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 дисциплине "Анализ алгоритмов"

Тема Поиск по ключу в словаре

Студент Шацкий Р.Е.

Группа ИУ7-55Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

Одной из частых задач в программировании является задача доступа к некоторому набору элементов. Например, обычные списки (массивы) представляют собой набор пронумерованных элементов, то есть для обращения к какому-либо элементу списка необходимо указать его номер. Номер элемента в списке однозначно идентифицирует сам элемент.

Но идентифицировать данные по числовым номерам не всегда оказывается удобно. Например, маршруты поездов в России идентифицируются численно-буквенным кодом (число и одна цифра), также численно-буквенным кодом идентифицируются авиарейсы, то есть для хранения информации о рейсах поездов или самолетов в качестве идентификатора удобно было бы использовать не число, а текстовую строку.

Структура данных, позволяющая идентифицировать ее элементы не по числовому индексу, а по произвольному, называется словарем или ассоциативным массивом. Каждый элемент словаря состоит из двух объектов: ключа и значения. В жизни широко распространены словари, например, привычные бумажные словари (толковые, орфографические, лингвистические). В них ключом является слово-заголовок статьи, а значением — сама статья. Для того, чтобы получить доступ к статье, необходимо указать слово-ключ.

Другой пример словаря, как структуры данных — телефонный справочник. В нем ключом является имя, а значением — номер телефона. И словарь, и телефонный справочник хранятся так, что легко найти элемент словаря по известному ключу (например, если записи хранятся в алфавитном порядке ключей, то легко можно найти известный ключ, например, бинарным поиском), но если ключ неизвестен, а известно лишь значение, то поиск элемента с данным значением может потребовать последовательного просмотра всех элементов словаря.

Особенностью ассоциативного массива является его динамичность: в него можно добавлять новые элементы с произвольными ключами и удалять уже существующие элементы. При этом размер используемой памяти пропорционален размеру ассоциативного массива. Доступ к элементам ассоциативного массива выполняется хоть и медленнее, чем к обычным массивам, но в целом довольно быстро.

Словари нужно использовать в следующих случаях.

1. Подсчет числа каких-то объектов. В этом случае нужно завести словарь, в котором ключами являются объекты, а значениями — их количество.

2. Хранение каких-либо данных, связанных с объектом. Ключи — объекты, значения — связанные с ними данные.

3. Установка соответствия между объектами (например, “родитель—потомок”). Ключ — объект, значение — соответствующий ему объект.

4. Если нужен обычный массив, но при этом максимальное значение индекса элемента очень велико, но при этом будут использоваться не все возможные индексы (так на-

зываемый “разреженный массив”), то можно использовать ассоциативный массив для экономии памяти.

Целью данной лабораторной работы является изучение и сравнение алгоритмов поиска по словарю.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить существующие алгоритмы поиска по словарю;
- определить требования к программе для решения задачи поиска по словарю;
- описать структуры данных;
- определить классы эквивалентности для тестирования;
- определить тестовые данные;
- осуществить выбор средств реализации;
- реализовать алгоритмы программно;
- исследовать время работы алгоритмов;
- сравнить время работы алгоритмов и сделать соответствующие выводы.

1 Аналитическая часть

В данном разделе будут рассмотрены существующие алгоритмы поиска по словарю и определены требования к разрабатываемой программе.

1.1 Алгоритмы поиска по словарю

Для решения задачи поиска по словарю можно использовать следующие алгоритмы:

- алгоритм полного перебора;
- алгоритм двоичного поиска;
- алгоритм частотного анализа.

Рассмотрим каждый из них.

1.1.1 Алгоритм полного перебора

Алгоритмом полного перебора называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты исходного набора данных. В случае словарей будет произведен последовательный перебор элементов словаря до тех пор, пока не будет найден необходимый. Сложность такого алгоритма зависит от количества всех возможных решений, а время работы может стремиться к экспоненциальному.

Пусть алгоритм нашел элемент на первом сравнении. Тогда, в лучшем случае, будет затрачено $k_0 + k_1$ операций, на втором – $k_0 + 2k_1$, на N – $k_0 + Nk_1$. тогда, средняя трудоемкость может быть рассчитано по формуле (1.1), где Ω - множество всех возможных случаев.

$$\sum_{i \in \Omega} p_i t_i = (k_0 + k_1)\frac{1}{N+1} + (k_0 + 2k_1) * \frac{1}{N+1} + \cdots + (k_0 + Nk_1) * \frac{1}{N+1} \tag{1.1}$$

Из (1.1), сгруппировав слагаемые, получим итоговую формулу для расчета средней трудоемкости работы алгоритма:

$$k_0 + k1(\frac{N}{N+1} + \frac{N}{2}) = k_0 + k1(1 + \frac{N}{2} - \frac{1}{N+1}) \tag{1.2}$$

1.1.2 Алгоритм двоичного поиска

Данный алгоритм применяется к заранее упорядоченным словарям. Процесс двоичного поиска можно описать при помощи шагов:

- сравнить значение ключа, находящегося в середине рассматриваемого интервала (изначально – весь словарь), с данным;
- в случае, если значение меньше (в контексте типа данных) данного, продолжить поиск в левой части интервала, в обратном - в правой;
- продолжать до тех пор, пока найденное значение не будет равно данному или длина интервала не станет равной нулю (означает отсутствие искомого ключа в словаре).

Использование данного алгоритма в для поиска в словаре в любом из случаев будет иметь трудоемкость равную $O(log_2(N))$ [1]. Несмотря на то, что в среднем и худшем случаях данный алгоритм работает быстрее алгоритма полного перебора, стоит отметить, что предварительная сортировка больших данных требует дополнительных затрат по времени и может оказать серьезное действие на время работы алгоритма. Тем не менее, при многократном поиске по одному и тому же словарю, применение алгоритм сортировки понадобится всего один раз.

1.1.3 Алгоритм частотного анализа

Алгоритм частотного анализа строит частотный анализ полученного словаря. Чтобы провести частотный анализ, нужно взять первый элемент каждого значения в словаре по ключу и подсчитать частотную характеристику, т.е. сколько раз этот элемент встречался в качестве первого. По полученным данным словарь разбивается на сегменты так, что все записи с одинаковым первым элементом оказываются в одном сегменте.

Сегменты упорядочиваются по значению частотной характеристики таким образом, чтобы к элементу с наибольшим значением характеристики был предоставлен самый быстрый доступ.

Затем каждый из сегментов упорядочивается по значению. Это необходимо для реализации бинарного поиска, который обеспечит эффективный поиск в сегмента при сложности $O(nlog(n))$ таким образом, сначала выбирается нужный сегмент, а затем в нем проводится бинарный поиск нужного элемента. Средняя трудоемкость при длине алфавита M может быть рассчитана по формуле (1.3).

$$\sum_{i \in [1,M]} (f_{select_i} + f_{search_i}) \tag{1.3}$$

1.2 Словарь

В качестве словаря предлагается использовать домен - "Модели телефонов".

Сущностью является модель телефона и компания-производитель, записанные в одной строке через разделитель.

1.3 Вывод

В данном разделе были рассмотрены алгоритмы для решения задачи поиска по словарю.

Для изучения данных алгоритмов необходимо реализовать программное обеспечение, выполняющее следующие функциональные требования:

1. На вход подается словарь из записей, вида {model: string, value: string} для поиска по нему и ключ для поиска в словаре.

2. Выходными данными является найденная в словаре запись для каждого из реализуемых алгоритмов.

Вывод

В данном разделе были описаны принципы работы муравьиного алгоритма и алгоритма полного перебора. Выдвинуты требования к разрабатываемому ПО: входные данные, выходные данные и наличие обработки некорректного ввода.

2 Конструкторская часть

В данном разделе будут описаны структуры данных и алгоритмы для решения задачи поиска по словарю, которые будут реализованы в программе. Кроме того, будут определены классы эквивалентности для тестирования.

2.1 Схемы алгоритмов

На рисунках 2.1, 2.2 и 2.3 представлены схемы использованных алгоритмов.

Рис. 2.1: Схема алгоритма поиска с полным перебором.

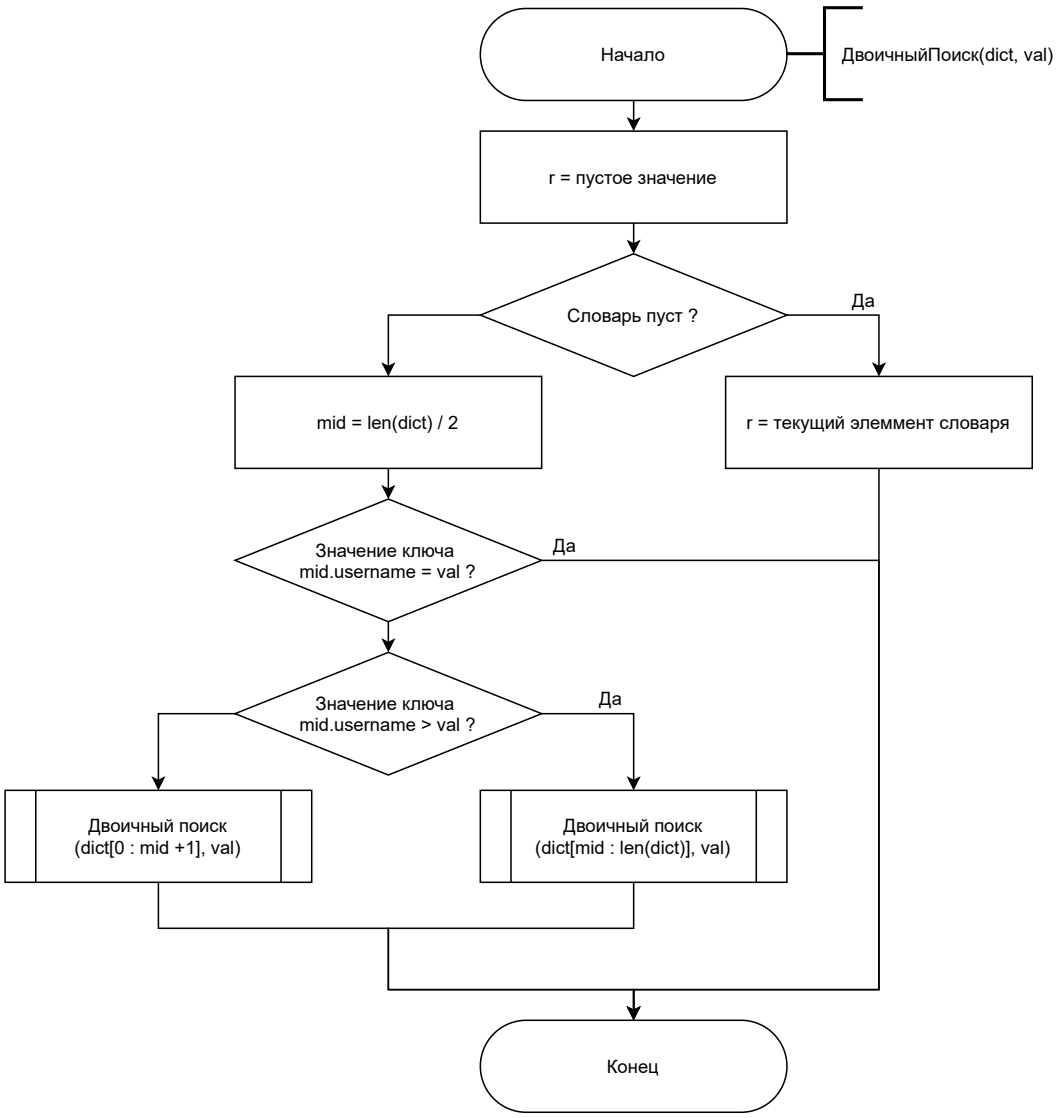

Рис. 2.2: Схема алгоритма с бинарным поиском.

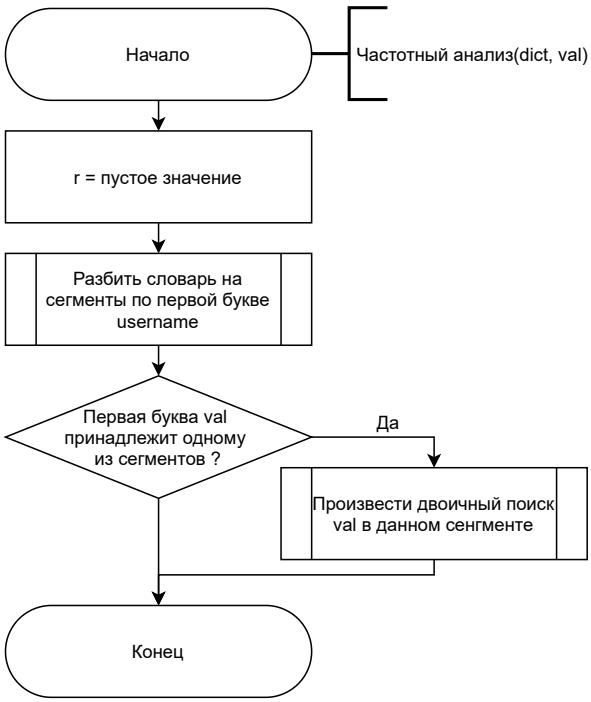

Рис. 2.3: Схема алгоритма с частотным анализом.

2.2 Структуры данных

Для реализации алгоритмов необходимо использовать следующие структуры данных:

1. Структура, описывающая ассоциативный массив с ключами типа string.
2. Структура, описывающая сегмент частотного анализа ассоциативного массива.

Для реализации структуры, описывающей ассоциативный массив, будет использован словарь (dict).

Для реализации структуры, описывающей сегмент частотного анализа, будет использован следующий набор полей:

1. `l` – строка, содержащая признак сегмента – букву из используемого алфавита;
2. `cnt` – частотная характеристика для данного сегмента;
3. `darr` – ассоциативный массив, содержащий записи, соответствующие данному сегменту.

2.3 Тестирование

Для детального тестирования программы, можно выделить несколько классов эквивалентности:

- ключ отсутсвует в словаре;

- ключ является первым элементом словаря;
- ключ является последним элементом словаря;
- ключ является произвольным элементом словаря.

2.4 Структура программного обеспечения

На рисунке 2.4 показана функциональная схема программного обеспечения.

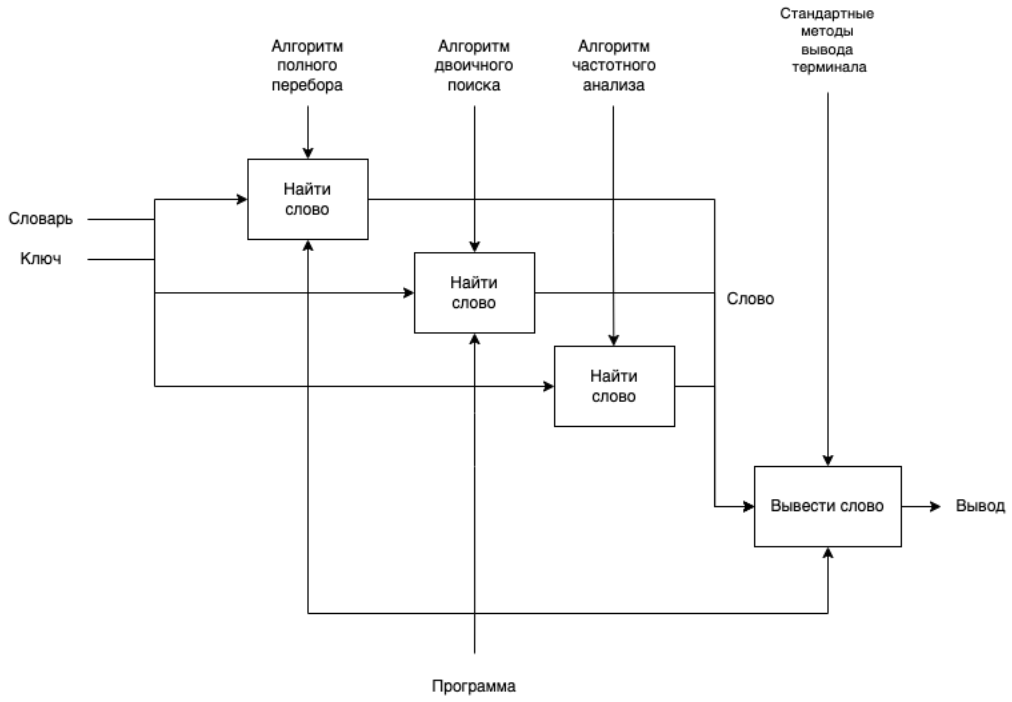

Рис. 2.4: Функциональная схема программного обеспечения, idef0

2.5 Вывод

В данном разделе мы исследовали алгоритмы и структуры данных, которые будут использованы в программе, определили классы эквивалентности для тестирования и определили структуру программного обеспечения.

3 Технологическая часть

В данном разделе будет осуществлен выбор средств реализации, разобран код реализованных алгоритмов и определены тестовые данные.

3.1 Выбор средств реализации

В качестве средства реализации было решено использовать язык программирования Python, так как уже имеется опыт работы с библиотеками и инструментами языка, которые позволяют реализовать и провести исследования алгоритмов.

Для замера времени выполнения использовалась функция time() из библиотеки time.

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- main.py - модуль с тестами и пользовательским вводом;
- search.py - модуль с реализованными классами, наследованными от BaseSearcher, с методами для выполнения поиска в словаре.

3.3 Листинг кода реализованных алгоритмов

В листингах 3.1, 3.2 и 3.3 представлен код используемых алгоритмов.

Листинг 3.1: Алгоритм полного перебора.

```
    class SimpleSearcher(BaseSearcher):
      def __init__(self, _dict: dict[str, str]):
        super().__init__(_dict)

      def search(self, key: str) -> str or None:
        for item in self._dict.items():
          if item[0] == key:
            return item[1]
          return None
```

Листинг 3.2: Алгоритм бинарного поиска.

```
    class BinarySearcher(BaseSearcher):
      def __init__(self, _dict: dict[str, str]):
        super().__init__(_dict)

      def search(self, key: str) -> str or None:
        return BinarySearcher._search(key, self._dict)

      @staticmethod
      def _search(key: str, _dict: dict[str, str]) -> str or None:
        """
        _dict must be already sorted (ascending)

        :param key: key to find in _dict
        :param _dict: ascending sorted dictionary
        :return: _dict value by key
        """
        sorted_keys = list(_dict.keys())
        sorted_values = list(_dict.values())

        down_limit: int = 0
        up_limit: int = len(_dict) - 1
        center_i: int = up_limit // 2

        while sorted_keys[center_i] != key and down_limit < up_limit:
          if key > sorted_keys[center_i]:
            down_limit = center_i + 1
          else:
            up_limit = center_i - 1
          center_i = int((down_limit + up_limit) / 2)

        if down_limit >= up_limit:
          return None
        return sorted_values[center_i]
```

Листинг 3.3: Алгоритм частотного анализа.

```
    class CombinedSearcher(BaseSearcher):
      def __init__(self, _dict: dict[str, str]):
        super().__init__(_dict)
        self.segments: [dict] = self.frequency_analysis()

      def frequency_analysis(self):
        # Сколько раз встречается первый символ каждого ключа  _dict
        frequency_dict: dict[str, int] = {}
        for key in self._dict.keys():
          if key[0] in frequency_dict.keys():
            frequency_dict[key[0]] += 1
          else:
            frequency_dict[key[0]] = 1
        segmented_list: [dict] = []
        for key_letter in frequency_dict.keys():
          keys = {'letter': key_letter, 'count': frequency_dict[key_letter],
              'dict': {}}

          for key in self._dict.keys():
            if key[0] == key_letter:
              keys['dict'][key] = self._dict[key]
          # сортировка словаря по ключу
          keys['dict'] = dict(sorted(keys['dict'].items()))

          segmented_list.append(keys)

        segmented_list.sort(key=lambda value: value['count'], reverse=True)
        return segmented_list

      def search(self, key: str) -> str or None:
        """
        _dict can be any dictionary

        :param key: key to find in _dict
        :param _dict: any dictionary
        :return: _dict value by key
        """
        found_dict = {}
        for i in range(len(self.segments)):
          if self.segments[i]['letter'] == key[0]:
            found_dict = self.segments[i]['dict']
            break

        if len(found_dict) == 0:
          return None

        return BinarySearcher._search(key, found_dict)
```

3.4 Тесты для проверки корректности программы

В рамках данной лабораторной работы будет проведено функциональное тестирование реализованного программного обеспечения.

Тестирование проводилось на словаре, содержащем следующие записи:

- `{key: "C1 -- Nokia"; value:"successfully found"}`;
- `{key: "2.3 -- Nokia"; value:"successfully found"}`;
- `{key: "7.2 -- Nokia"; value:"successfully found"}`.

В Таблице 3.1 приведены соответствующие тесты.

Таблица 3.1: Таблица тестов

Ключ	**Результат**	**Ожидаемое значение**
`C1 -- Nokia`	"successfully found"	"successfully found"
`7.2 -- Nokia`	"successfully found"	"successfully found"
`asd`	**None**	**None**

При проведении функционального тестирования, результаты работы программы совпали с ожидаемыми. Таким образом, функциональное тестирование пройдено успешно.

3.5 Вывод

В данном разделе было выбрано средство реализации, разобран код реализованных алгоритмов и определены тестовые данные.

4 Экспериментальная часть

В данном разделе будет исследовано время работы алгоритмов:

- алгоритм полного перебора;
- алгоритм двоичного поиска;
- алгоритм частотного анализа.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows [2] 10 64-bit;
- оперативная память: 16 Гб;
- процессор: Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz [3].

4.2 Временные характеристики

В реализованном программном обеспечении особый интерес представляет количество сравнений, необходимое для нахождения записи в словаре по ключу, так как это характеристика прямо пропорциональна времени работы алгоритма. Поэтому, будет проводиться анализ количества сравнений каждого из ключей словаря в словаре и время поиска ключа не находящегося в словаре. Для большей наглядности число записей в словаре возьмем равным 2000.

Поиск будет производиться каждым алгоритмом по отдельности, после чего будут представлены соответствующие гистограммы. Отметим, что, в общем случае, распределение носит достаточно "случайный" характер, так как исходный массив не обязательно является упорядоченным. В связи с этим, каждая из построенных гистограмм будет продублирована второй, отображающей данные по убыванию числа сравнений.

4.2.1 Алгоритм полного перебора

В данном алгоритме зависимости числа сравнений от количества элементов является линейной. На пример, для обнаружения первого элемента понадобится 1 сравнение, для 2ого – 2, а для N-ого – N. В связи с этим, поиск последнего элемента в массиве потребует N сравнений.

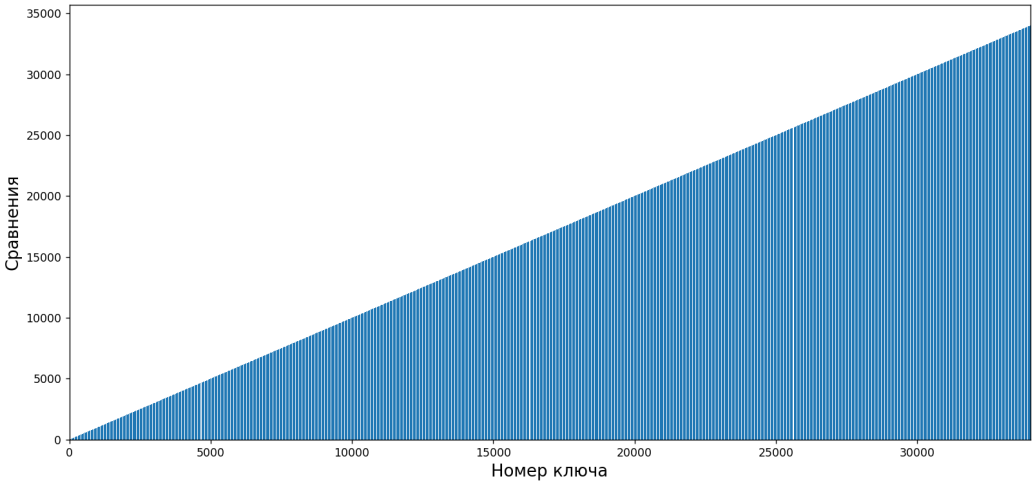

Рис. 4.1: Число сравнений для нахождения полным перебором.

На Рисунке 4.1 видна линейная зависимость числа сравнений от положения элемента в массиве.

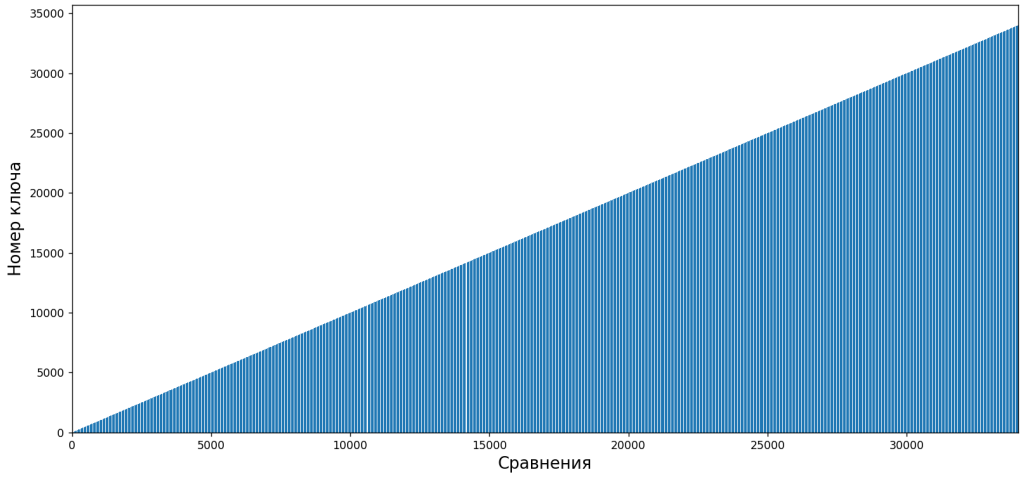

Рис. 4.2: Число сравнений для нахождения полным перебором по убыванию.

4.2.2 Алгоритм бинарного поиска

В данном алгоритме сложность поиска равна $O(log(N))$, в связи с этим, на общей гистограмме будет наблюдаться достаточно произвольное распределение, зависящее от входного массива.

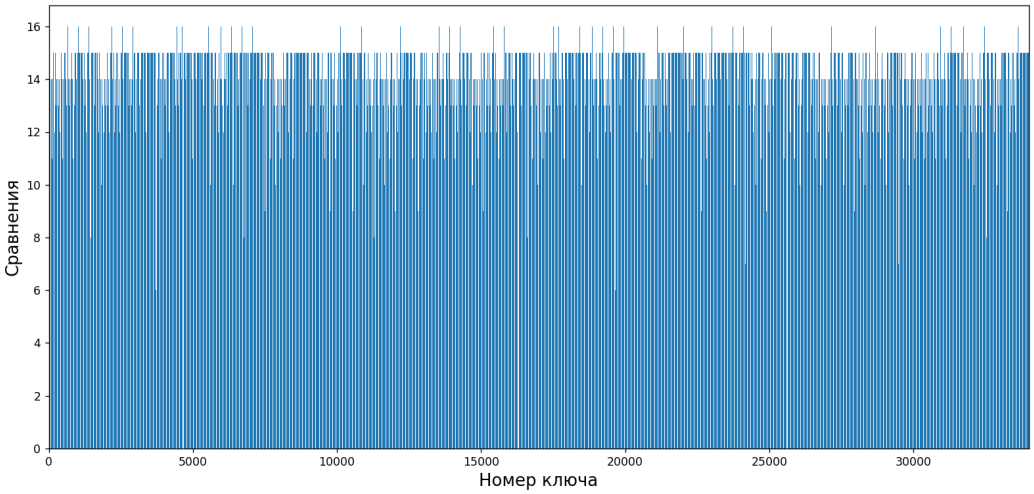

Рис. 4.3: Число сравнений для нахождения бинарным поиском.

На Рисунке 4.3 не наблюдается явной зависимости числа сравнений для поиска элемента от его значения. В связи с этим, стоит упорядочить элементы по числу сравнений и построить вторую гистограмму.

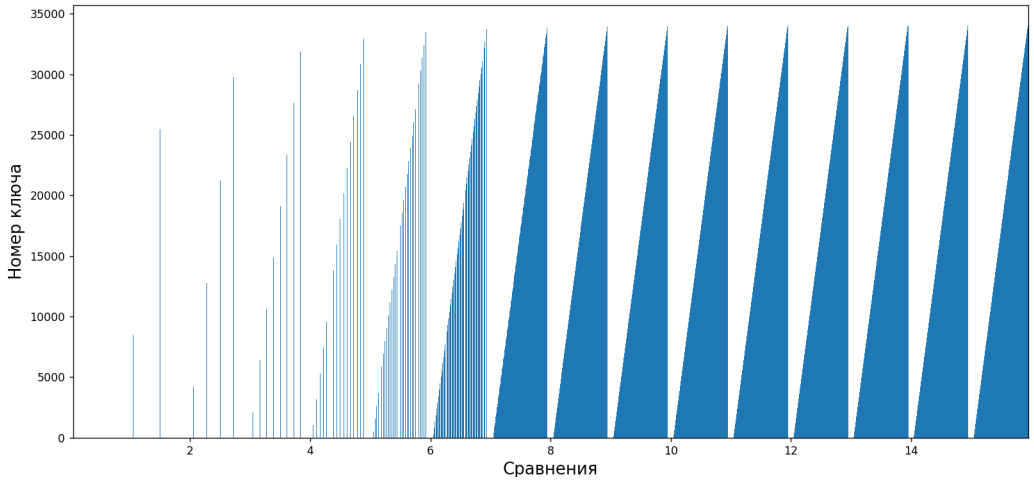

Рис. 4.4: Число сравнений для нахождения бинарным поиском по убыванию.

4.2.3 Алгоритм частотного анализа

В данном алгоритме сложность поиска зависит не только от положения в отсортированном сегменте, но и от размера используемого словаря, так как поиск в нем осуществляется полным перебором. Тем не менее, в среднем случае это не должно оказывать серьезного действия на число требуемых сравнений.

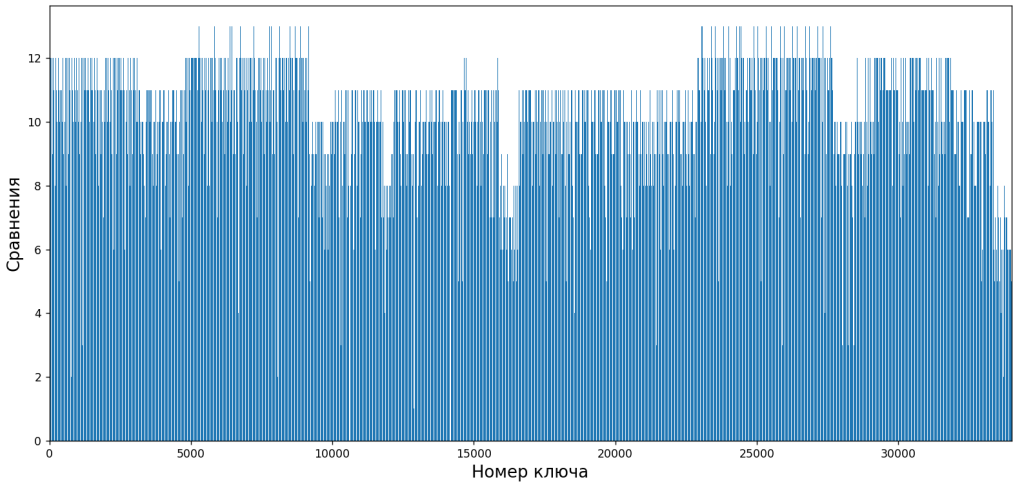

Рис. 4.5: Число сравнений для нахождения частотным анализом.

Как и в случае с бинарным поиском, не наблюдается явной зависимости числа сравнений для поиска элемента от его значения. В связи с этим, стоит так же упорядочить элементы по числу сравнений и построить вторую гистограмму.

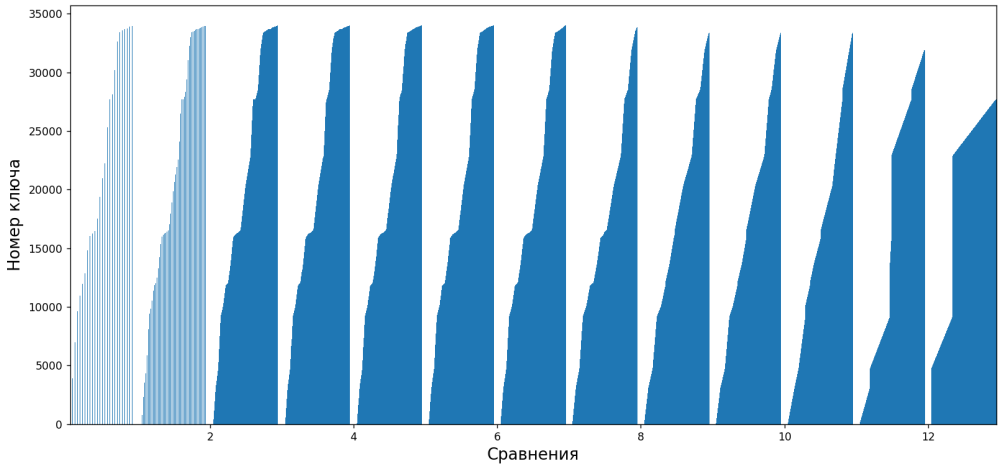

Рис. 4.6: Число сравнений для нахождения частотным анализом по убыванию.

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 надо отметить, что алгоритм бинарного поиска требует 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 случае имеет сложность $O(nlog(n))$, в связи с чем алгоритм бинарного поиска становится менее эффективным, чем алгоритм частотного анализа, сортирующий данные по сегментам.

Заключение

Задачами данной работы являлись:

- изучить существующие алгоритмы поиска по словарю;
- определить требования к программе для решения задачи поиска по словарю;
- описать структуры данных;
- определить классы эквивалентности для тестирования;
- определить тестовые данные;
- осуществить выбор средств реализации;
- реализовать алгоритмы программно;
- исследовать время работы алгоритмов;
- сравнить время работы алгоритмов и сделать соответствующие выводы.

Все задачи были выполнены в ходе работы.

Первым этапом было дано определение поиску по словарю.

Затем были исследованы существующие алгоритмы решения задачи поиска и были выбраны 3 алгоритма для сравнения:

- алгоритм полного перебора;
- алгоритм двоичного поиска;
- алгоритм частотного анализа.

Далее были определены требования к программе, составлены блок-схемы алгоритмов и произведена разработка на языке Python.

В итоге было произведено исследование поиска по словарю с помощью трех алгоритмов.

По результатам исследования 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.

Литература

[1] Бинарный поиск [Электронный ресурс]. Режим доступа: https://progcpp.ru/search-binary/. Дата обращения 18.12.2021.

[2] Сравните выпуски Windows 10 [Электронный ресурс]. Режим доступа: https://www.microsoft.com/ru-ru/windows/compare-windows-10-home-vs-pro. Дата обращения: 21.09.2021.

[3] Процессор Intel® Core™ i5-3550 [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/97150/intel-core-i57600-processor-6m-cache-up-to-4-10-ghz.html. Дата обращения: 21.09.2021.