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»

на тему: «Конвейерная обработка данных»

Студент ИУ7-53Б (Группа) _____ (Подпись, дата) В. Марченко (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Ю. В. Строганов (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Л. Л. Волкова (И. О. Фамилия)

Москва — 2022 г.

Оглавление

Введение

Параллелизм — это возможность одновременного выполнения более одной арифметико-логической операции или программной ветви [1]. Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность [2].

Параллельная обработка. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что имеется пять таких же независимых устройств, способных работать одновременно и независимо, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично, система из N устройств ту же работу выполнит примерно за $\frac{1000}{N}$ единиц времени [2].

Конвейерная обработка. Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передает результат следующему, одновременно принимая новую порцию входных данных [2].

В рамках данной лабораторной работы конвейерная обработка данных будет исследоваться на материале алгоритмов, осуществляющих поиск строки в тексте.

Формальная постановка задачи.

Даны строка S и текст T. Требуется определить индекс, начиная с которого строка S содержится в тексте T. Если S не содержится в T — вернуть индекс, который не может быть интерпретирован как позиция в тексте (например, отрицательное число). Требуется найти все вхождения строки S в текст T.

1 Аналитическая часть

1.1 Цели и задачи

Цель работы: изучить принципы конвейерной обработки данных и реализовать классический алгоритм поиска строки в тексте и алгоритм Кнута-Морриса-Пратта.

Задачи лабораторной работы:

1) изучить понятие конвейерной обработки данных;
2) реализовать классический алгоритм поиска строки в тексте и алгоритм Кнута-Морриса-Пратта;
3) реализовать последовательную обработку данных на основе двух алгоритмов поиска строки в тексте;
4) реализовать конвейерную обработку данных на основе двух алгоритмов поиска строки в тексте;
5) провести сравнительный анализ времени работы линейной и конвейерной обработки данных на основе экспериментальных данных.

1.2 Классический алгоритм поиска строки в тексте

Первый приходящий в голову алгоритм для поиска строки S в тексте T последовательно проверяет равенство $S[1..m] = T[s + 1..s + m]$ для всех $n - m + 1$ возможных значений s [3].

Можно сказать, что мы двигаем строку вдоль текста и проверяем все ее положения [3].

Время работы этого алгоритма в худшем случае есть $\Theta((n - m + 1)m)$. В самом деле, пусть $T = a^n$ (буква a, повторенная n раз), а $S = a^m$. Тогда для каждой из $n - m + 1$ проверок будет выполнено m сравнений символов, всего $(n - m + 1)m$, что есть $\Theta(n^2)$ (при $m = [n/2]$) [3].

1.3 Алгоритм Кнута-Морриса-Пратта

Расмотрим алгоритм сравнения строк, который был предложен Кнутом, Моррисом и Праттом, и время работы которого линейно зависит от объема входных данных. В этом алгоритме удается избежать вычисления функции переходов δ, а благодаря использованию вспомогательной функции $\pi[1..m]$, которая вычисляется по заданному образцу за время $\Theta(m)$, время сравнения в этом алгоритме равно $\Theta(n)$. Массив π позволяет эффективно вычислять функцию δ «на лету», т. е. по мере необходимости. Грубо говоря, для любого состояния $q = 0, 1, ..., m$ и любого символа $a \in \Sigma$ величина $\pi[q]$ содержит информацию, которая не зависит от символа a и необходима для вычисления величины $\delta(q, a)$. Поскольку массив π содержит только m элементов (в то время как в массиве δ их $\Theta(m|\Sigma|)$, вычисляя на этапе предварительной обработки функцию π вместо функции δ, удается уменьшить время предварительной обработки образца в $|\Sigma|$ раз [3].

Префиксная функция π, предназначенная для какого-нибудь образца, инкапсулирует сведения о том, в какой мере образец совпадает сам с собой после сдвигов. Эта информация позволяет избежать ненужных проверок в простейшем алгоритме поиска подстрок или предвычисления функции δ при использовании конечных автоматов [3].

Дадим формальное определение. Префиксной функцией заданного образца $P[1..m]$ называется функция $\pi : 1, 2, ..., m \to 0, 1, ..., m - 1$, такая что

$$\pi[q] = \max\{k : k < q \text{ и } P_k \supset P_q\}. \tag{1.1}$$

Другими словами, $\pi[q]$ равно длине наибольшего префикса образца P, который является истинным суффиксом строки P_q [3].

1.4 Конвейерная обработка данных

Заявка состоит из текста и строки. Конвейер состоит из трех лент. На первой происходит поиск всех вхождений строки в текст с помощью классического алгоритма поиска строки в тексте. На второй ленте происходит поиск

всех вхождений строки в текст с помощью алгоритма Кнута-Морриса-Пратта. На третьей полученные результаты записываются в журнал (файл). В журнал для каждой заявки необходимо записать строку, количество сравнений и все вхождения строки в текст для обоих алгоритмов.

Вывод из аналитической части

В текущем разделе были рассмотрены два алгоритма поиска строки в тексте — классический и Кнута-Морриса-Пратта — которые будут использоваться соответственно на первом и втором этапах конвейерной обработки данных.

2 Конструкторская часть

2.1 Описание алгоритмов

2.1.1 Алгоритм Кнута-Морриса-Пратта

На рисунке 2.1 показана схема алгоритма заполнения вспомогательного массива.

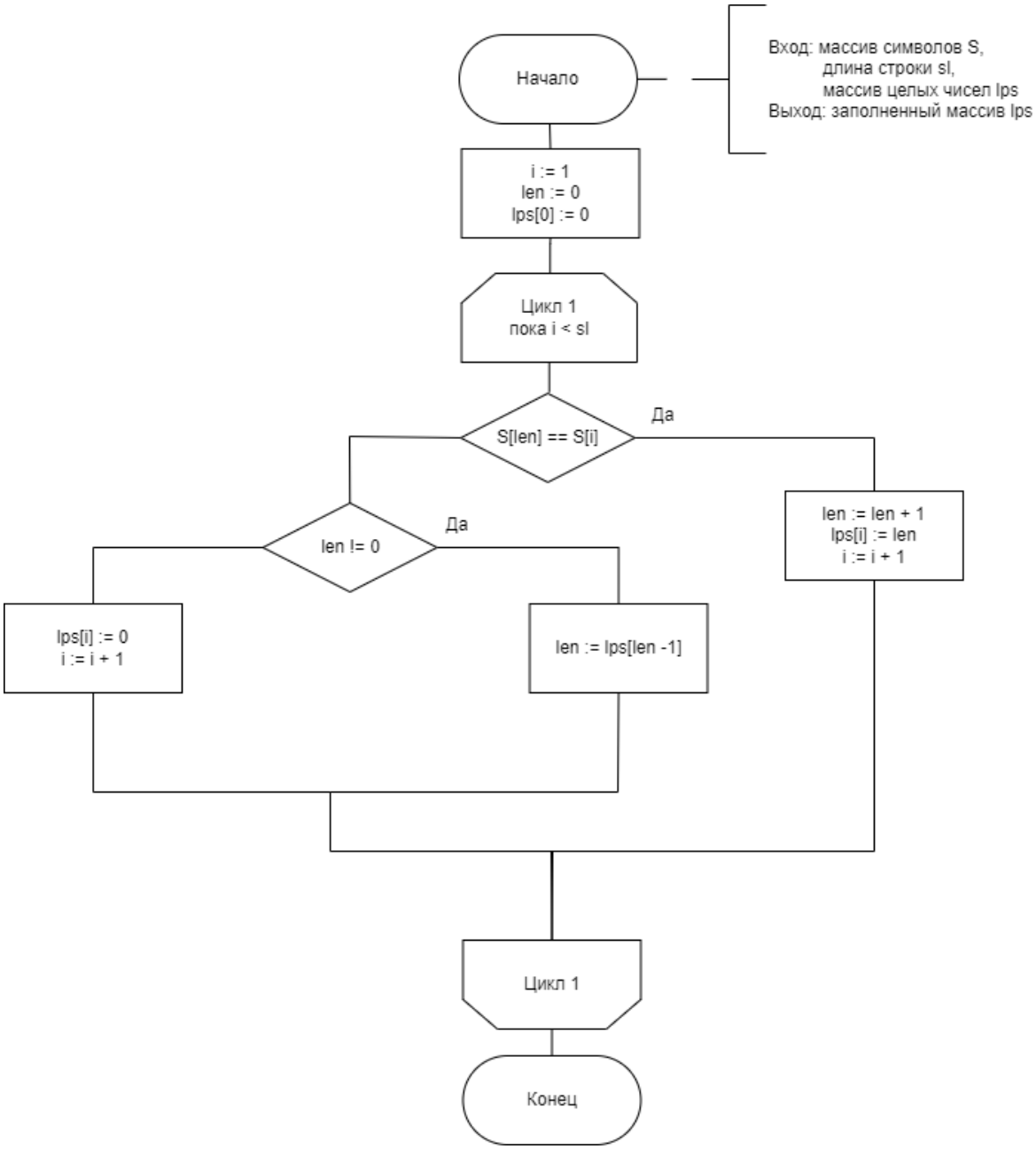

Рисунок 2.1 – Алгоритм заполнения вспомогательного массива

На рисунках 2.2–2.3 показана схема алгоритма Кнута-Морриса-Пратта.

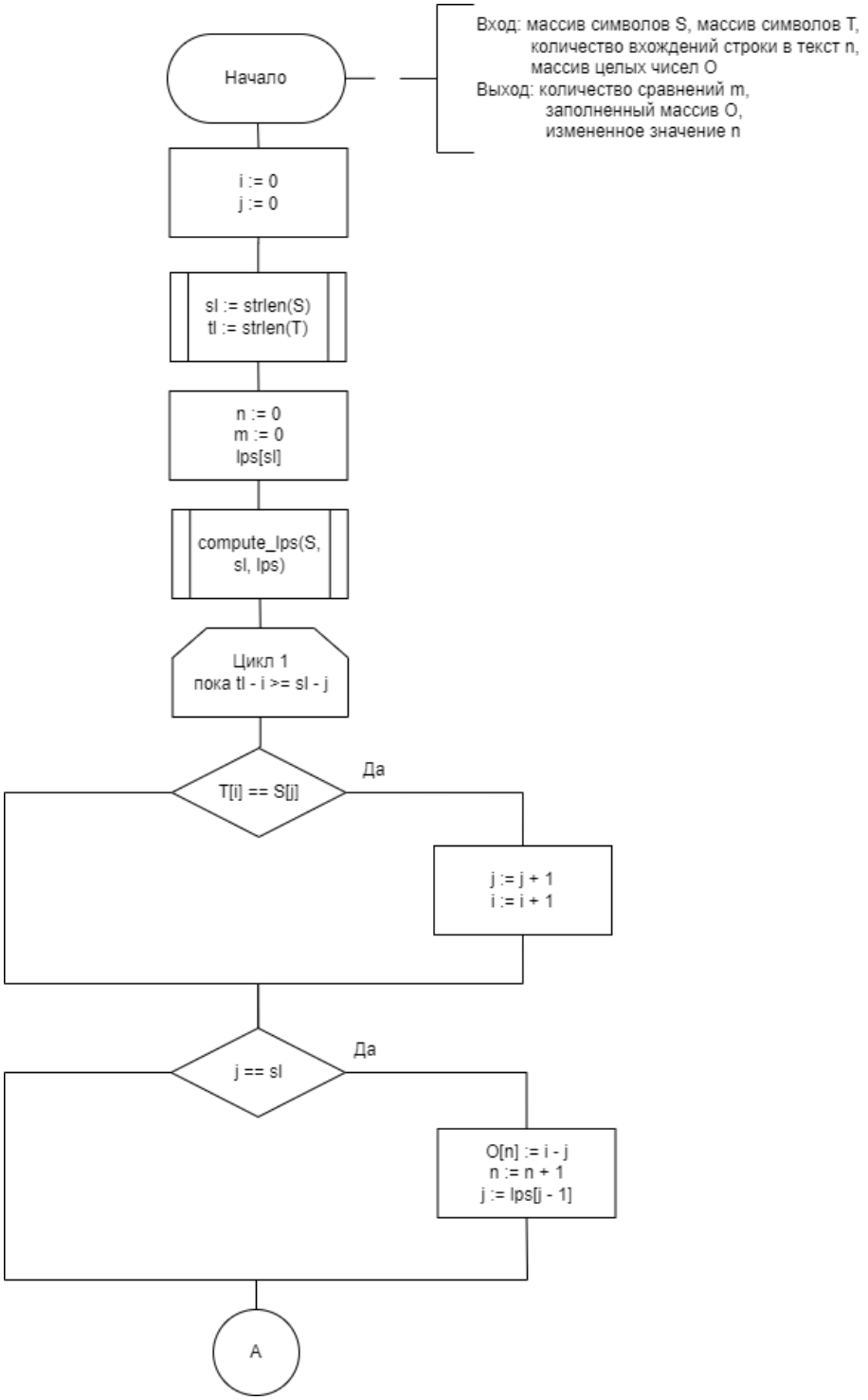

Рисунок 2.2 – Алгоритм Кнута-Морриса-Пратта — ч. 1

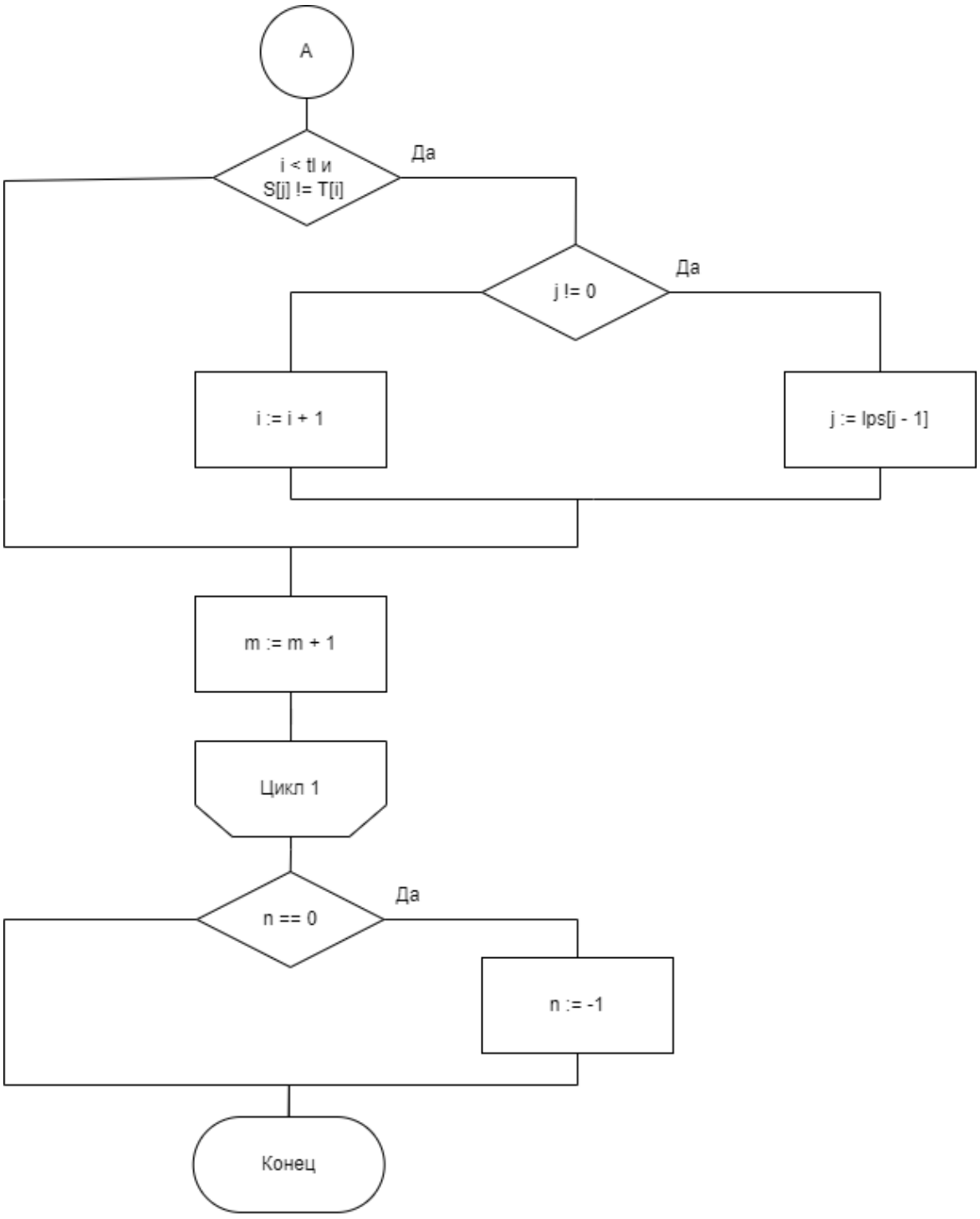

Рисунок 2.3 – Алгоритм Кнута-Морриса-Пратта — ч. 2

2.1.2 Классический алгоритм поиска строки в тексте

На рисунке 2.4 показана схема классического алгоритма поиска строки в тексте.

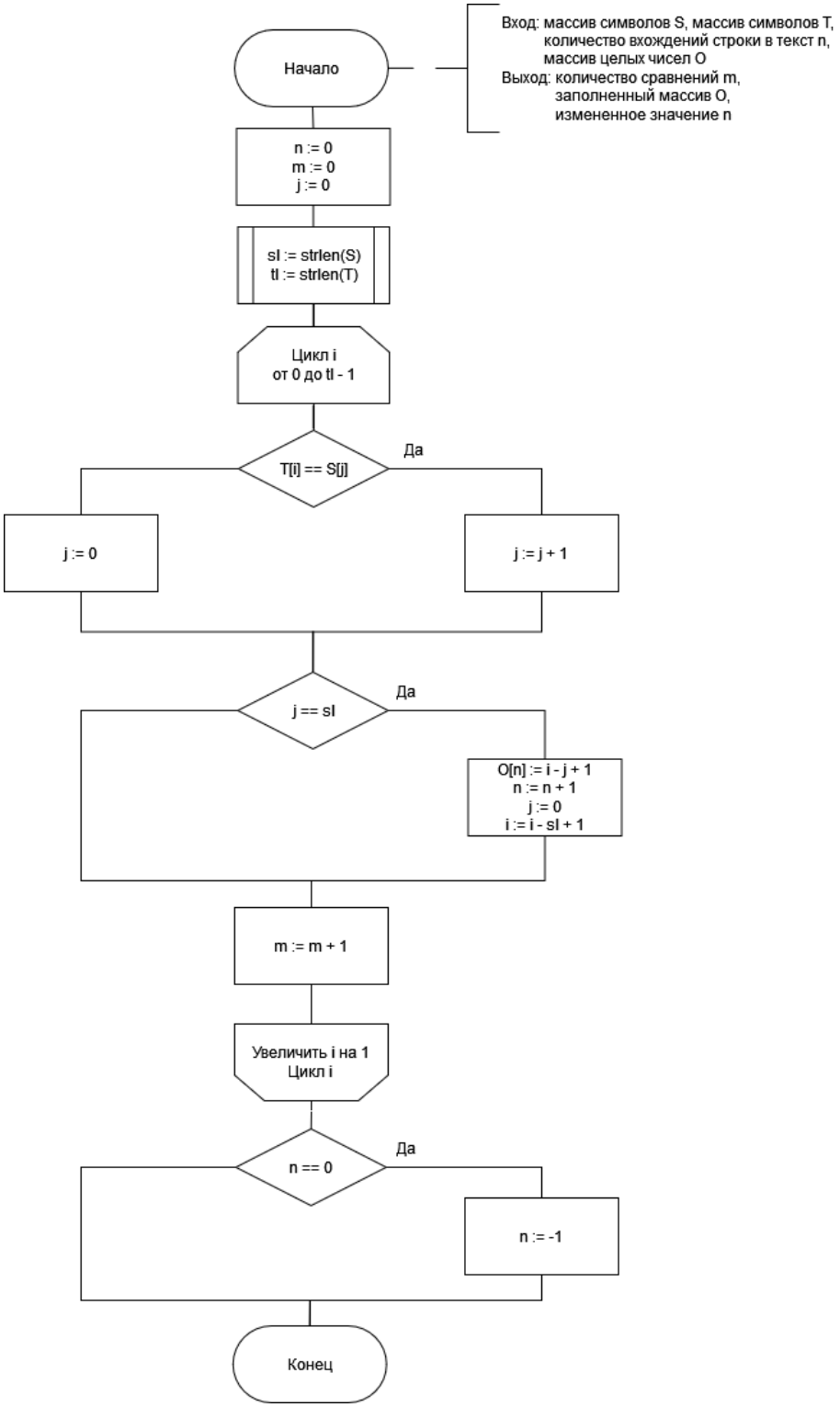

Рисунок 2.4 – Классический алгоритм поиска строки в тексте

Вывод из конструкторской части

В текущем разделе на основе теоретических данных, полученных из аналитического раздела, были построены схемы двух алгоритмов поиска строки в тексте — классического и Кнута-Морриса-Пратта.

3 Технологическая часть

В текущем разделе приведены средства реализации двух алгоритмов поиска строки в тексте, алгоритмов последовательной и конвейерной обработки данных и листинги кода.

3.1 Требования к программному обеспечению

На вход программе передается два аргумента командной строки — название файла, содержащего текст, и название файла, в котором записано количество строк и сами строки. Затем программа обрабатывает заявки с помощью последовательной и конвейерной обработки данных. В результате работы программы создаются два файла (журнала), в которых записаны результаты обработки заявок.

Программа должна обрабатывать ошибки (например, отсутствие файла с названием, переданным в качестве аргумента командной строки) и корректно завершать работу с выводом информации об ошибке на экран.

3.2 Средства реализации

Для реализации программного обеспечения был выбран язык **C++** ввиду следующих причин:

1) есть возможность создания массивов символов;
2) есть возможность создания потоков с помощью класса **std::thread**;
3) для считывания данных из файла в стандартной библиотеке реализованы функции **fscanf()** и **fgets()**;
4) для записи данных в файл в стандартной библиотеке реализована функция **fprintf()**;
5) есть возможность принимать аргументы командной строки.

Таким образом, с помощью языка **C++** можно реализовать программное обеспечение, которое соответствует перечисленным выше требованиям.

3.3 Реализация алгоритмов

3.3.1 Классический алгоритм поиска строки в тексте

В листинге 3.1 показана реализация классического алгоритма поиска строки в тексте.

Листинг 3.1 – Реализация классического алгоритма поиска строки в тексте

```
#include <cstring>

int search_default(const char *const substring, const char
   *const text, int *const occurrences, int *const num_of_occur)
{
    *num_of_occur = 0;
    int num_of_cmp = 0;
    int substring_len = strlen(substring);
    int text_len = strlen(text);
    for (int i = 0, j = 0; i < text_len; i++)
    {
        if (text[i] == substring[j])
            j++;
        else
            j = 0;
        if (j == substring_len)
        {
            occurrences[*num_of_occur] = i - j + 1;
            (*num_of_occur)++;
            j = 0;
            i -= (substring_len - 1);
        }
        num_of_cmp++;
    }
    if (*num_of_occur == 0)
        (*num_of_occur)--;
    return num_of_cmp;
}
```

3.3.2 Алгоритм Кнута-Морриса-Пратта

В листинге 3.2 показана реализация алгоритма Кнута-Морриса-Пратта.

Листинг 3.2 – Реализация алгоритма Кнута-Морриса-Пратта

```
#include <cstring>

static void compute_lps_array(const char *const substring, int
    substring_len, int *const lps)
{
    int len = 0, i = 1;
    lps[0] = 0;
    while (i < substring_len)
    {
        if (substring[i] == substring[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0)
                len = lps[len - 1];
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int search_kmp(const char *const substring, const char *const
    text, int *const occurrences, int *const num_of_occur)
{
    *num_of_occur = 0;
    int num_of_cmp = 0;
    int substring_len = strlen(substring);
```

```
33      int text_len = strlen(text);
34      int lps[substring_len];
35      compute_lps_array(substring, substring_len, lps);
36      int i = 0, j = 0;
37      while ((text_len - i) >= (substring_len - j))
38      {
39          if (substring[j] == text[i])
40          {
41              j++;
42              i++;
43          }
44          if (j == substring_len)
45          {
46              occurrences[*num_of_occur] = i - j;
47              (*num_of_occur)++;
48              j = lps[j - 1];
49          }
50          else if (i < text_len && substring[j] != text[i])
51          {
52              if (j != 0)
53                  j = lps[j - 1];
54              else
55                  i++;
56          }
57          num_of_cmp++;
58      }
59      if (*num_of_occur == 0)
60          (*num_of_occur)--;
61      return num_of_cmp;
62  }
```

3.3.3 Алгоритм последовательной обработки данных

В листинге 3.3 показана реализация последовательной обработки данных.

Листинг 3.3 – Реализация последовательной обработки данных

```
1  #include <queue>
2  #include <mutex>
3  #include <cstdio>
```

```
4  #include <cstdlib>
5
6  static void fprint_array(FILE *f, const int *const array, const
       int size)
7  {
8      for (int i = 0; i < size; i++)
9          fprintf(f, "%d ", array[i]);
10     fprintf(f, "\n");
11 }
12
13 void log_data(const request_t *const request)
14 {
15     fprintf(request->f, "Substring: %s\n", request->substring);
16     fprintf(request->f, "Default: %d comparisons; occurences: ",
           request->noc_1);
17     fprint_array(request->f, request->occurrences_1,
           request->noo_1);
18     fprintf(request->f, "KMP     : %d comparisons; occurences: ",
           request->noc_2);
19     fprint_array(request->f, request->occurrences_2,
           request->noo_2);
20     fprintf(request->f, "\n");
21 }
22
23 static void stage_1(std::queue<request_t> &q1,
       std::queue<request_t> &q2)
24 {
25     std::mutex m;
26     m.lock();
27     request_t request = q1.front();
28     m.unlock();
29     request.noc_1 = search_default(request.substring,
           request.text, request.occurrences_1, &request.noo_1);
30     m.lock();
31     q2.push(request);
32     m.unlock();
33     m.lock();
34     q1.pop();
35     m.unlock();
```

```
36 }
37
38 static void stage_2(std::queue<request_t> &q2,
      std::queue<request_t> &q3)
39 {
40     std::mutex m;
41     m.lock();
42     request_t request = q2.front();
43     m.unlock();
44     request.noc_2 = search_kmp(request.substring, request.text,
         request.occurrences_2, &request.noo_2);
45     m.lock();
46     q3.push(request);
47     m.unlock();
48     m.lock();
49     q2.pop();
50     m.unlock();
51 }
52
53 static void stage_3(std::queue<request_t> &q3, int task_num)
54 {
55     std::mutex m;
56     m.lock();
57     request_t request = q3.front();
58     m.unlock();
59     log_data(&request);
60     m.lock();
61     q3.pop();
62     m.unlock();
63 }
64
65 void linear_processing(int num_of_requests, char *text, char
      substrings[][50])
66 {
67     size_t time_start = 0, time_end = 0;
68     FILE *f = fopen("linear.log", "w");
69     std::queue<request_t> q1, q2, q3;
70     std::mutex m;
71     for (int i = 0; i < num_of_requests; i++)
```

```
72      {
73          request_t request = generate_request(f, text,
                substrings[i]);
74          q1.push(request);
75      }
76      get_time(time_start);
77      for (int i = 0; i < num_of_requests; i++)
78      {
79          stage_1(std::ref(q1), std::ref(q2));
80          stage_2(std::ref(q2), std::ref(q3));
81          stage_3(std::ref(q3), i + 1);
82      }
83      get_time(time_end);
84      printf("Time: %zu ticks.\n", time_end - time_start);
85      fclose(f);
86  }
```

3.3.4 Алгоритм конвейерной обработки данных

В листинге 3.4 показана реализация двух структур данных — заявки и состояния заявки.

Листинг 3.4 – Реализация структур данных

```
1   #include <cstdio>
2
3   #define N 1024
4
5   typedef struct
6   {
7       char *text;
8       char *substring;
9       int occurrences_1[N], occurrences_2[N];
10      int noc_1, noc_2;
11      int noo_1, noo_2;
12      FILE *f;
13  } request_t;
14
15  typedef struct request_state
16  {
```

```
    bool stage_1;
    bool stage_2;
    bool stage_3;
} request_state_t;
```

В листинге 3.5 показана реализация конвейерной обработки данных.

Листинг 3.5 – Реализация конвейерной обработки данных

```
#include <queue>
#include <mutex>
#include <thread>
#include <cstdio>
#include <cstdlib>

static request_t generate_request(FILE *f, char *text, char
   *substring)
{
    request_t request;
    request.text = text;
    request.substring = substring;
    request.noo_1 = 0;
    request.noo_2 = 0;
    request.noc_1 = 0;
    request.noc_2 = 0;
    request.f = f;
    return request;
}

static void
   init_request_state_array(std::vector<request_state_t>
   &request_state_array, int num_of_requests)
{
    request_state_array.resize(num_of_requests);
    for (int i = 0; i < num_of_requests; i++)
    {
        request_state_t request_state;
        request_state.stage_1 = false;
        request_state.stage_2 = false;
        request_state.stage_3 = false;
        request_state_array[i] = request_state;
```

```
30      }
31  }
32
33  static void parallel_stage_1(std::queue<request_t> &q1,
        std::queue<request_t> &q2,
34                          std::vector<request_state_t>
                              &request_state_array,
35                          bool &q1_is_empty)
36  {
37      int task_num = 1;
38      while(!q1.empty())
39      {
40          stage_1(std::ref(q1), std::ref(q2));
41          request_state_array[task_num - 1].stage_1 = true;
42          task_num++;
43      }
44      q1_is_empty = true;
45  }
46
47  static void parallel_stage_2(std::queue<request_t> &q2,
        std::queue<request_t> &q3,
48                          std::vector<request_state_t>
                              &request_state_array,
49                          bool &q1_is_empty, bool &q2_is_empty)
50  {
51      int task_num = 1;
52      while(true)
53      {
54          if (!q2.empty())
55          {
56              if (request_state_array[task_num - 1].stage_1 ==
                    true)
57              {
58                  stage_2(std::ref(q2), std::ref(q3));
59                  request_state_array[task_num - 1].stage_2 = true;
60                  task_num++;
61              }
62          }
63          else if (q1_is_empty)
```

```
64              break;
65      }
66      q2_is_empty = true;
67  }
68
69  static void parallel_stage_3(std::queue<request_t> &q3,
70                               std::vector<request_state_t>
                                    &request_state_array,
71                               bool &q2_is_empty)
72  {
73      int task_num = 1;
74      while(true)
75      {
76          if (!q3.empty())
77          {
78              if (request_state_array[task_num - 1].stage_2 ==
                    true)
79              {
80                  stage_3(std::ref(q3), task_num);
81                  request_state_array[task_num - 1].stage_3 = true;
82                  task_num++;
83              }
84          }
85          else if(q2_is_empty)
86              break;
87      }
88  }
89
90  void conveyor_processing(int num_of_requests, char *text, char
        substrings[][50])
91  {
92      size_t time_start = 0, time_end = 0;
93      FILE *f = fopen("conveyor.log", "w");
94      std::queue<request_t> q1, q2, q3;
95      std::mutex m;
96      for (int i = 0; i < num_of_requests; i++)
97      {
98          request_t request = generate_request(f, text,
              substrings[i]);
```

```
99          q1.push(request);
100     }
101     bool q1_is_empty = false, q2_is_empty = false;
102     std::vector<request_state_t> request_state_array;
103     init_request_state_array(request_state_array,
           num_of_requests);
104     std::thread threads[THREADS_COUNT];
105     get_time(time_start);
106     threads[0] = std::thread(parallel_stage_1, std::ref(q1),
           std::ref(q2), std::ref(request_state_array),
           std::ref(q1_is_empty));
107     threads[1] = std::thread(parallel_stage_2, std::ref(q2),
           std::ref(q3), std::ref(request_state_array),
           std::ref(q1_is_empty), std::ref(q2_is_empty));
108     threads[2] = std::thread(parallel_stage_3, std::ref(q3),
           std::ref(request_state_array), std::ref(q2_is_empty));
109     get_time(time_end);
110     for (int i = 0; i < THREADS_COUNT; i++)
111         threads[i].join();
112     printf("Time: %zu ticks.\n", time_end - time_start);
113     fclose(f);
114 }
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные для двух функций, реализующих соответственно классический алгоритм поиска строки в тексте и алгоритм Кнута-Морриса-Пратта.

Тесты выполнялись по методологии черного ящика (модульное тестирование). Все тесты пройдены успешно.

Таблица 3.1 – Тестовые данные

Текст	**Строка**	**Классический алгоритм**	**Алгоритм Кнута-Морриса-Пратта**
NULL	NULL	-1	-1
ababcabcbabcbcbab	ababcabcbabcbcbab	0	0
ababcabcbabcbcbab	q	-1	-1
ababcabcbabcbcbab	ab	0, 2, 5, 9, 15	0, 2, 5, 9, 15
abababababa-bababababab	aba	0, 2, 4, 6, 8, 10, 12, 14, 16, 18	0, 2, 4, 6, 8, 10, 12, 14, 16, 18

Вывод из технологической части

В данном разделе был написан исходный код двух алгоритмов поиска строки в тексте — классического и Кнута-Морриса-Пратта. Также написан исходный код последовательной и конвейерной обработки данных. Описаны тесты и приведены результаты тестирования.

4 Исследовательская часть

4.1 Технические характеристики устройства

Технические характеристики устройства, на котором было проведено измерение времени работы алгоритмов:

1) операционная система Windows 11 Pro x64;
2) оперативная память 16 ГБ;
3) процессор Intel® Core™ i7-4790K @ 4.00 ГГц.

4.2 Время работы алгоритмов

Время работы функции замерено с помощью ассемблерной инструкции **rdtsc**, которая читает счетчик Time Stamp Counter и возвращает 64-битное количество тиков с момента последнего сброса процессора.

В таблице 4.1 приведено время работы в тиках линейной и конвейерной обработки заявок в зависимости от количества заявок. На рисунке 4.1 изображена зависимость времени работы в тиках функций, реализующих линейную и конвейерную обработку заявок в зависимости от количества заявок.

Таблица 4.1 – Время работы в тиках линейной и конвейерной обработки заявок

Количество заявок	**Линейная обработка**	**Конвейерная обработка**
10	167304	471175
20	298027	544363
30	435152	546690
40	523112	556768
50	570023	601194
60	686216	657707
70	786345	680234
80	800943	710394
90	1000324	897345
100	1382930	987234

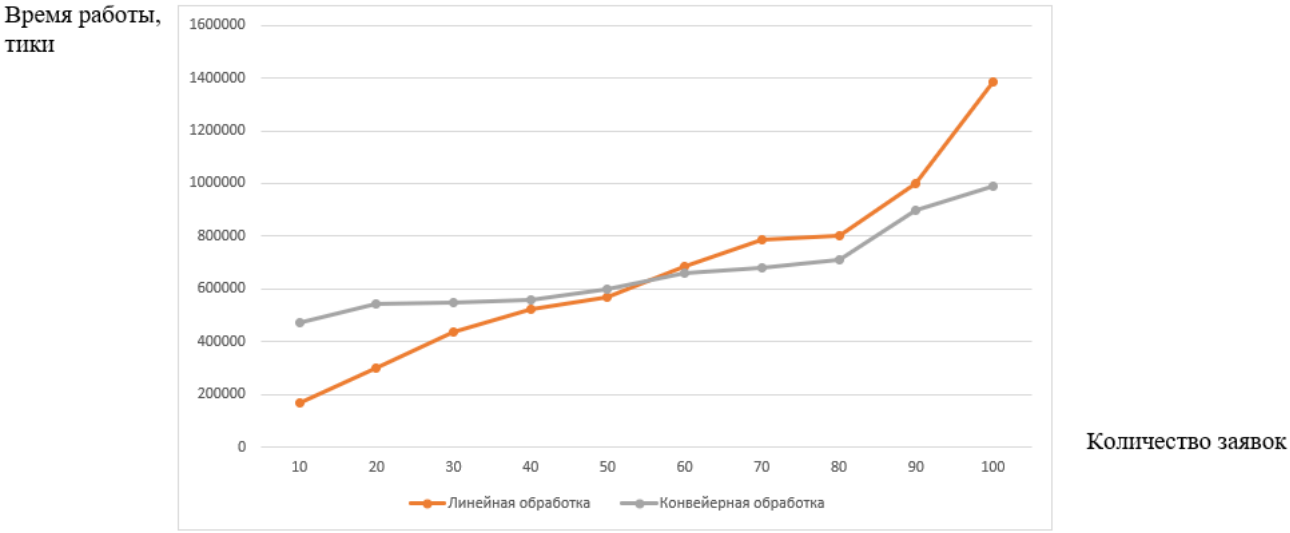

Рисунок 4.1 – Зависимость времени работы в тиках функций, реализующих линейную и конвейерную обработку заявок в зависимости от количества заявок

4.3 Пример файла журнала

В листинге 4.1 показан пример файла журнала, который генерируется программой после обработки заявок.

Листинг 4.1 – Пример файла журнала

```
Substring: a
Default: 83 comparisons; occurences: 0 3 5 7 9 13 14 15 16 18 20
   22 24 27 28 31
KMP    : 83 comparisons; occurences: 0 3 5 7 9 13 14 15 16 18 20
   22 24 27 28 31

Substring: ab
Default: 117 comparisons; occurences: 0 3 5 7 9 18 20 22 24 31
   33 35 37 39 41 43
KMP    : 83 comparisons; occurences: 0 3 5 7 9 18 20 22 24 31 33
   35 37 39 41 43

Substring: ba
Default: 113 comparisons; occurences: 2 4 6 8 12 17 19 21 23 32
   34 36 38 40 42
```

```
KMP    : 82 comparisons; occurences: 2 4 6 8 12 17 19 21 23 32
   34 36 38 40 42

Substring: abb
Default: 89 comparisons; occurences: 0 9 24 28 63 66
KMP    : 81 comparisons; occurences: 0 9 24 28 63 66

Substring: babb
Default: 86 comparisons; occurences: 8 23 62 65
KMP    : 82 comparisons; occurences: 8 23 62 65
```

4.4 Вывод из исследовательской части

В данном разделе был проведен эксперимент по измерению времени работы линейной и конвейерной обработки заявок. Согласно полученным при проведении эксперимента данным, конвейерная обработка данных начинает работать быстрее линейной при количестве входных заявок больше 50.

Заключение

В рамках данной лабораторной работы была достигнута поставленная цель: были изучены принципы конвейерной обработки данных и реализованы классический алгоритм поиска строки в тексте и алгоритм Кнута-Морриса-Пратта.

Решены все поставленные задачи:

1) изучено понятие конвейерной обработки данных;
2) реализованы классический алгоритм поиска строки в тексте и алгоритм Кнута-Морриса-Пратта;
3) реализована последовательная обработка данных на основе двух алгоритмов поиска строки в тексте;
4) реализована конвейерная обработка данных на основе двух алгоритмов поиска строки в тексте;
5) проведен сравнительный анализ времени работы линейной и конвейерной обработки данных на основе экспериментальных данных.

В ходе проведения эксперимента был сделан вывод, что конвейерная обработка данных начинает работать быстрее при количестве входных заявок больше 50.

Список использованных источников

1. *Шпаковский Г. И.* Коротко о параллельном программировании и аппаратуре // Минск. — 2013. — С. 5.
2. *Антонов А. С.* Введение в параллельные вычисления // Московский государственный университет им. М. В. Ломоносова. — 2002. — С. 17—18.
3. Алгоритмы: построение и анализ / Т. Кормен [и др.]. — 2-е изд. — Вильямс, 2013. — С. 1296.