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 Козлова И.В.

Группа ИУ7-52Б

Оценка (баллы) _____

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

Словарь, как тип данных, применяется везде, где есть связь “ключ - значение” или “объект - данные”: поиск налогов по ИНН и другое. Поиск - основная задача при использовании словаря. Данная задача решается различными способами, которые дают различную скорость решения.

Со временем стали разрабатывать алгоритмы поиска в словаре. В данной лабораторной работе мы рассмотрим три алгоритма.

1. Поиск полным перебором.
2. Бинарный поиск.
3. С помощью сегментов.

Цель данной работы: получить навык работы со словарём, как структурой данных, реализовать алгоритмы поиска по словарю (указаны выше) с применением оптимизаций.

Для достижения цели поставлены следующие задачи.

1. Изучить три алгоритма поиска в словаре.
2. Привести схемы алгоритмов поиска в словаре.
3. Описать структуру разрабатываемого программного обеспечения.
4. Применить изученные основы для реализации поиска значений в словаре по ключу.
5. Провести функциональное тестирование разработанного алгоритма.
6. Получить замеры количества сравнений для каждого ключа (для всех трех алгоритмов).
7. Провести сравнительный анализ по времени для реализованных алгоритмов.
8. Подготовить отчет по лабораторной работе.

1 Аналитическая часть

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

1.1 Словарь как структура данных

Словарь (или "*ассоциативный массив*") [1] - абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- `INSERT(k, v)`;
- `FIND(k)`;
- `REMOVE(k)`.

В паре `(k, v)`: `v` называется значением, ассоциированным с ключом `k`. Где `k` — это ключ, а `v` — значение. Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

Операция `ПОИСК(k)` возвращает значение, ассоциированное с заданным ключом, или некоторый специальный объект `НЕ_НАЙДЕНО`, 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).

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов — например, строки.

В данной лабораторной работе в качестве ключа будет использоваться строка: название города, а в качестве значения – строка: страна, в которой расположен город.

1.2 Алгоритм полного перебора

Алгоритмом полного перебора [2] называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты. В случае реализации алгоритма в рамках данной работы будут последовательно перебираться ключи словаря до тех пор, пока не будет найден нужный.

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

Пусть на старте алгоритм затрагивает k_0 операций, а при сравнении k_1 операций.

Пусть алгоритм нашёл элемент на первом сравнении (лучший случай), тогда будет затрачено $k_0 + k_1$ операций, на втором - $k_0 + 2 \cdot k_1$, на последнем (худший случай) - $k_0 + N \cdot k_1$. Если ключа нет в массиве ключей, то мы сможем понять это, только перебрав все ключи, таким образом трудоёмкость такого случая равно трудоёмкости случая с ключом на последней позиции. Средняя трудоёмкость может быть рассчитана как математическое ожидание по формуле (1.1), где Ω – множество всех возможных случаев.

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

1.3 Алгоритм поиска в упорядоченном словаре двоичным поиском

Бинарный поиск базируется на том, что словарь изначально отсортирован, что позволяет сравнивать ключ с средним элементом, и, если, он меньше, то продолжать искать в левой части, таким же методом, иначе в правой.

Таким образом, при двоичном поиске [3] обход можно представить деревом, поэтому трудоёмкость в худшем случае составит $k_0 + \log_2 N$ (в худшем случае нужно спуститься по двоичному дереву от корня до листа).

Лучшим случаем будет случай, если искомый ключ окажется средним элементом, тогда трудоемкость будет равна $k_0 + \log_2 1$.

Скорость роста функции $\log_2 N$ меньше, чем скорость линейной функции, полученной для полного перебора.

1.4 Поиск с помощью сегментов

Алгоритм на вход получает словарь. Далее словарь разбивается на сегменты так, что все элементы с некоторым общим признаком попадают в один сегмент (для букв это может быть первая буква, для чисел - остаток от деления).

Сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементам с наибольшей частотной характеристикой был самый быстрый доступ. Например, такой характеристикой может быть размер сегмента.

Вероятность обращения к определенному сегменту равна сумме вероятностей обращений к его ключам, то есть $P_i = \sum_j p_j = N \cdot p$, где P_i – вероятность обращения к i-ому сегменту, p_j - вероятность обращения к j-ому элементу, который принадлежит i-ому сегменту.

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

Далее ключи в каждом сегменте сортируются для того, чтобы внутри каждого сегмента можно было использовать бинарный поиск, который обеспечит эффективный поиск со сложностью $O(\log_2 m)$ (где m - количество ключей в сегменте) внутри сегмента.

Таким образом, сначала выбирается нужный сегмент, а затем в нем проводится бинарный поиск нужного элемента.

Средняя трудоёмкость при множестве всех возможных случаев Ω может быть рассчитана по формуле (1.2).

$$\sum_{i \in \Omega} (f_{\text{выбор сегмента i-ого элемента}} + f_{\text{бинарный поиск i-ого элемента}}) \cdot p_i \tag{1.2}$$

Худший случай – это случай, когда выбор сегмента – $N \cdot p_i$, при этом N равно числу сегментов. А поиск элемента (ключа) в данном сегменте – $\log_2 N$, где N – число элементов в сегменте.

Лучший случай – если искомый сегмент оказывается первым, и в этом сегменте исходный ключ – серединный элемент, так как бинарный поиск сначала укажет в середину.

1.5 Вывод

В данном разделе был рассмотрен абстрактный тип данных словарь и возможные реализации алгоритмов поиска в нём.

2 Конструкторская часть

В этом разделе будут представлено описание используемых типов данных, а также схемы алгоритмов поиска в словаре.

2.1 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие типы данных:

- словарь – встроенный тип dict [4] в Python[5] будет использован в созданном классе Dictionary;
- массив ключей – встроенный тип list [6] в Python[5];
- длина массива/словаря – целое число int.

2.2 Структура разрабатываемого ПО

В данном ПО буде реализован метод структурного программирования, при этом также будет реализован класс Dictionary для работы со словарем.

Взаимодействие с пользователем будет через консоль, будет дана возможность ввода ключа для поиска значений в словаре.

Для работы будут разработаны следующие процедуры:

- главная процедура – является точкой входа в программу, входных данных нет, выходных данных нет;
- процедура замера времени – замеряет время работы всех алгоритмов поиска в словаре, выводит график зависимости времени от ключа, входные данные – словарь (объект класса Dictionary), выходных данных нет;
- процедура построения гистограмм, входные данные – словарь (объект класса Dictionary), выходные данных нет.

2.3 Схемы алгоритмов

На рисунке 2.1 представлена схема алгоритма поиска в словаре полным перебором.

Рисунок 2.1 – Схема алгоритма поиска в словаре полным перебором

На рисунке 2.2 представлена схема алгоритма бинарного поиска в словаре.

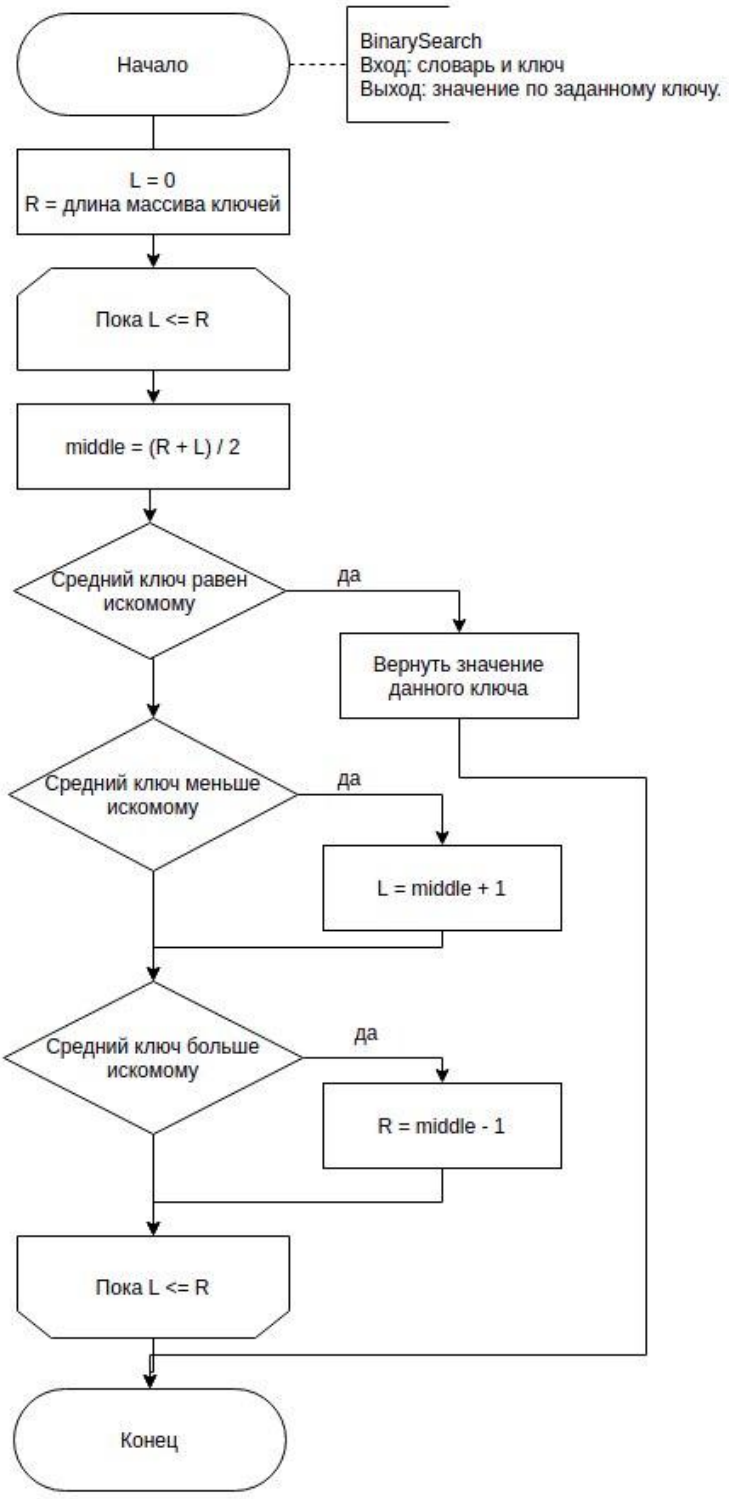

Рисунок 2.2 – Схема алгоритма бинарного поиска в словаре

На рисунке 2.3 представлена схема поиска в словаре с помощью сегментирования.

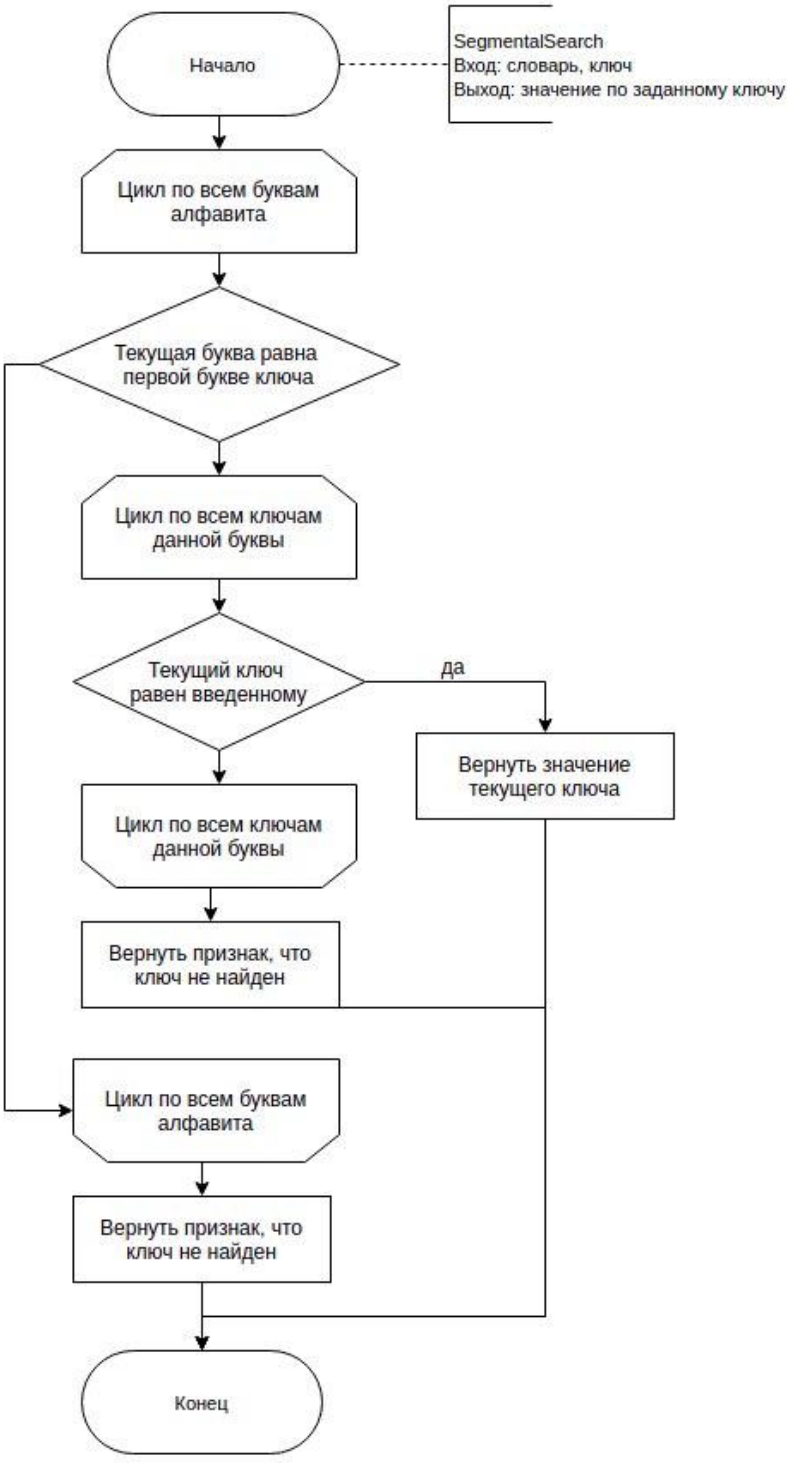

Рисунок 2.3 – Схема алгоритма поиска в словаре с помощью сегментирования

2.4 Классы эквивалентности при тестировании

Для тестирования выделены классы эквивалентности, представленные ниже.

1. Некорректный ввод ключа – пустая строка.
2. Корректный ввод, но ключа нет в словаре – вывод будет (-1), как знак того, что такого ключа нет словаре.
3. Корректный ввод и ключ есть в словаре – вывод верного значения.

2.5 Вывод

В данном разделе были построены схемы алгоритмов, рассматриваемых в лабораторной работе, были описаны классы эквивалентности для тестирования, структура программы.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаётся ключ, значение которого необходимо найти;
- на выходе – три результата поиска в словаре (для каждого алгоритма выводится отдельный результат).

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран ЯП Python [5].

Данный язык достаточно удобен и гибок в использовании.

Время работы алгоритмов было замерено с помощью функции process_time() из библиотеки time [7]

В качестве среды разработки выбор сделан в сторону Visual Studio Code [8]. Данная среда подходит как для Windows, так и для Linux.

3.3 Сведения о модулях программы

Данная программа разбита на модули:

- main.py – файл, содержащий меню программы и основную функцию;
- info_for_prog.py – файл, содержащий класс, который описывает информацию о данных;
- dictionary.py – файл, содержащий класс "Словарь".

3.4 Реализация алгоритмов

В листинге 3.1 представлена реализация алгоритма поиска в словаре полным перебором.

Листинг 3.1 – Реализация алгоритма поиска полным перебором

```
def BruteForceSearch(self, key):
  k = 0
  kk = list(self.data.keys())
  for elem in self.data:
    k += 1
    if key == elem:
      // Writing to the log file
      self.f.write(f"{kk.index(key)},{key},{k}\n")
      return self.data[elem]
  return -1
```

В листинге 3.2 представлена реализация алгоритма бинарного поиска в словаре.

Листинг 3.2 – Реализация алгоритма бинарного поиска

```
def BinarySearch(self, key, list_keys):
  l, r = 0, len(list_keys) - 1
  k = 0
  kk = list(self.data.keys())
  while l <= r:
    middle = (r + l)
    elem = list_keys[middle]
    k += 1
    if elem == key:
      // Writing to the log file
      self.f1.write(f"{kk.index(key)},{key},{k}\n")
      return self.data[elem]
    elif elem < key:
      l = middle + 1
    else:
      r = middle - 1
  return -1
```

В листинге 3.3 представлена реализация алгоритма поиска в словаре с помощью разбиения на сегменты.

Листинг 3.3 – Реализация алгоритма поиска в словаре с помощью разбиения на сегменты

```
def SegmentalSearch(self, key, new_dict):
  c = 0
  kk = list(self.data.keys())
  for k in new_dict:
    c += 1
    if key[0] == k:
      c += 1
      for elem in new_dict[k]:
        if elem == key:
          // Writing to the log file
          self.f2.write(f"{kk.index(key)},{key},{c}\n")
          return new_dict[k][elem]
      return -1
  return -1
```

3.5 Функциональное тестирование

В данном разделе будет приведена таблица с тестами (таблица 3.1).

Таблица 3.1 – Таблица тестов

Входные данные	Пояснение	Результат
Moscow	Средний элемент	Ответ верный
Tokyo	Первый элемент	Ответ верный
Belgium	Последний элемент	Ответ верный
123	Несуществующий элемент	Ответ верный (-1)
Mosccow	Несуществующий элемент	Ответ верный (-1)

Все тесты пройдены успешно для всех алгоритмов.

3.6 Вывод

В данном разделе были представлены листинги рассматриваемых алгоритмов поиска в словаре, приведена информация о средствах реализации, сведения о модулях программы и было проведено функциональное тестирование.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программа, а также проведен сравнительный анализ алгоритмов для разных ключей на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись эксперименты:

- операционная система: Ubuntu 20.04.3 [9] Linux [10] x86_64;
- память: 8 ГБ;
- процессор: 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz [11].

Эксперименты проводились на ноутбуке, включенном в сеть электропитания. Во время экспериментов ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Демонстрация работы программы

На рисунках 4.1 и 4.2 представлены результаты работы программы.

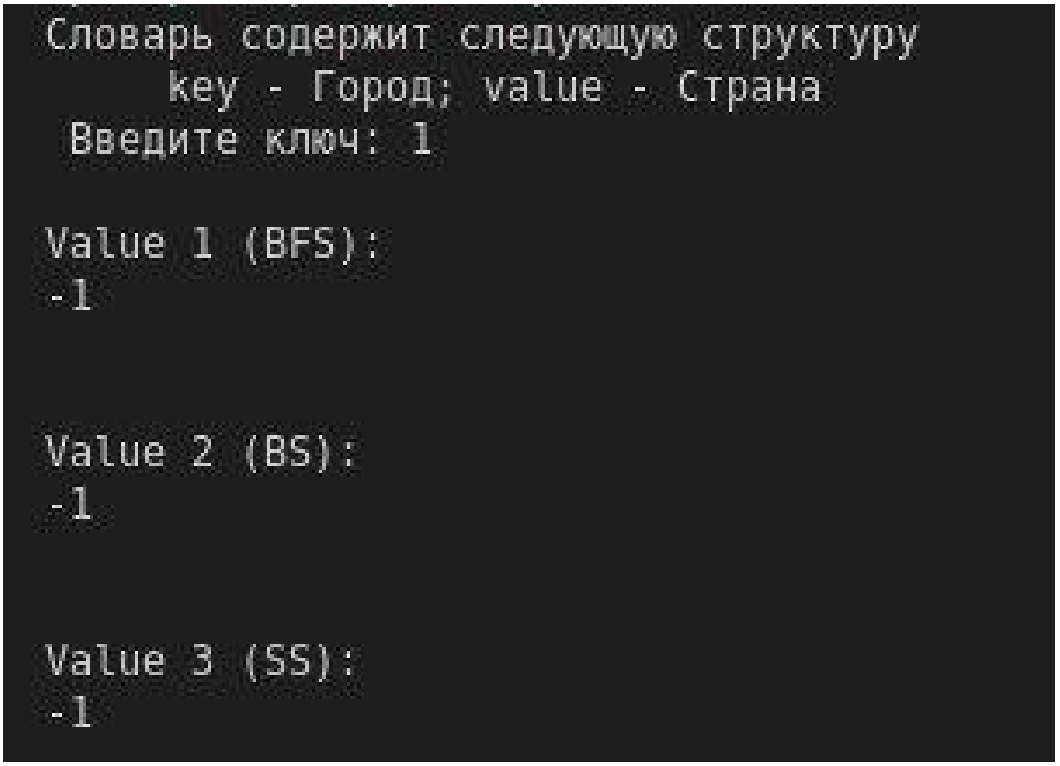

Рисунок 4.1 – Пример 1 работы программы

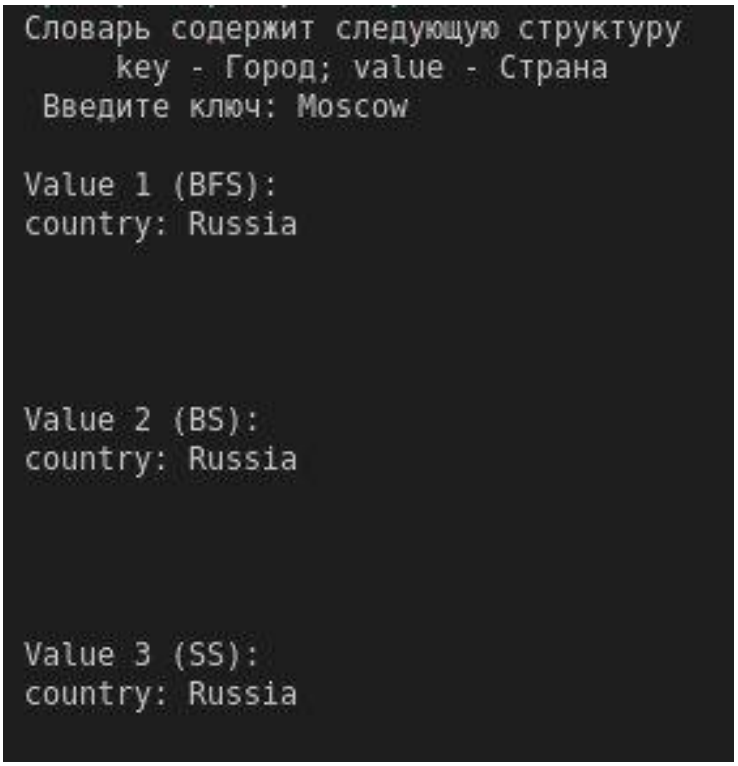

Рисунок 4.2 – Пример 2 работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, для замера времени выполнения части кода используется функция process_time() из библиотеки time [7].

На рисунке 4.3 представлен график зависимости времени поиска от индекса ключа словаря, для построения которого использовались данные о времени поиска каждого элемента в словаре(2000 элемент). Индекс ключа указан на горизонтальной оси.

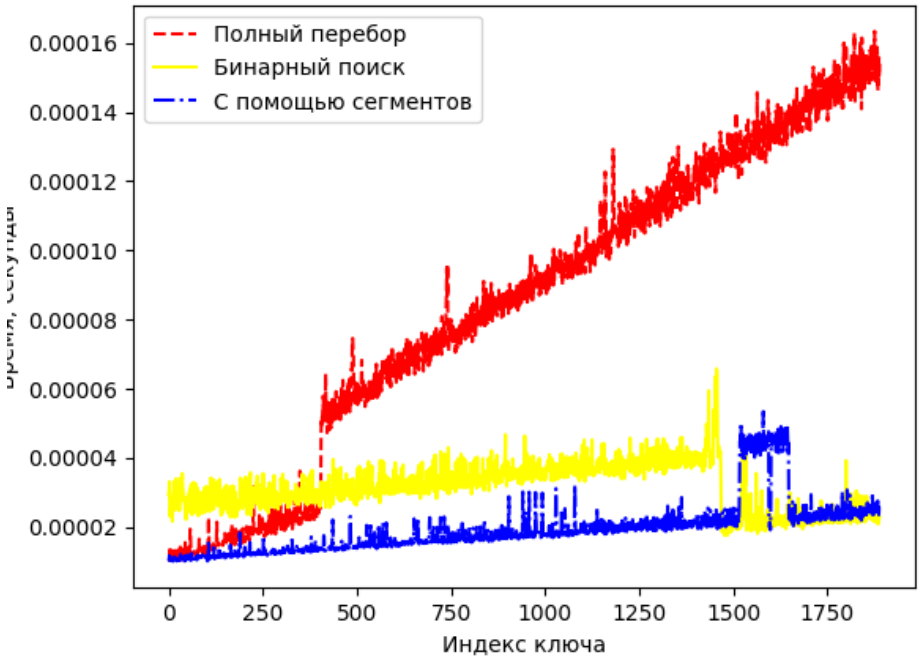

Рисунок 4.3 – Время работы алгоритмов для поиска каждого ключа словаря

На рисунке 4.4 представлен график на меньшем количестве точек для большей наглядности.

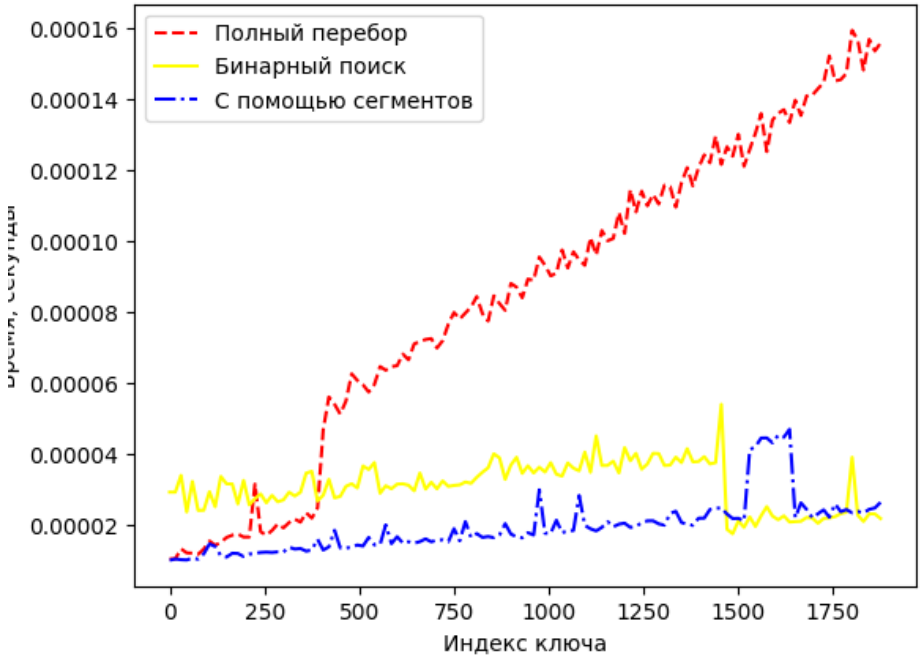

Рисунок 4.4 – Время работы алгоритмов для поиска каждого ключа словаря

4.4 Количество сравнений

В ходе эксперимента было подсчитано количество сравнений для каждого ключа в словаре, и на основе полученных данных составлены гистограммы.

Гистограммы составлены двух видов для каждого алгоритма:

- ключи отсортированы в лексикографическом порядке (для словаря, рассматриваемого в данной работе);
- ключи расположены в том порядке, в котором они идут в самом словаре.

Гистограммы для алгоритма поиска в словаре полным перебором представлены на рисунке 4.5.

Гистограммы для алгоритма бинарного поиска в словаре представлены на рисунке 4.6.

Гистограммы для алгоритма поиска в словаре с помощью разбиения на сегменты представлены на рисунке 4.7.

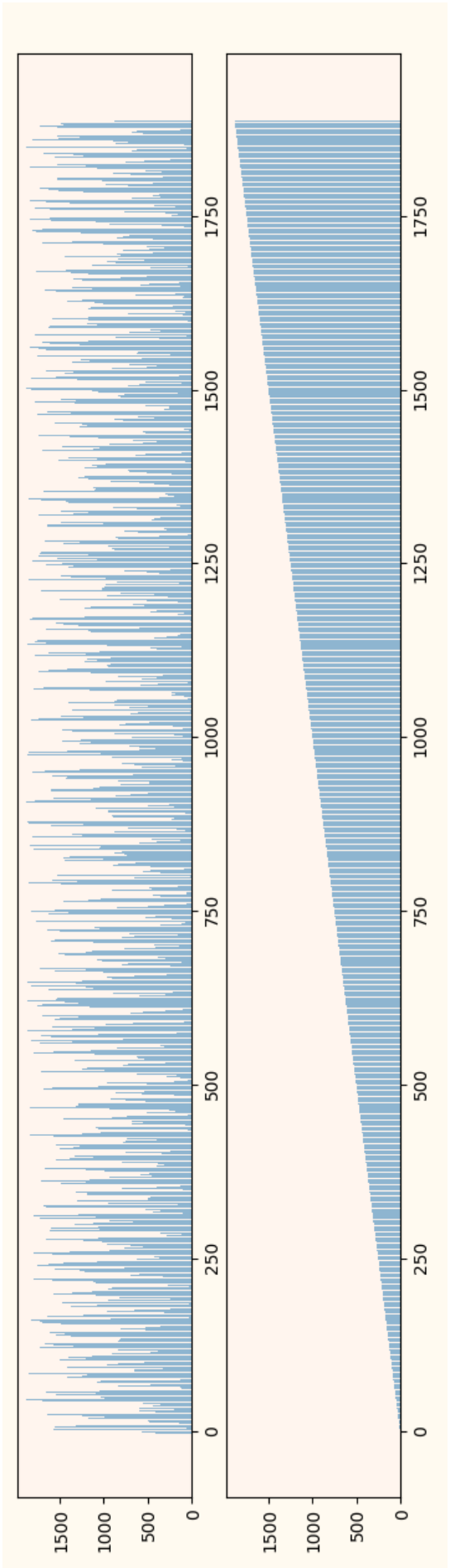

Рисунок 4.5 – Поиск с помощью полного перебора

Рисунок 4.6 – Бинарный поиск

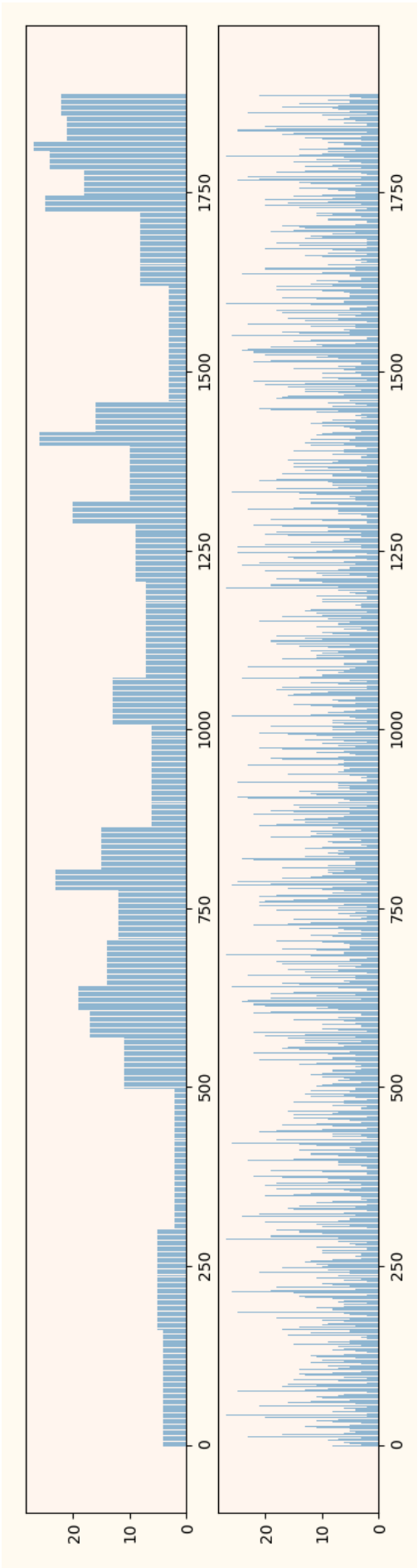

Рисунок 4.7 – С помощью сегментов

4.5 Вывод

В результате эксперимента и полученных графиков и гистограмм, приведенных выше, видно, что самый медленный алгоритм - алгоритм полного перебора. Время в нём растет линейно и увеличивается с увеличением индекса элемента словаря. Также растет количество сравнений, что напрямую зависит от времени работы программы.

Алгоритм бинарного поиска и алгоритм поиска с помощью сегментов тратят примерно одинаковое количество времени, при этом они требуют дополнительных расходов времени на подготовку данных к работе с алгоритмом, но эти расходы малы.

Заключение

В результате эксперименты было определено, что алгоритм полного перебора работает медленнее всего. Алгоритм бинарного поиска и алгоритм поиска с помощью сегментов тратят примерно одинаковое количество времени, но они требуют дополнительного времени на первичную обработку данных (выделение сегментов и сортировка).

В рамках выполнения работы были выполнены следующие задачи:

- Изучены три алгоритма поиска в словаре.
- Приведены схемы алгоритмов поиска в словаре.
- Описаны структуру разрабатываемого программного обеспечения.
- Применены изученные основы для реализации поиска значений в словаре по ключу.
- Проведено функциональное тестирование разработанного алгоритма.
- Получены замеры количества сравнений для каждого ключа (для всех трех алгоритмов).
- Проведен сравнительный анализ по времени для реализованных алгоритмов.
- Подготовлен отчет по лабораторной работе.

Поставленная цель достигнута.

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] Коршунов Ю. М. Коршуном Ю. М. Математические основы кибернетики // Энергоатомиздат. 1972.

[4] dict Python [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/2to3.html?highlight=dict#to3fixer-dict (дата обращения: 04.09.2021).

[5] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 04.09.2021).

[6] list Python [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/pdb.html?highlight=list#pdbcommand-list (дата обращения: 04.09.2021).

[7] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#functions (дата обращения: 04.09.2021).

[8] Visual Studio Code [Электронный ресурс]. Режим доступа: https://code.visualstudio.com/ (дата обращения: 30.09.2021).

[9] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: https://releases.ubuntu.com/20.04/ (дата обращения: 04.09.2021).

[10] Linux – Википедия [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Linux (дата обращения: 04.09.2021).

[11] Процессор Intel® Core™ i5-1135G7 [Электронный ресурс]. Режим доступа: https://www.

intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz/specifications.html (дата обращения: 04.09.2021).