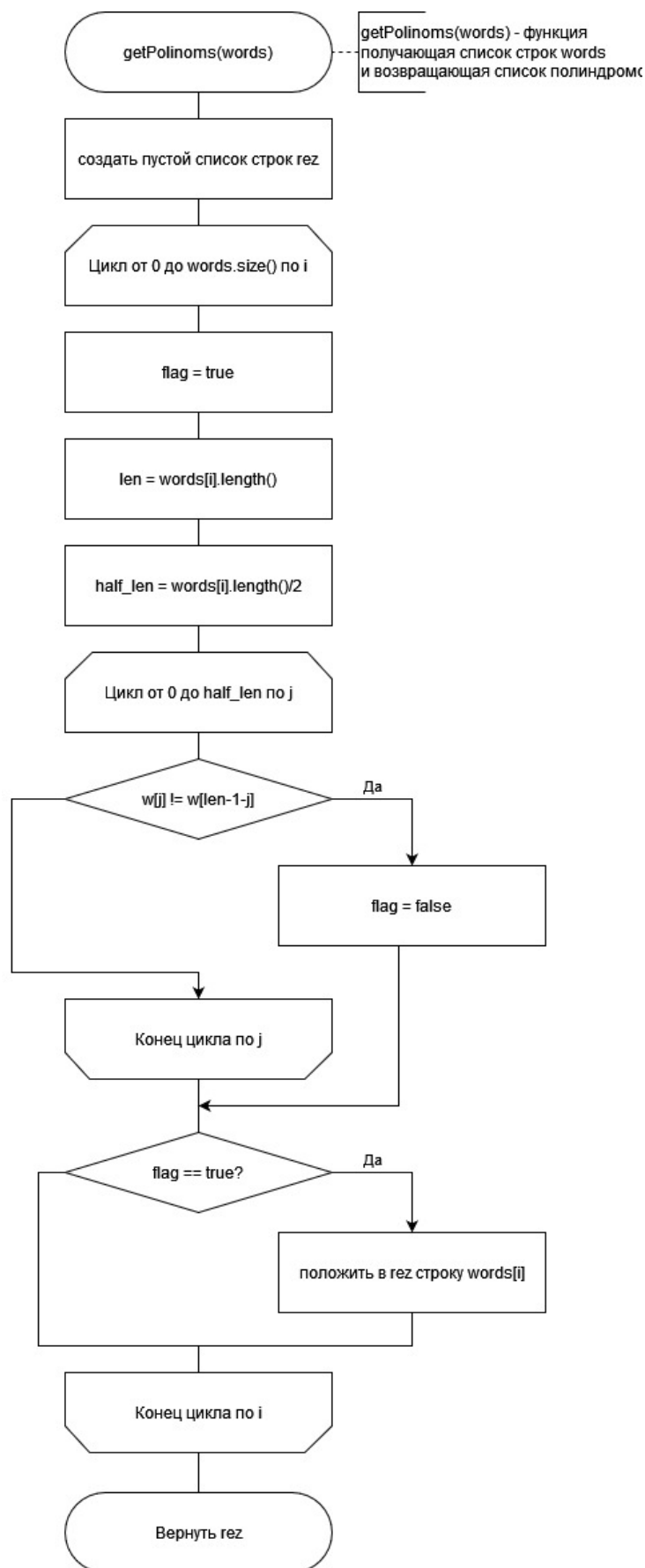


Рис. 2.4: Схема алгоритма нахождения всех полиндромов из полученных слов

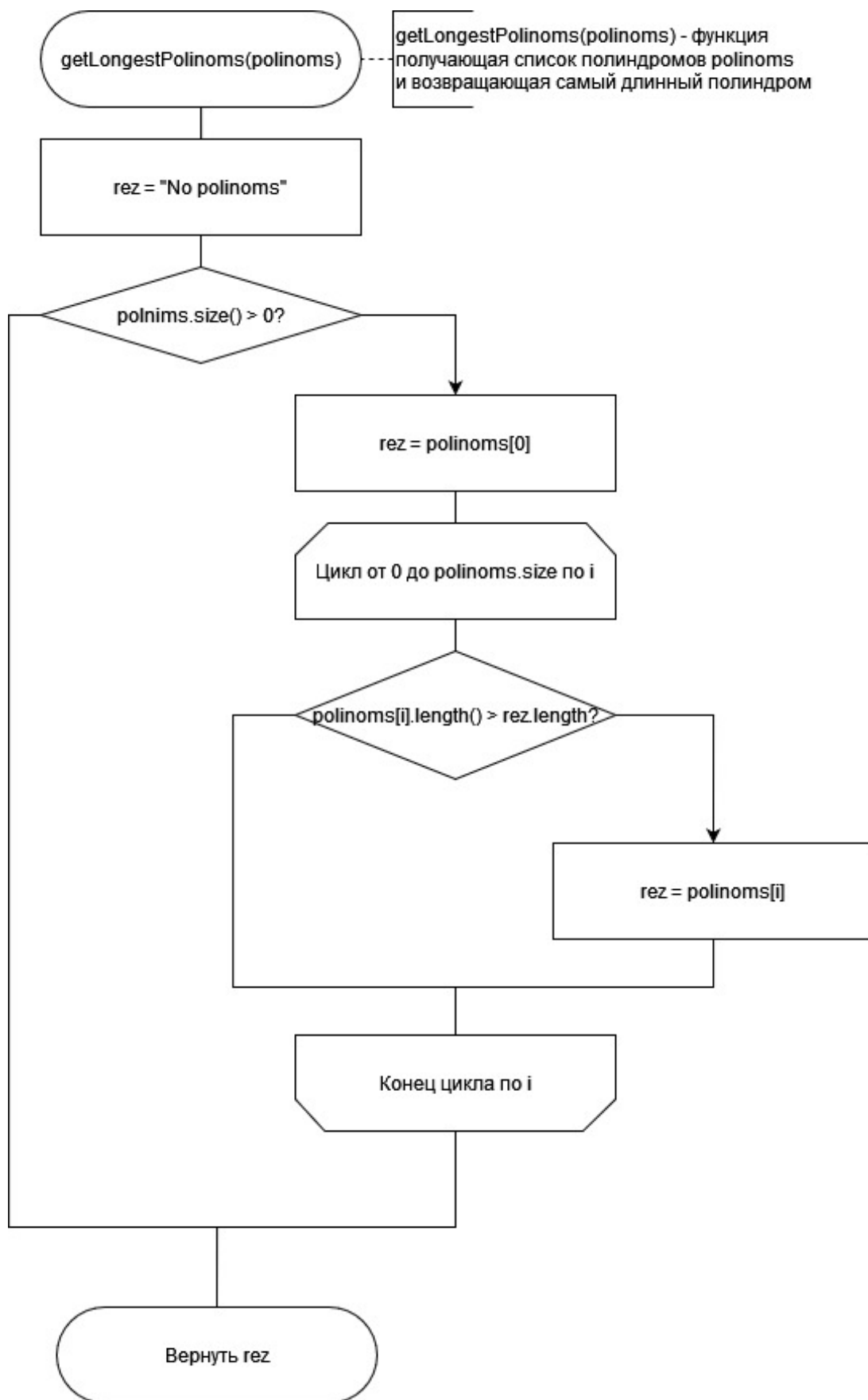


Рис. 2.5: Схема алгоритма самого длинного полиндрома

2.2 Вывод

В данном разделе были разработаны схемы алгоритмов работы главного потока, обобщенная схема алгоритма работы ленты конвейера и схемы алгоритмов обработки заявок.

3 Технологический раздел

В данном разделе представлены требования к ПО, выбор инструментов для реализации и оценки алгоритмов, а также листинги полученного кода.

3.1 Требование к ПО

К программе предъявляется ряд требований:

- на вход подаётся строка;
- на выходе — самый длинный полиндром, если таких несколько, то возвращается первый в строке.

3.2 Выбор инструментов

Поскольку наиболее освоенным языком для разработчика является `c++`, для реализации поставленной задачи был выбран именно он, т.к. таким образом работа будет проделана наиболее быстро и качественно.

Соответственно для компиляции кода будет использоваться компилятор `G++`.

Чтобы оценить время выполнения программы, будет замеряться реальное время, т.к. таким образом можно будет сравнить реализацию с конвейером и без. Для замера реального времени работы программы используется функция `chrono :: high_resolution_clock :: now()` т.к. программа апробируется на компьютере с установленной ОС Windows. [1]

Кроме этого, необходимо отключить оптимизации компилятора для более честного сравнения алгоритмов. В моём случае это делается с помощью ключа `-O0` т.к. используется компилятор `G++`. [2]

3.3 Реализация алгоритмов

На листингах 3.1-3.4 представлены реализации алгоритма работы главного потока и реализации алгоритмов лент конвейера.

Листинг 3.1: Реализация алгоритма работы главного потока

```
1 void Conveyor::process(vector<Request> &requests)
2 {
3     start_time = Clock::now();
4
5     thread t1(&Conveyor::firstBent, this);
6     thread t2(&Conveyor::secondBent, this);
7     thread t3(&Conveyor::thirdBent, this);
8
9     for (size_t i = 0; i < requests.size(); i++)
10    {
11        m1.lock();
12        requests[i].push_time1 = duration_cast<nanoseconds>(Clock::now() -
13            start_time).count()/1000000000.;
14        queue1.push(requests[i]);
15        m1.unlock();
16        Sleep(10);
17    }
18
19    t1.join();
20    t2.join();
21    t3.join();
22 }
```

Листинг 3.2: Реализация алгоритма первой ленты конвейера

```
1 void Conveyor::firstBent()
2 {
3     int cur_n = 0;
4     while (true)
5     {
6         if (cur_n == n)
7             break;
8         m1.lock();
9         if (queue1.empty())
10        {
11            m1.unlock();
12            continue;
13        }
14        Request r = queue1.front();
15        queue1.pop();
16        r.pop_time1 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
17        m1.unlock();
18        r.getWords();
19        m2.lock();
20        r.push_time2 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
21        queue2.push(r);
22        m2.unlock();
23        cur_n ++;
24    }
25 }
```

Листинг 3.3: Реализация алгоритма второй ленты конвейера

```
1 void Conveyor::secondBent()
2 {
3     int cur_n = 0;
4     while (true)
5     {
6         if (cur_n == n)
7             break;
8         m2.lock();
9         if (queue2.empty())
10        {
11            m2.unlock();
12            continue;
13        }
14        Request r = queue2.front();
15        queue2.pop();
16        r.pop_time2 = duration_cast<nanoseconds>(Clock::now() -
17            start_time).count()/1000000000.;
18        m2.unlock();
19        r.getPolinoms();
20        m3.lock();
21        r.push_time3 = duration_cast<nanoseconds>(Clock::now() -
22            start_time).count()/1000000000.;
23        queue3.push(r);
24        m3.unlock();
25        cur_n ++;
26    }
27 }
```


Листинг 3.4: Реализация алгоритма третьей ленты конвейера

```
1 void Conveyor::thirdBent()
2 {
3     int cur_n = 0;
4     while (true)
5     {
6         if (cur_n == n)
7             break;
8         m3.lock();
9         if (queue3.empty())
10        {
11            m3.unlock();
12            continue;
13        }
14        Request r = queue3.front();
15        queue3.pop();
16        r.pop_time3 = duration_cast<nanoseconds>(Clock::now() -
17            start_time).count()/1000000000.;
18        m3.unlock();
19        r.getLongestPolinom();
20        r.processing_time = duration_cast<nanoseconds>(Clock::now() -
21            start_time).count()/1000000000.;
22
23        m4.lock();
24        res.push_back(r);
25        m4.unlock();
26        cur_n++;
27    }
28 }
```

3.4 Тестирование

Для проверки написанных алгоритмов были подготовлены следующие тесты:

- проверка результата обработки строки "test lol 23323232 s sss ll 2332323322 sls sls lls"
- проверка результата обработки строки "test lol 23323232 s sss ll 233232332 sls sls lls"
- проверка результата обработки строки "te str"

Для подготовленных тестов ожидаются следующие результаты соответственно:

- "lol"
- "233232332"
- "No polinoms"

На рисунке 3.1 приведены результаты тестирования. Тестирование проводилось по методу чёрного ящика.

```
Request #1:
str = test lol 23323232 s sss ll 2332323322 sls sll lls
words = test; lol; 23323232; s; sss; ll; 2332323322; sls; sll; lls;
polinoms = lol; s; sss; ll; sls;
longest_polinom = lol

Request #2:
str = test lol 23323232 s sss ll 233232332 sls sll lls
words = test; lol; 23323232; s; sss; ll; 233232332; sls; sll; lls;
polinoms = lol; s; sss; ll; 233232332; sls;
longest_polinom = 233232332

Request #3:
str = te str
words = te; str;
polinoms =
longest_polinom = No polinoms
```

Рис. 3.1: Результаты тестирования

Как видно, все тесты пройдены.

3.5 Вывод

В данном разделе были выдвинуты требования к ПО, выбраны инструменты реализации выбранных алгоритмов, представлены листинги реализованных алгоритмов, а также проведено тестирование.

4 Исследовательский раздел

В данном разделе представлены технические характеристики компьютера, используемого для тестирования и экспериментов, и результат работы программы.

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором были проведены эксперименты при помощи разработанного ПО:

- операционная система: Windows 10 (64-разрядная);
- оперативная память: 32 GB;
- процессор: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz;
- количество ядер: 4;
- количество потоков: 8.

4.2 Результат работы программы

На рисунке 4.1 представлены результаты работы программы для обработки 10 заявок, каждая из которых содержит строку из 10000 слов, состоящих из 1-10 букв.

```

Conveyor time:
194276500 ns
194.276 ms
0.194276 s

```

Request #	push_time1	pop_time1	push_time2	pop_time2	push_time3	pop_time3	proc_time
1	0.002001	0.002001	0.008001	0.009003	0.020000	0.022000	0.036000
2	0.020000	0.020000	0.042000	0.043001	0.052000	0.054001	0.066691
3	0.036000	0.043001	0.056000	0.058002	0.066691	0.068693	0.082699
4	0.052000	0.057003	0.070692	0.071690	0.082699	0.084703	0.098678
5	0.066691	0.071690	0.088702	0.090703	0.098678	0.100678	0.114688
6	0.082699	0.089704	0.102678	0.103678	0.114688	0.116689	0.130705
7	0.098678	0.103678	0.120688	0.122690	0.130705	0.132709	0.146718
8	0.114688	0.121688	0.135708	0.136710	0.146718	0.148719	0.162246
9	0.130705	0.136710	0.150718	0.151718	0.162246	0.164249	0.178259
10	0.146718	0.150718	0.166248	0.168250	0.178259	0.180262	0.193275

```

min_queue_time = 0
max_queue_time = 0.0070042
avg_queue_time = 0.002668

min_proc_time = 0.0339991
max_proc_time = 0.0480402
avg_proc_time = 0.045978

avg_proc1_time = 0.0145242
avg_proc2_time = 0.00972407
avg_proc3_time = 0.0137257

avg_queue1_time = 0.00460116
avg_queue2_time = 0.00140095
avg_queue3_time = 0.0020019

Without conveyor time:
464698900 ns
464.699 ms
0.464699 s

```

Рис. 4.1: Результат работы программы

min_queue_time - минимальное время простоя в очереди.
max_queue_time - максимальное время простоя в очереди.
avg_queue_time - среднее время простоя в очереди.
min_proc_time - минимальное время обработки заявки.
max_proc_time - максимальное время обработки заявки.
avg_proc_time - среднее время обработки заявки.
avg_procX_time - среднее время обработки для X-ой ленты.
avg_queueX_time - среднее время простоя в X-ой очереди.

Из результатов следует, что больше всего времени в среднем тратится на разбиение строки на слова (1-я лента конвейера), меньше всего — на поиск полиндромов среди них (2-я лента конвейера). Соответственно, время простоя в первой очереди — наибольшее, а во второй — наименьшее. Также, реализация с конвейером быстрее справилась с заявками, чем реализация с последовательной обработкой.

4.3 Вывод

По итогу исследования выяснилось, что разработанный алгоритм работает верно, то есть находит самый длинный полиндром в строке. Кроме этого был проведён анализ и сделан вывод по логу программы.

Заключение

По итогу проделанной работы была достигнута цель – получен навык организации асинхронной передачи данных между потоками на примере конвейерной обработки информации.

Также были решены все поставленные задачи, а именно:

- выбраны и описаны методы обработки данных, которые сопоставлены методам конвейера;
- реализована конвейерная система, а также сформирован лог событий с указанием времени их происхождения;
- произведено сравнение реализации с конвейером и без.

Список использованных источников

- [1] MSDN – библиотека <chrono> [Электронный ресурс] // Техническая документация Майкрософт. URL: <https://docs.microsoft.com/ru-ru/cpp/standard-library/chrono?view=msvc-160&viewFallbackFrom=vs-2017> (дата обращения: 07.12.21)
- [2] Как применить настройки оптимизации GCC в QT? [Электронный ресурс] // Блог ПерСа. URL: <http://blog.kislenko.net/show.php?id=1991> (дата обращения: 07.12.21)