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»

на тему: «Поиск в словаре»

Вариант № Статистика по заболевшим covid

Студент ИУ7-51Б _____ Мицевич М. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2021 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **3**

1 Аналитический раздел **4**

1.1 Линейный поиск 4

1.2 Двоичный поиск 4

1.3 Поиск по сегментам 5

1.4 Описание словаря 5

2 Конструкторский раздел **6**

2.1 Описание структур данных 8

3 Технологический раздел **10**

3.1 Средства реализации 10

3.2 Реализация алгоритмов 10

3.3 Тестирование 11

4 Исследовательский раздел **13**

4.1 Технические характеристики 13

4.2 Анализ алгоритмов по количеству сравнений 13

4.3 Вывод 17

ЗАКЛЮЧЕНИЕ **18**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **19**

ВВЕДЕНИЕ

Словарь – структура данных, построенная на основе пар значений. Первое значение пары – ключ для идентификации элементов, второе – собственно сам хранимый элемент. Например, в телефонном справочнике номеру телефона соответствует фамилия абонента. Задача поиска в слове является очень актуальной в современных системах, так как чаще всего идентификатор не может быть представлен индексом (то есть числовым значением).

Цель лабораторной работы

Целью данной лабораторной работы является разработка эффективного алгоритма поиска в словаре.

Задачи лабораторной работы

Для достижения поставленной цели необходимо решить следующие задачи:

- исследовать различные алгоритмы поиска;
- оценить трудоёмкости алгоритмов в лучшем случае, худшем и в среднем;
- привести схемы рассматриваемых алгоритмов;
- описать использующиеся структуры данных;
- определить средства реализации разрабатываемого программного обеспечения;
- реализовать три алгоритма поиска в словаре: линейный, двоичный, по сегментам;
- провести тестирование реализованного программного продукта;
- провести анализ алгоритмов по количеству сравнений.

1 Аналитический раздел

В данном разделе представленные теоретические сведения о рассматриваемых алгоритмах.

1.1 Линейный поиск

Алгоритм линейного поиска заключается в проходе по словарю, до того момента, пока не будет найден искомый ключ. В рассматриваемом алгоритме возможно $N+1$ случаев расположения ключа: ключ является i-ым элементом словаря либо его нет в словаре в принципе.

Лучший случай (трудоемкость $O(1)$): ключ расположен в самом начале словаря и найден за одно сравнение). Худший случай (трудоемкость $O(N)$): ключ расположен в самом конце словаря либо ключ не находится в словаре. Средний случай: $O(N/2) = O(N)$.

1.2 Двоичный поиск

Данный алгоритм подходит только для заранее упорядоченного словаря. Процесс двоичного поиска можно описать следующим образом:

- получить значение находящееся в середине словаря и сравнить его с ключом;
- в случае, если ключ меньше данного значения, продолжить поиск в младшей части словаря, в обратном случае – в старшей части словаря;
- на новом интервале снова получить значение из середины этого интервала и сравнить с ключом.
- поиск продолжать до тех пор, пока не будет найден искомый ключ, или интервал поиска не окажется пустым.

Обход словаря данным алгоритм можно представить в виде дерева, поэтому трудоемкость в худшем случае и в среднем составит $\log_2 N$. Трудоёмкость в лучшем случае: $O(1)$ (элемент сразу оказался средним). Можно сделать вывод, что алгоритм двоичного поиска работает значительно быстрее, чем алгоритм линейного поиска, однако при этом он требует предварительной обработки данных (сортировки).

1.3 Поиск по сегментам

Данный алгоритм также требует предварительной обработки данных, а именно:

- упорядочить словарь;
- разбить словарь на сегменты.

Словарь разбивается на сегменты по какому-либо признаку и сортируется по частоте. Например, если ключ является строкой, то можно сделать разбиение по первой букве в ключе. Если ключ является целым числом, можно провести разбиение по остатку от деления ключа на некоторое число K.

После выполнения разбиения, нужно определить к какому сегменту относится искомый ключ и провести на этом сегменте двоичный поиск.

Таким образом, так же, как и алгоритм двоичного поиска, поиск по сегментам требует предварительной обработки данных.

Трудоёмкость поиска по сегментам складывается из двух величин: трудоёмкости линейного поиска сегмента и бинарного поиска внутри сегмента.

1.4 Описание словаря

Словарь представляет собой массив пар (название страны, число заболевших коронавирусом). Информация взята с API, предоставляемого сайтом covid-19 tracking [1].

Вывод

В данном разделе были рассмотрены особенности алгоритмов поиска в словаре. Кроме того, были приведены трудоёмкости каждого из алгоритмов. Входными данными для программного обеспечения являются словарь, ключ для поиска в словаре и функция, задающая операцию отношения на множестве элементов словаря.

2 Конструкторский раздел

На рисунках 2.1, 2.2 и 2.3 приведены схемы алгоритмов поиска в словаре. На рисунке 2.4 приведён алгоритм сегментирования для поиска по сегментам.

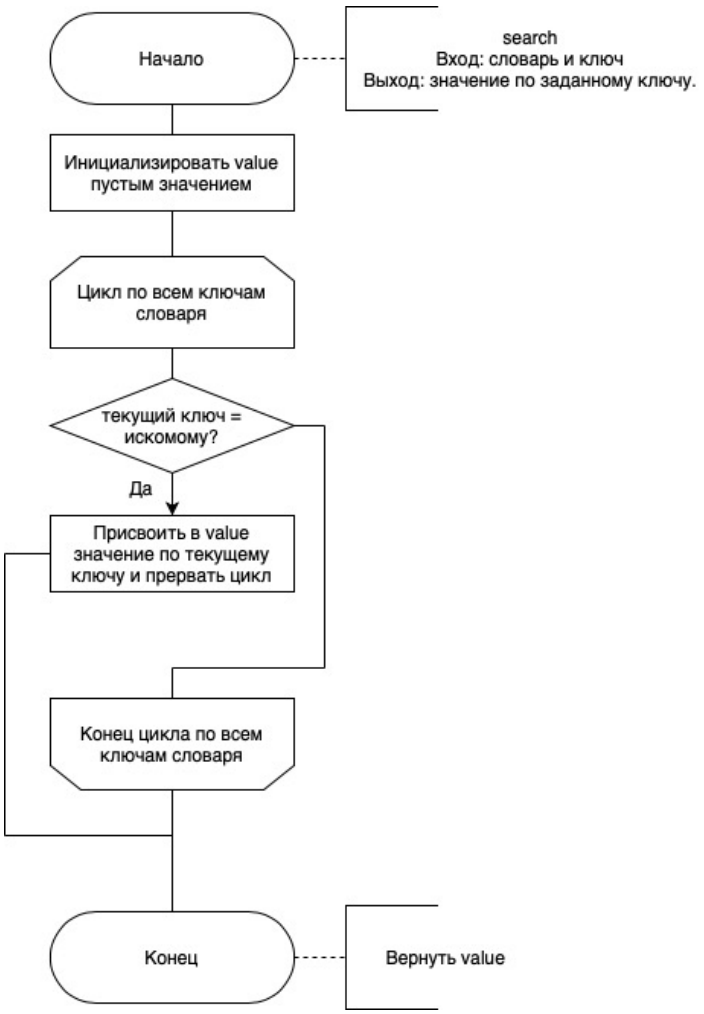

Рисунок 2.1 – Схема алгоритма полного перебора.

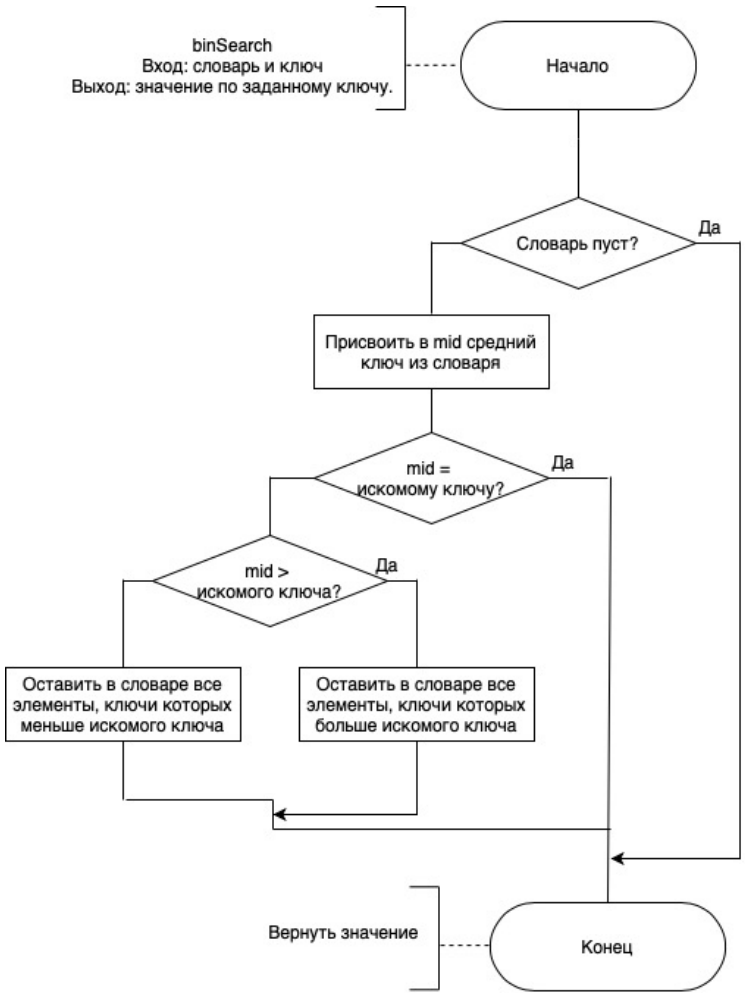

Рисунок 2.2 – Схема алгоритма двоичного поиска.

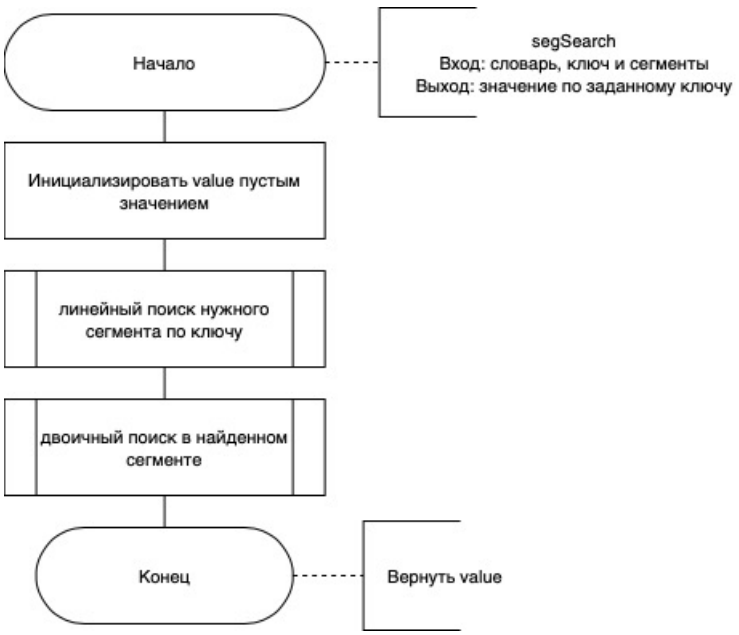

Рисунок 2.3 – Схема алгоритма поиска по сегментам.

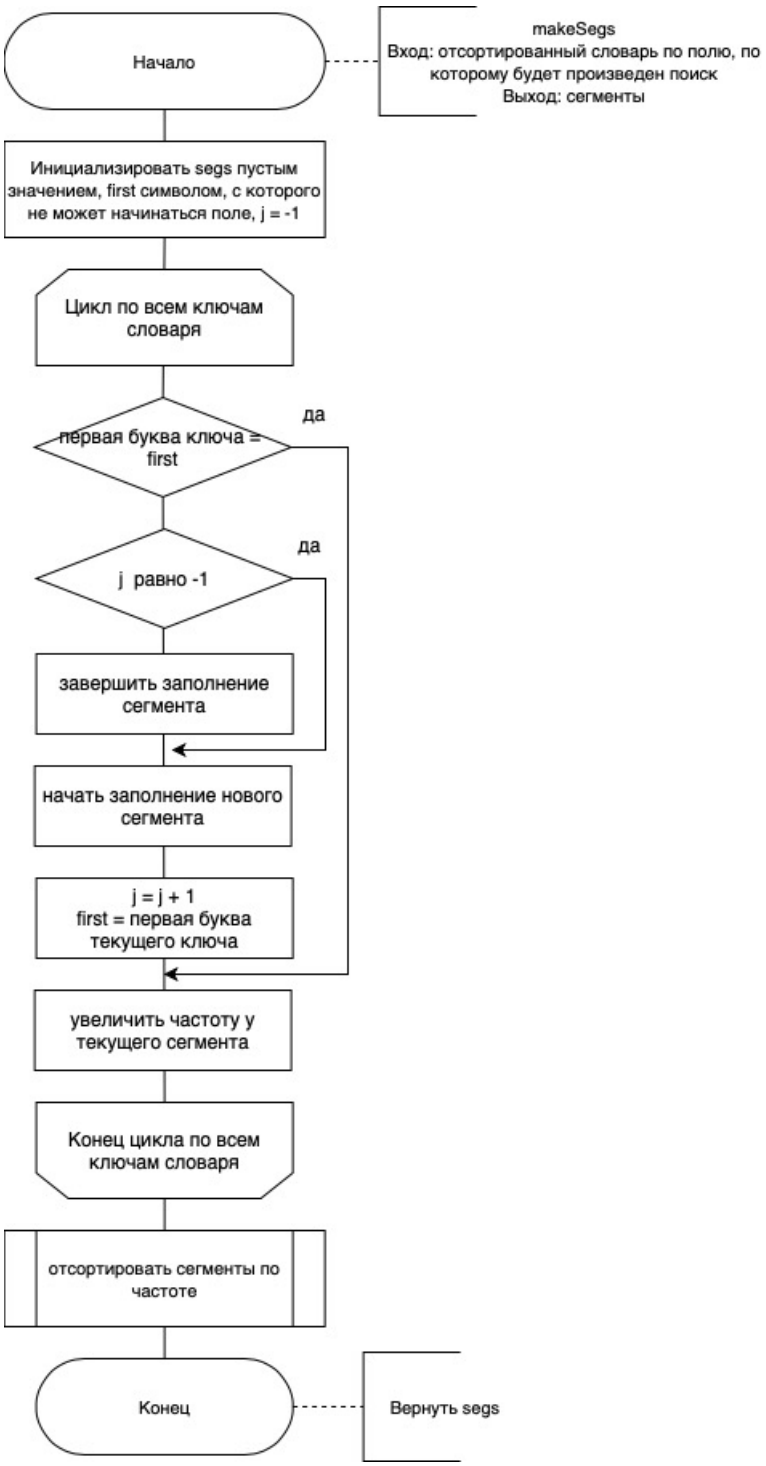

Рисунок 2.4 – Схема алгоритма сегментирования.

2.1 Описание структур данных

Для работы программы требуется ввести три основные структуры данных. Одна из них нужна для хранения записей, вторая – для словаря, третья – для описания информации о сегменте.

Каждая запись представляет собой структуру из двух полей. Первое отвечает за название страны, второе – за количество случаев заболевания

коронавирусом в этой стране.

Словарь – массив структур типа записи, описанной выше.

Для описания сегмента требуется создать структуру, хранящую четыре поля:

- start_index – индекс начала сегмента;
- end_index – индекс конца сегмента;
- letter – буква, с которой начинаются названия стран в этом сегменте
- freq – число записей в сегменте.

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы алгоритмов поиска в словаре.

Также были определены структуры данных, которые нужны для реализации алгоритмов поиска.

3 Технологический раздел

В данном разделе приведены средства реализации и листинги кода.

3.1 Средства реализации

Для реализации ПО я выбрал язык программирования Python [2]. Данный выбор обусловлен тем, что язык обладает мощными инструментами работы со списками и строками, которые облегчают написание программ. Кроме того, существует множество библиотек для него, в том числе, библиотека для работа с json [3], и для построения гистограмм matplotlib [4].

3.2 Реализация алгоритмов

В листингах 3.1, 3.2 и 3.3 представлены листинги алгоритмов поиска в словаре.

Листинг 3.1 – Алгоритм линейного поиска

```
def lin_search(dictionary, key, comp_func):
for i in range(len(dictionary)):
if comp_func(dictionary[i], key) == 0:
return dictionary[i], i + 1
return "not␣found", len(dictionary)
```

Листинг 3.2 – Алгоритм двоичного поиска

```
def bin_search(dictionary, key, comp_func, left, right, is_sorted
   =False):
if (not is_sorted):
dictionary.sort(key=lambda x: x[0])
cmp = 0
while left < right:
cmp += 1
mid = (left+right)//2
midval = dictionary[mid]
if comp_func(midval, key) < 0:
left = mid+1
cmp += 1
elif comp_func(midval, key) > 0:
right = mid
cmp += 2
else:
```

```
cmp += 2
return dictionary[mid], cmp
return "not found", cmp
```

Листинг 3.3 – Алгоритм поиска по сегментам

```
def make_segments(dictionary):
segments = []
last = '\n'
j = -1
dictionary.sort(key=lambda x: x[0])
for i in range (len(dictionary)):
if dictionary[i][0][0] != last:
if j != -1:
segments[j].end_index = i - 1

segments.append(Segment(dictionary[i][0][0], i))
j += 1
last = dictionary[i][0][0]
else:
segments[j].freq += 1
segments.sort(key = lambda s: s.freq, reverse = True)
return segments

def seg_search(dictionary, segments, key, comp_func):
seg = lin_search(segments, key[0], seg_comp)
res = bin_search(dictionary, key, comp_func, seg[0].start_index,
   seg[0].end_index + 1, is_sorted=True)
return res[0], seg[1] + res[1]
```

3.3 Тестирование

В таблице 3.1 приведены данные, на которых производилось тестирование.

Таблица 3.1 – Таблица тестовых данных алгоритмов поиска в словаре.

Входные данные	Ожидаемый результат
Russia	10,159,389
USA	51,435,652
neg_test	not found

Все тесты были пройдены успешно.

Вывод

В данном разделе была разработаны и протестированы алгоритмы поиска в словаре. Кроме того, было показано, что алгоритмы двоичного поиска и алгоритм поиска по сегментам требуют предварительный обработки данных (сортировки и разбиение на сегменты соответственно), в отличие от алгоритма линейного поиска.

4 Исследовательский раздел

В данном разделе приведен анализ характеристик разработанного ПО.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование и исследование, приведены ниже.

- Операционная система: Ubuntu Linux 64-bit.
- Оперативная память: 16 GB.
- Количество логических ядер - 8.
- Процессор: Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz [5].

Тестирование проводилось на компьютере, включенном в сеть электропитания. Во время тестирования компьютер был нагружен только встроенными приложениями окружения рабочего стола, окружением рабочего стола, а также непосредственно системой тестирования. Во время тестирования оптимизации компилятора были отключены.

4.2 Анализ алгоритмов по количеству сравнений

На рисунках 4.1, 4.2 и 4.3 приведены графики линейного, бинарного поиска и поиска по сегментам по количеству сравнений. В таблице 4.1 приведено соотвестсвие индексов названиям фильмов.

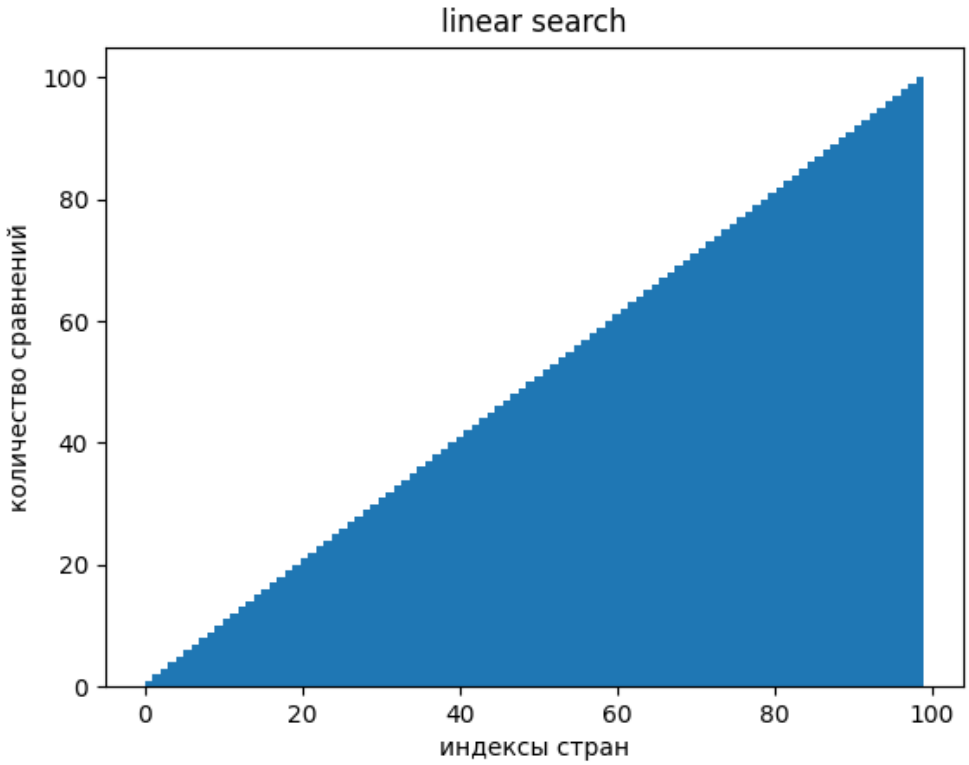

Рисунок 4.1 – График алгоритма линейного поиска.

По рисунку 4.1 можно сделать вывод, что сложность алгоритма линейного поиска зависит от индекса в словаре линейно.

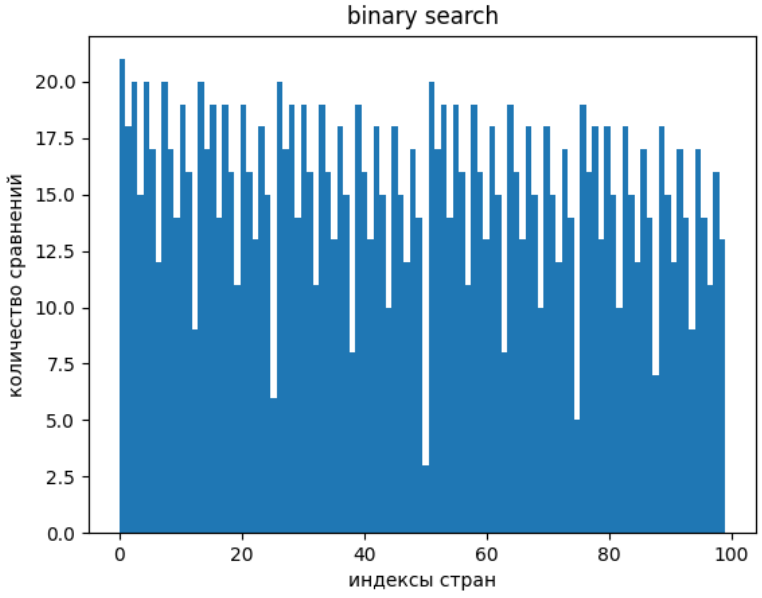

Рисунок 4.2 – График алгоритма бинарного поиска.

Можно заметить, что алгоритму бинарного требуется наименьшее число сравнений, когда элемент находится по центру массива. Из рисунков 4.1 и 4.2

следует, что худшему случаю бинарного поиска требуется в пять раз меньше сравнений, чем алгоритму линейного поиска.

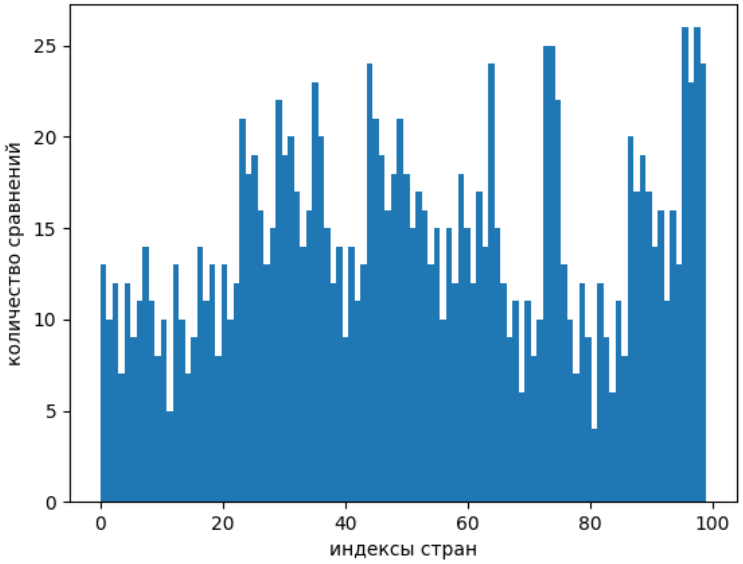

Рисунок 4.3 – График алгоритма поиска по сегментам.

В это случае число сравнений складывается из числа сравнений при линейном поиске сегмента и бинарном поиске в нем. На рисунке 4.3 видно, что нет прямой связи между индексом записи в исходном словаре и числе сравнений при поиске по сегментам.

На рисунках 4.4, 4.5 и 4.6 приведена та же информация, отсортированная по количеству сравнений.

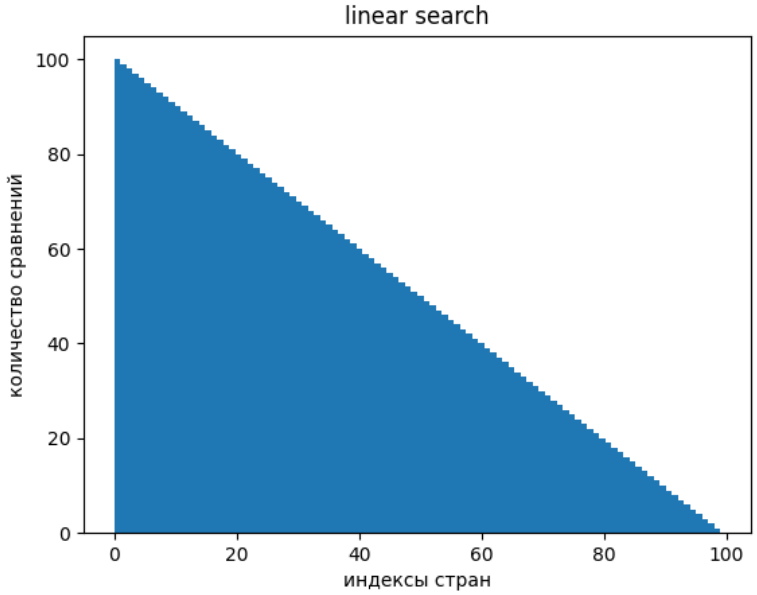

Рисунок 4.4 – График алгоритма линейного поиска.

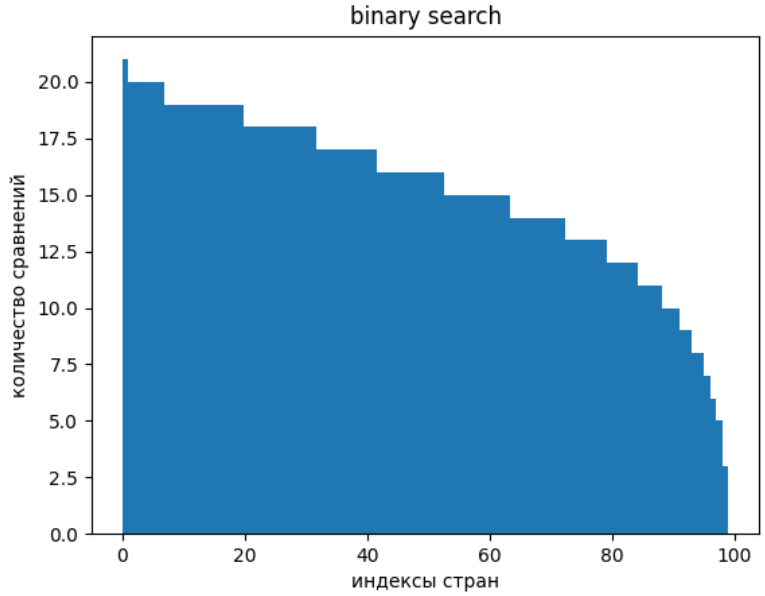

Рисунок 4.5 – График алгоритма бинарного поиска.

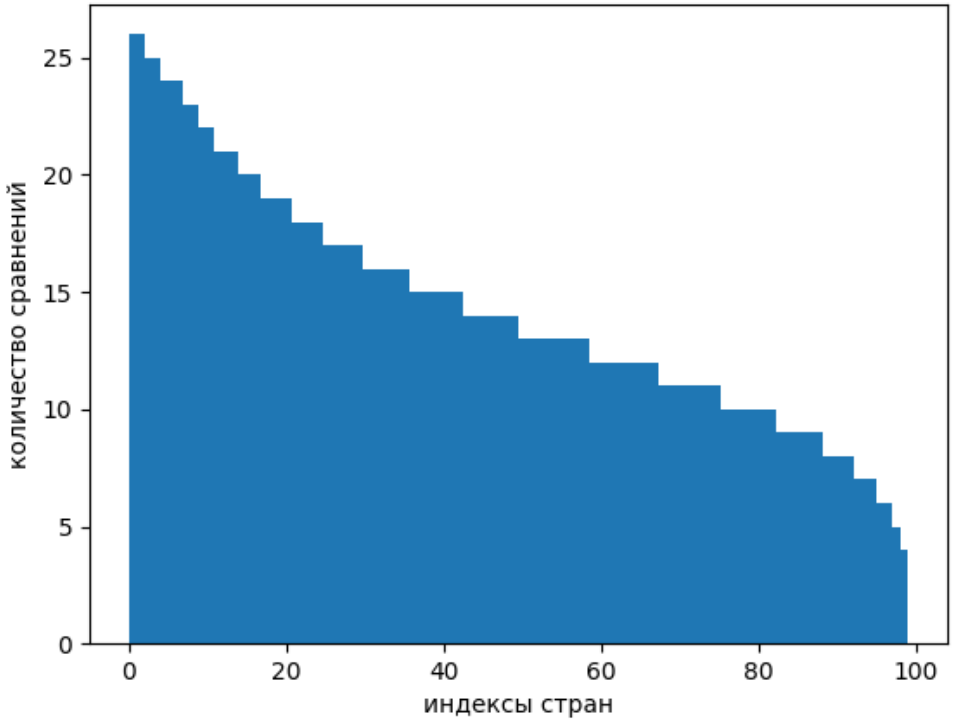

Рисунок 4.6 – График алгоритма поиска по сегментам.

4.3 Вывод

Исходя из полученных данных, можно сделать вывод, что алгоритм поиска в словаре, использующий частотный анализ, является боле эффективным, чем алгоритм полного перебора лишь в ряде случаев, в остальных же, он является менее эффективным, в связи с использованием сегментации и бинарным поиском внутри сегмента.

Отдельно отметим, что алгоритм бинарного поиска требует, в целом, меньшего числа сравнений, в связи с чем является более эффективным, чем алгоритм с частотным анализом. Однако, алгоритм бинарного поиска требует сортировки всего входного массива, что, в среднем случаи имеет сложность $O(nlog(n))$, в связи с чем алгоритм бинарного поиска становится менее эффективным, чем алгоритм частотного анализа, сортирующий данные по сегментам.

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы лабораторной работы была достигнута её цель: разарботан эффективный алгоритм поиска в словаре. Также выполнены следующие задачи:

- исследовать различные алгоритмы поиска;
- оценить трудоёмкости алгоритмов в лучшем случае, худшем и в среднем;
- привести схемы рассматриваемых алгоритмов;
- описать использующиеся структуры данных;
- определить средства реализации разрабатываемого программного обеспечения;
- реализовать три алгоритма поиска в словаре: линейный, двоичный, по сегментам;
- провести тестирование реализованного программного продукта;
- провести анализ алгоритмов по количеству сравнений.

В результате проведения анализа по количеству сравнений было выяснено, что алгоритм поиска по сегментам использует меньше сравнений, чем бинарный и линейный. При этом стоит отметить, что и алгоритм двоичного поиска, и алгоритм поиска по сегментам требуют предварительный обработки данных (сортировки и разбиение на сегменты соответственно), в отличие от алгоритма линейного поиска. Но именно это позволяет стать этим алгоритмам намного эффективнее в сравнении с алгоритмом линейного поиска.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. API по числу заболевших коронавируом людей [Электронный ресурс]. — Режим доступа: `https://covid-19-tracking.p.rapidapi.com` (дата обращения: 17.12.2021).
2. Документация по python [Электронный ресурс]. — Режим доступа: `https://docs.python.org/3/index.html` (дата обращения: 16.10.2021).
3. Документация по json [Электронный ресурс]. — Режим доступа: `https://www.json.org/json-en.html` (дата обращения: 16.10.2021).
4. Документация по json [Электронный ресурс]. — Режим доступа: `https://matplotlib.org` (дата обращения: 16.10.2021).
5. Процессор Intel® Core™ i5-4460 (6 МБ кэш-памяти, до 3,40 ГГц) [Электронный ресурс]. — Режим доступа: `https://www.intel.ru/content/www/ru/ru/products/sku/80817/intel-core-i54460-processor-6m-cache-up-to-3-40-ghz/specifications.html` (дата обращения: 14.09.2021).