



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 №7 по курсу "Анализ алгоритмов"

Тема Поиск в словаре

Студент Якуба Д. В.

Группа ИУ7-53Б

Оценка (баллы) \_\_\_\_\_

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

# Оглавление

<b>Введение</b>	<b>4</b>
<b>1 Аналитическая часть</b>	<b>5</b>
1.1 Словарь . . . . .	5
1.2 Алгоритм поиска по словарю полным перебором . . . . .	6
1.3 Алгоритм бинарного поиска по словарю . . . . .	7
1.4 Частотный анализ . . . . .	7
<b>2 Конструкторская часть</b>	<b>9</b>
2.1 Структура записи в словаре . . . . .	9
2.2 Схема алгоритма поиска по словарю полным перебором . . . . .	9
2.3 Схема алгоритма бинарного поиска по словарю . . . . .	9
2.4 Схема алгоритма поиска по словарю с использованием разбиения на сегменты . . . . .	12
<b>3 Технологическая часть</b>	<b>14</b>
3.1 Требования к программному обеспечению . . . . .	14
3.2 Средства реализации программного обеспечения . . . . .	14
3.3 Листинг кода . . . . .	15
3.4 Тестирование программного продукта . . . . .	17
<b>4 Исследовательская часть</b>	<b>19</b>
4.1 Технические характеристики . . . . .	19
4.2 Пример работы программного обеспечения . . . . .	19
4.3 Время выполнения алгоритмов . . . . .	20
<b>Заключение</b>	<b>23</b>



# Введение

## Цель лабораторной работы

Реализация алгоритмов поиска по словарю: перебором, бинарным поиском и с применением частотного анализа.

## Задачи лабораторной работы

- 1) изучить алгоритм поиска по словарю полным перебором;
- 2) изучить алгоритм бинарного поиска по словарю;
- 3) изучить алгоритм поиска по словарю с применением частотного анализа;
- 4) протестировать реализованные алгоритмы;
- 5) провести анализ временных характеристик реализованных алгоритмов;
- 6) подготовить отчёт по проведенной работе.

# 1 | Аналитическая часть

В данном разделе описаны определение словаря как структуры данных, а также алгоритмы поиска по словарю.

## 1.1 Словарь

Словарь (ассоциативный массив)[1] — это абстрактный тип данных, хранящий пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- *INSERT*(*ключ*, *значение*);
- *FIND*(*ключ*);
- *REMOVE*(*ключ*).

Предполагается, что все ключи в словаре являются уникальными.

В паре (*ключ*, *значение*) *значение* называется значением, ассоциированным с ключом.

Операция *FIND*(*ключ*) возвращает значение, ассоциированное с заданным ключом, или некоторый специальный объект, означающий, что значения, ассоциированного с заданным ключом, нет. Две другие операции ничего не возвращают (за исключением, возможно, информации о том, успешно ли была выполнена данная операция).

## 1.2 Алгоритм поиска по словарю полным перебором

Алгоритм полного перебора [2] — это алгоритм разрешения математических задач, который можно отнести к классу способов нахождения решения рассмотрением всех возможных вариантов.

Для решения поставленной задачи поиска с использованием метода полного перебора потребуется последовательно просматривать каждую запись в словаре. В том случае, если у рассматриваемой пары ключ совпадает с искомым - алгоритм завершает свою работу, задача выполнена.

Трудоёмкость алгоритма зависит от того, присутствует ли искомым ключ в словаре, и, если присутствует - насколько он далеко от начала массива ключей.

При решении задачи возможно возникновение  $(N + 1)$  случаев, где  $N$  - это количество записей в словаре: ключ не найден и  $N$  возможных случаев расположения ключа в словаре.

Лучшим случаем для рассматриваемого алгоритма будет факт того, что искомым ключ был обнаружен за одно сравнение (то есть ключ находится в начале словаря).

Худший случай наступает при следующих стечениях обстоятельств:

- элемент не был найден за  $N$  сравнений;
- ключ был обнаружен на последнем сравнении.

Пусть на старте алгоритм поиска затрачивает  $k_0$  операций, а при каждом сравнении  $k_1$  операций. Тогда в лучшем случае потребуется  $k_0 + k_1$  операций; в случае, если ключ был найден на втором сравнении - потребуется  $k_0 + 2k_1$  операций; в случае нахождения ключа на последней позиции или его отсутствия в словаре - потребуется  $k_0 + Nk_1$  операций.

Средняя трудоёмкость алгоритма может быть рассчитана как математическое ожидание по формуле 1.2 ( $\Omega$  - множество всех возможных исходов).

$$\begin{aligned}
\sum_{i \in \Omega} p_i \cdot f_i &= (k_0 + k_1) \cdot \frac{1}{N+1} + (k_0 + 2 \cdot k_1) \cdot \frac{1}{N+1} + (k_0 + 3 \cdot k_1) \cdot \frac{1}{N+1} + \\
&\quad + (k_0 + N k_1) \frac{1}{N+1} + (k_0 + N \cdot k_1) \cdot \frac{1}{N+1} = \\
&= k_0 \frac{N+1}{N+1} + k_1 + \frac{1+2+\dots+N+N}{N+1} = k_0 + k_1 \cdot \left( \frac{N}{N+1} + \frac{N}{2} \right) = \\
&= k_0 + k_1 \cdot \left( 1 + \frac{N}{2} - \frac{1}{N+1} \right) \quad (1.1)
\end{aligned}$$

### 1.3 Алгоритм бинарного поиска по словарю

При двоичном поиске обход можно представить деревом, поэтому трудоёмкость в худшем случае составит  $\log_2 N$  (спуск по двоичному дереву от корня до листа). Скорость роста функции  $\log_2 N$  меньше, чем у  $N$ .

### 1.4 Частотный анализ

Некоторый алгоритм на получает словарь и по нему составляется частотный анализ:

- по частоте использования ключа на реальных данных;
- по частоте появления в выборке первого символа ключа (или его остатка от деления на некоторое значение, в случае чисел).

По предоставленному отчёту словарь разбивается на сегменты. В каждом сегменте находятся элементы с некоторым, определённым анализатором, признаком.

Сегменты также могут быть упорядочены, например, по размеру сегмента, если множество исходов обращений к сегменту имеет высокую дисперсию.

Вероятность обращения к определенному сегменту равна сумме вероятностей обращений к его ключам (формула 1.2, где  $P_i$  - вероятность обращения к  $i$ -ому сегменту,  $p_j$  - вероятность обращения к  $j$ -ому элементу, который принадлежит  $i$ -ому сегменту).

$$P_i = \sum_j p_j \quad (1.2)$$

Если обращения ко всем ключам равновероятны, то можно заменить сумму на произведение (формула 1.3, где  $N$  - количество элементов в  $i$ -ом сегменте, а  $p$  - вероятность обращения к произвольному ключу).

$$P_i = N \cdot p \quad (1.3)$$

Ключи в сегментах также упорядочиваются для проведения бинарного поиска.

Как итог, сначала с помощью бинарного поиска выбирается требующийся сегмент. В найденном сегменте с помощью алгоритма бинарного поиска обнаруживается требующийся ключ. Средняя трудоёмкость представленного алгоритма действий будет определяться формулой 1.4, где  $M$  - количество сегментов.

$$f_{cp} = \sum_{i \in [1, M]} (f_{\text{выбора } i\text{-го сегмента}} + f_{\text{ср. поиска в } i\text{-м сегменте}}) \cdot p_i \quad (1.4)$$

## Вывод

Были рассмотрены определение словаря как структуры данных, а также алгоритмы поиска по словарю и оптимизации поиска.

В данной работе стоит задача реализации трёх рассмотренных алгоритмов поиска.



## 2 | Конструкторская часть

В данном разделе представлены структура записей в словаре, а также схемы алгоритма поиска по словарю полным перебором, с использованием бинарного поиска, а также с использованием сегментирования и частотного анализа.

### 2.1 Структура записи в словаре

Каждая запись в словаре описана парой вида ( $ID_{студента}$  : *Название курсового проекта*).

### 2.2 Схема алгоритма поиска по словарю полным перебором

Схема алгоритма поиска полным перебором предоставлена на рисунке 2.2.

### 2.3 Схема алгоритма бинарного поиска по словарю

Схема алгоритма бинарного поиска по словарю предоставлена на рисунке ??.

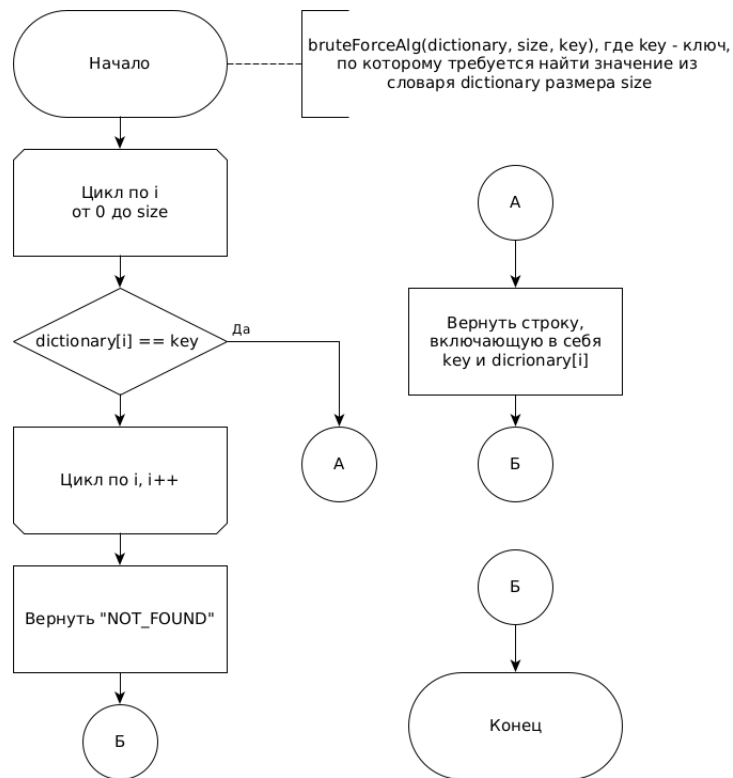


Рис. 2.1: Схема алгоритма поиска полным перебором.

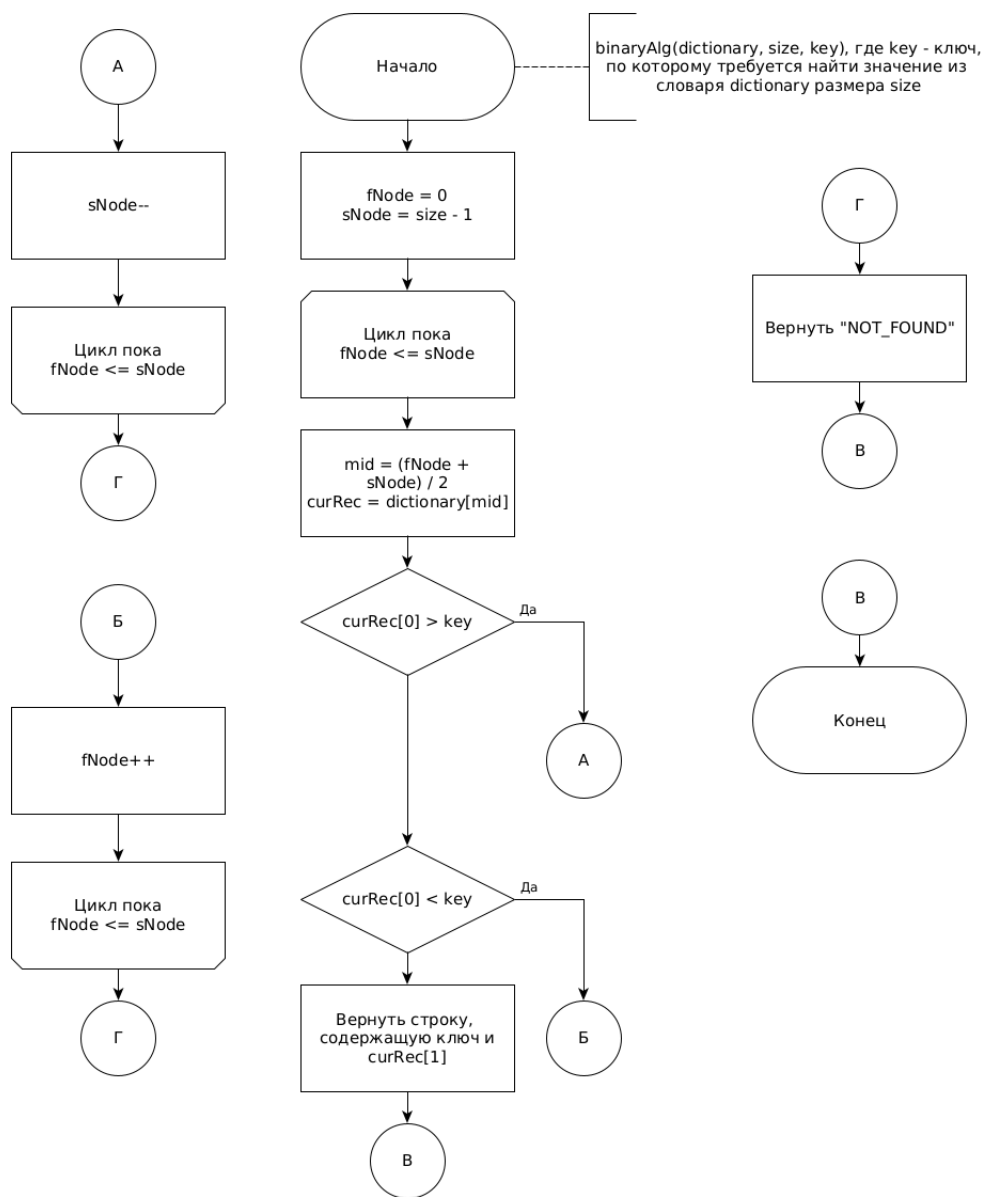


Рис. 2.2: Схема алгоритма бинарного поиска по словарю.

## 2.4 Схема алгоритма поиска по словарю с использованием разбиения на сегменты

Схема алгоритма бинарного поиска по словарю с использованием разбиения на сегменты предоставлена на рисунке ??.

### Вывод

Были представлены структура записей в словаре, а также схемы алгоритма поиска по словарю полным перебором, с использованием бинарного поиска, а также с использованием сегментирования и частотного анализа.

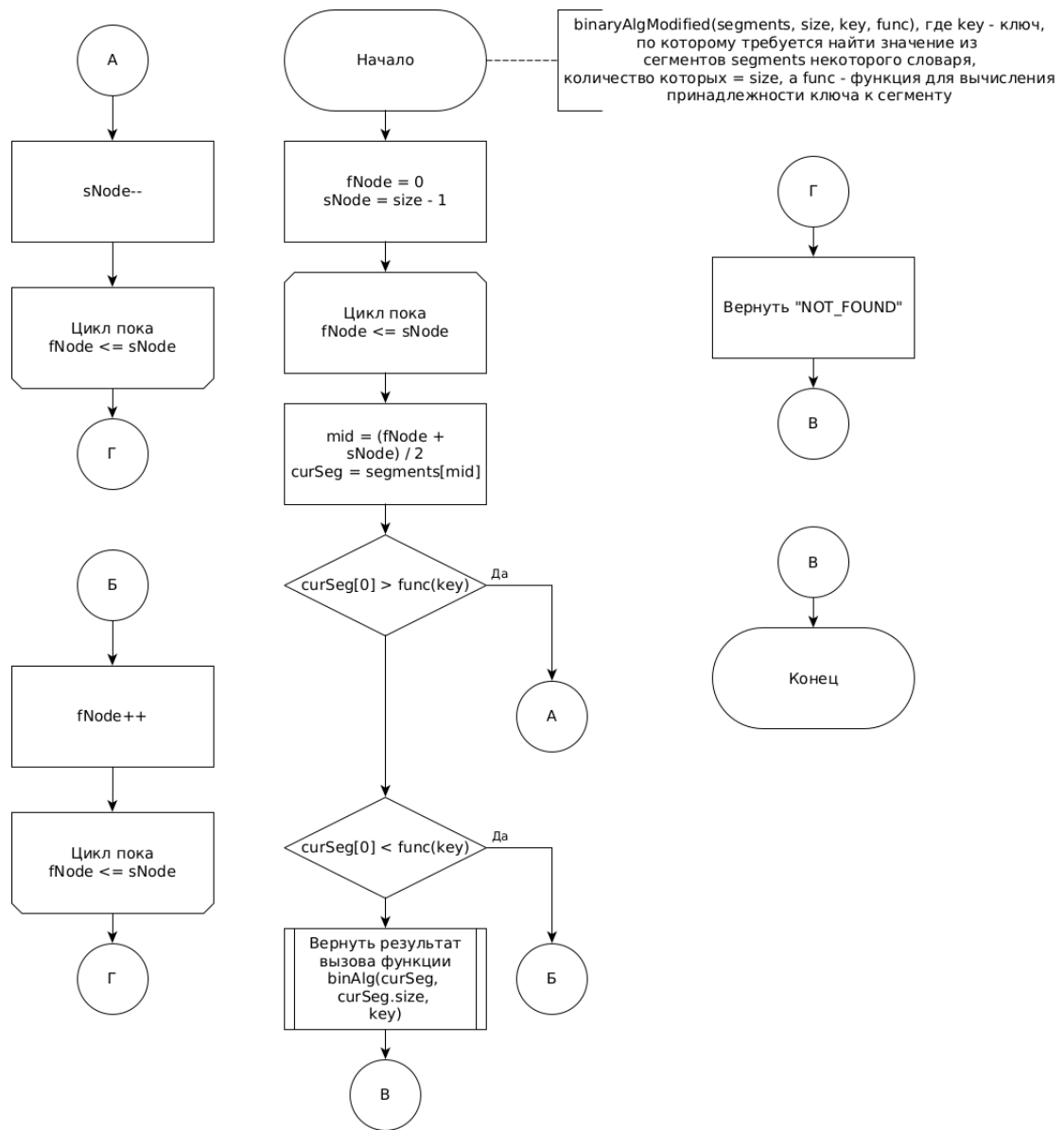


Рис. 2.3: Схема алгоритма бинарного поиска по словарю с сегментированием.

## 3 | Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации программного обеспечения, а также листинг кода.

### 3.1 Требования к программному обеспечению

- входные данные - словарь и искомый ключ;
- выходные данные - значение, соответствующее ключу в словаре, если он в нём присутствует, иначе - строка "*NOT\_FOUND*".

### 3.2 Средства реализации программного обеспечения

При написании программного продукта был задействован язык программирования Kotlin [3].

Данный выбор обусловлен следующими факторами:

- Возможность портирования алгоритмов для работы с Android;
- Большое количество справочной литературы, связанной с ЯП Java.

Для тестирования производительности реализаций алгоритмов использовалась утилита `measureTimedValue`.

При написании программного продукта использовалась среда разработки IntelliJ IDEA [?].

Данный выбор обусловлен тем, что язык программирования Kotlin - это разработка компании JetBrains, поставляющей данную среду разработки.

Для написания утилиты, генерирующей данные для словаря, использовался ЯП Python[4]. Данный выбор обусловлен простотой языка.

Для генерации значений в словаре использовалась библиотека Faker [5] для ЯП Python.

### 3.3 Листинг кода

В листинге 3.1 предоставлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Реализации рассматриваемых алгоритмов поиска

```
1 class Dictionary
2 {
3     private val NOT_FOUND = "NOT_FOUND"
4
5     private val dictionary: MutableList<Pair<Int, String>> =
6         mutableListOf()
7     private val segmentedDictionary: MutableList<Pair<Int,
8         MutableList<Pair<Int, String>>>> = mutableListOf()
9
10    private val secretFunc: (Int) -> (Int) = { it % 100 }
11
12    fun isEmpty() : Boolean
13    {
14        return dictionary.isEmpty()
15    }
16
17    fun fullByFile(filePath: String)
18    {
19        val reader = Files.newBufferedReader(Paths.get(filePath))
20        val parser = CSVParser(reader, CSVFormat.DEFAULT.
21            withDelimiter(';'))
22
23        for (curRecord in parser)
24            dictionary.add(Pair(curRecord[0].toInt(), curRecord
25                [1]))
26
27        if (!parser.isClosed)
28            parser.close()
29
30        dictionary.shuffle()
31    }
32
33    fun getValueByBrutForce(key: Int) : String
```

```

30 {
31     for (curlIndex in 0 until dictionary.size)
32     {
33         if (dictionary[curlIndex].first == key)
34             return "{ %d : %s }".format(key, dictionary[
35                 curlIndex].second)
36     }
37     return NOT_FOUND
38 }
39 fun sortForBinarySearch(dictionary_ : MutableList<Pair<Int,
40     String>> = this.dictionary)
41 {
42     dictionary_.sortBy { it.first }
43 }
44 fun getValueByBinarySearch(key: Int, dictionary_ : MutableList
45     <Pair<Int, String>> = this.dictionary) : String
46 {
47     var firstNode = 0
48     var secondNode = dictionary_.size - 1
49     while (firstNode <= secondNode)
50     {
51         val middle = (firstNode + secondNode) / 2
52         val curRecordID = dictionary_[middle]
53
54         when
55         {
56             curRecordID.first > key -> secondNode--
57             curRecordID.first < key -> firstNode++
58             else -> return "{ %d : %s }".format(key,
59                 curRecordID.second)
60         }
61     }
62     return NOT_FOUND
63 }
64 fun createSegmentedDictionary()
65 {
66     for (i in 0 until 100)
67         segmentedDictionary.add(Pair(i, dictionary.filter {
68             secretFunc(it.first) == i }.toMutableList()))
69
70     segmentedDictionary.forEach { segment -> segment.second.

```



```

70         sortBy { it.first } }
71     segmentedDictionary.sortBy { it.second.size }
72 }
73 fun getValueBySegmentedAndBinaryModified(key: Int) : String
74 {
75     if (segmentedDictionary.isEmpty())
76         return "No segmented array."
77
78     var firstNode = 0
79     var secondNode = segmentedDictionary.size - 1
80     while (firstNode <= secondNode)
81     {
82         val middle = (firstNode + secondNode) / 2
83         val curSegment = segmentedDictionary[middle]
84
85         when
86         {
87             curSegment.first > secretFunc(key) -> secondNode =
88                 middle
89             curSegment.first < secretFunc(key) -> firstNode++
90             else -> return getValueByBinarySearch(key,
91                 curSegment.second)
92         }
93     }
94     return NOT_FOUND
95 }
96 fun print()
97 {
98     for (i in dictionary)
99         println(i)
100 }
101 }

```

### 3.4 Тестирование программного продукта

В таблице 3.1 приведены тесты для функций поиска значений в словаре по ключу. Тесты пройдены успешно.

Таблица 3.1: Тестирование функций

Ключ	Ожидаемый результат
666	Эксплуатация богатых парадигм
22	Ускорение сетевых схем
1001	<i>NOT_FOUND</i>
-1	<i>NOT_FOUND</i>

## Вывод

Спроектированные алгоритмы были реализованы и протестированы.

## 4 | Исследовательская часть

### 4.1 Технические характеристики

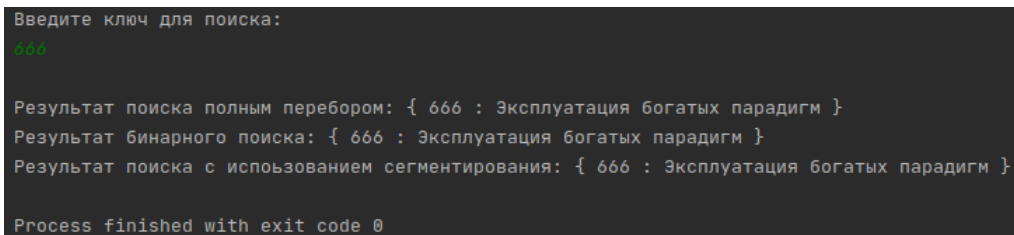
Технические характеристики ЭВМ, на котором выполнялись исследования:

- ОС: Manjaro Linux 20.1.1 Mikah;
- Оперативная память: 16 Гб;
- Процессор: Intel Core i7-10510U.

При проведении замеров времени ноутбук был подключен к сети электропитания.

### 4.2 Пример работы программного обеспечения

На рисунках 4.1, 4.2 приведены примеры работы программы.



```
Введите ключ для поиска:
666

Результат поиска полным перебором: { 666 : Эксплуатация богатых парадигм }
Результат бинарного поиска: { 666 : Эксплуатация богатых парадигм }
Результат поиска с использованием сегментирования: { 666 : Эксплуатация богатых парадигм }

Process finished with exit code 0
```

Рис. 4.1: Пример работы ПО.

```
Введите ключ для поиска:  
23146873  
  
Результат поиска полным перебором: NOT_FOUND  
Результат бинарного поиска: NOT_FOUND  
Результат поиска с использованием сегментирования: NOT_FOUND  
  
Process finished with exit code 0
```

Рис. 4.2: Пример работы ПО.

### 4.3 Время выполнения алгоритмов

Алгоритм тестировался на данных, сгенерированных случайным образом.

В таблице 4.1 предоставлено время работы над каждым отрезком в предоставленном примере каждого из выделенных этапов.

Из таблицы видно, что среднее время выполнения этапа 1 составляет  $\approx 17428.6$  наносекунд. Среднее время выполнения этапа 2 составляет  $\approx 38285.7$  наносекунд. Среднее время выполнения этапа 3 составляет  $\approx 18571.4$  наносекунд. Таким образом, этап 1 сравним по среднему времени выполнения с этапом 2. Но после выполнения этапа 1 заметно, что последующие вызовы функции работают за константное время, равное 3000 наносекунд, что при наличии начальных "прогревочных" запусков вылилось бы в факт того, что этап 1 не был бы сопоставим по среднему времени выполнения с этапом 3. Этап 2 является самым долго выполняющимся.

Таблица 4.1: Замеры времени для выполнения выделенных этапов.

Номер отрезка	Время обработки, нс		
	Этап 1	Этап 2	Этап 3
0	97000	74000	27000
1	10000	38000	21000
2	3000	32000	16000
3	3000	35000	19000
4	3000	22000	12000
5	3000	35000	16000
6	3000	32000	19000

## Вывод

При сравнении результатов замеров по времени стало известно, что самым быстрым этапом конвейера оказался этап 1. При этом, самым медленным из трех рассмотренных - этап 2.

В среднем этап 1 работает быстрее этапа 2 на  $\approx 20857.1$  наносекунд. При этом, при четвертой обработке отрезка разница в скорости выполнения составила 32000 наносекунд.

Этап 3 в среднем работает быстрее этапа 2 на  $\approx 19714.3$  наносекунд. При этом, при шестой обработке отрезка разница в скорости выполнения составила 19000 наносекунд.

Таким образом, среднее время выполнения алгоритма для каждого отрезка составило  $\approx 74285.71$  наносекунд.

# Заключение

В ходе выполнения лабораторной работы была выполнена цель и следующие задачи:

- 1) было изучено асинхронное взаимодействие на примере конвейерной обработки данных;
- 2) была спроектирована система конвейерных вычислений;
- 3) была реализована система конвейерных вычислений;
- 4) была протестирована реализованная система;
- 5) был подготовлен отчёт по проведенной работе.

Исследования показали, что в среднем:

- 1) этап 1 работает быстрее этапа 2 на  $\approx 20857.1$  наносекунд;
- 2) этап 3 работает быстрее этапа 2 на  $\approx 19714.3$  наносекунд;
- 3) среднее время выполнения алгоритма составило  $\approx 74285.71$  наносекунд.

# Литература

- [1] National Institute of Standards and Technology [Электронный ресурс]. Режим доступа: <https://xlinux.nist.gov/dads/HTML/assocarray.html> (дата обращения 13.12.2020).
- [2] Н. Нильсон. Искусственный интеллект. Методы поиска решений. М.: Мир, 1973. с. 273.
- [3] Kotlin language specification [Электронный ресурс]. Режим доступа: <https://kotlinlang.org/spec/introduction.html> (дата обращения 09.10.2020).
- [4] Python documentation [Электронный ресурс]. Режим доступа: <https://www.python.org/doc/> (дата обращения 14.12.2020).
- [5] Faker's documentation [Электронный ресурс]. Режим доступа: <https://faker.readthedocs.io/en/master/> (дата обращения 14.12.2020).