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»

Отчёт

по лабораторной работе № 7

Название: Поиск слов по словарю

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б _____ И. Е. Афимин
(Группа) (Подпись, дата) (И.О. Фамилия)

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

Москва, 2020

Содержание

Введение

Словарь – книга или любой другой источник, информация в котором упорядочена с помощью разбивки на небольшие статьи, отсортированные по названию или тематике. Различают энциклопедические и лингвистические словари. С развитием компьютерной техники всё большее распространение получают электронные словари и онлайн-словари. Первым русским словарём принято считать Азбуковник, помещённый в списке Кормчей книги 1282 года и содержащий 174 слова. Задача состоит в поиске слов из словаря в случайных данных любого размера(напр. в файле). Поскольку словарь меняется редко, то можно его подготовить (напр. отсортировать, создать дерево итд). Это зависит от алгоритма поиска, который будет использован.

Целью данной лабораторной работы является реализация алгоритмов поиска слов в словаре и исследование их трудоемкости.

Задачи данной лабораторной работы:

1) описать алгоритм полного перебора;
2) описать алгоритм двоичного поиска;
3) описать алгоритм поиска слов по сегментам;
4) реализовать 3 алгоритма поиска по словарю;
5) провести замеры времени работы алгоритмов.

1 Аналитический раздел

В данном разделе будут рассмотрены алгоритмы поиска слов в слове.

1.1 Алгоритм полного перебора

Алгоритм полного перебора – это алгоритм разрешения математических задач, который можно отнести к классу способов нахождения решения рассмотрением всех возможных вариантов. Полный перебор (или метод «грубой силы», англ. brute force) – метод решения математических задач. Относится к классу методов поиска решения исчерпыванием всевозможных вариантов. Сложность полного перебора зависит от количества всех возможных решений задачи. Если пространство решений очень велико, то полный перебор может не дать результатов в течение нескольких лет или даже столетий.

В данном случае следует перебирать слова в словаре, пока не встретится нужное слово, следовательно, время работы оценивается как $O(n)$.

1.2 Алгоритм двоичного поиска

Целочисленный двоичный поиск (бинарный поиск) (англ. binary search) – алгоритм поиска объекта по заданному признаку в множестве объектов, упорядоченных по тому же самому признаку, работающий за логарифмическое время.

Принцип двоичный поиска заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где может находится искомый объект. В зависимости от постановки задачи, процесс может останавливается, когда получен первый или же последний индекс вхождения элемента. Последнее условие – это левосторонний-/правосторонний двоичный поиск.

1.3 Алгоритм поиска по сегментам

Суть данного алгоритма заключается в том, что необходимо разбить словарь на сегменты. Каждый сегмент определяет первую букву слов, которые находятся в нем. Для того, чтобы найти слово в таком словаре необходимо определить сегмент, где может находиться слово, а после произвести поиск в данном сегменте.

2 Констукторский раздел

В данном разделе будут рассмотрены схемы алгоритмов, требования к функциональности ПО, и опредены способы тестирования.

2.1 Разработка алгоритмов

Ниже будут представлены схемы алгоритмов поиска:

1) алгоритм поиска полным перебором (рисунок 2.1);
2) алгоритм двоичного поиска (рисунок 2.2);
3) алгоритм поиска по сегментам (рисунок 2.3).

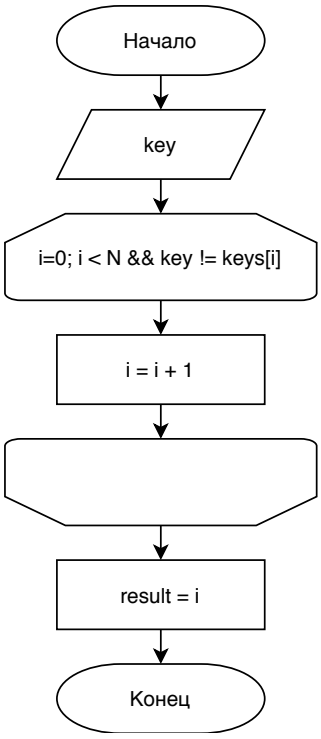

Рисунок 2.1 — Схема алгоритма поиска полным перебором

2.2 Требования к функциональности ПО

В данной работе требуется обеспечить следующую минимальную функциональность консольного приложения:

1) загрузка словаря из текстового файла;
2) вывод замеров времени работы каждого из алгоритмов в текстовый файл.

2.3 Тестирование

Тестирование ПО будет проводиться методом чёрного ящика. Необходимо проверить работу системы на случаях, когда словарь является пустым, содержит один и более элементов.

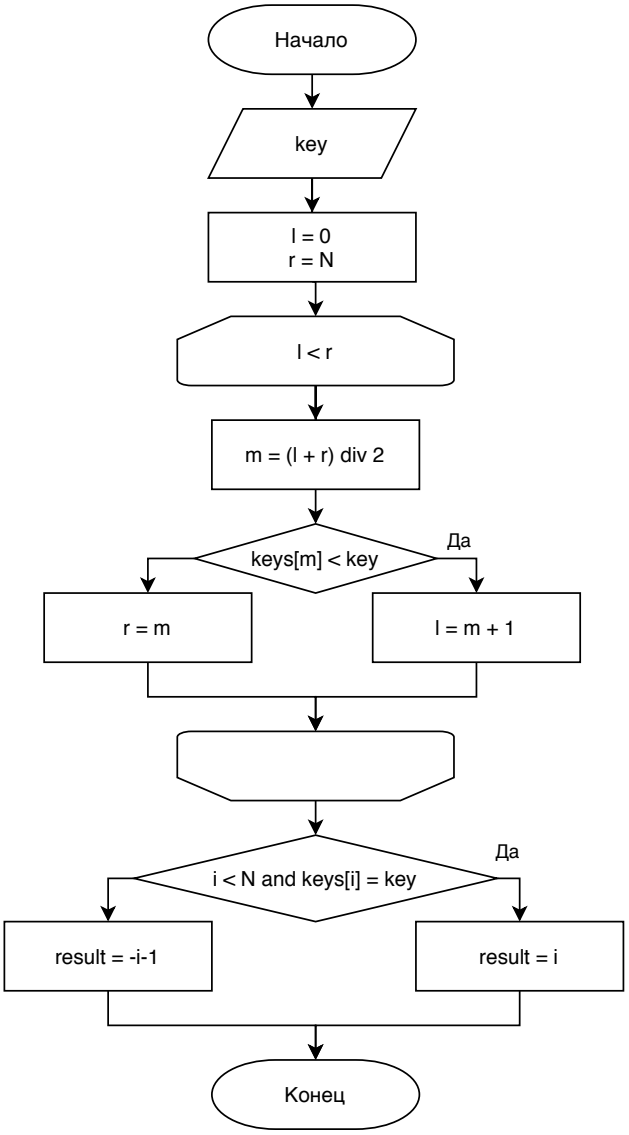

Рисунок 2.2 — Схема алгоритма бинарного поиска

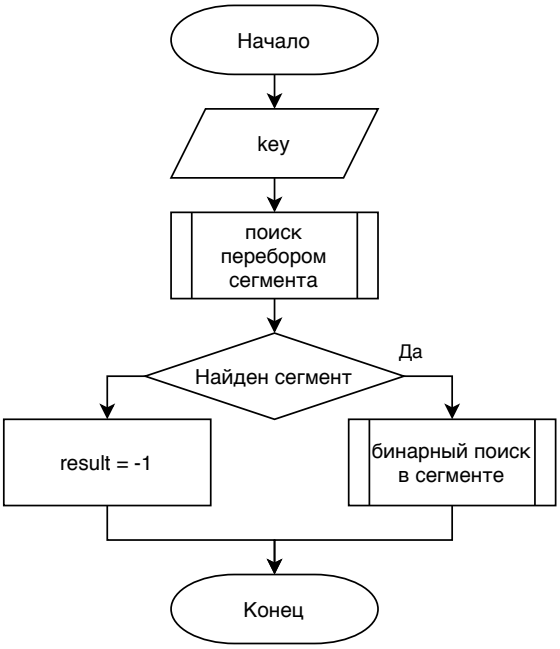

Рисунок 2.3 — Схема алгоритма поиска по сегментам

3 Технологический раздел

В данном разделе будут выбраны средства реализации ПО и представлен листинг кода.

3.1 Средства реализации

В данной работе используется язык программирования python [2], так как он позволяет написать программу в относительно малый срок. В качестве среды разработки использовался Jupyter Notebook.

Для замера процессорного времени была использована функция process_time модуля time. Она возвращает значение в долях секунды суммы системного и пользовательского процессорного времени текущего процесса и не включает время, прошедшее во время сна.

3.2 Листинг программы

Ниже представлены листинги кода алгоритмов поиска слова в словаре:

1) полным перебором (листинг 3.1);
2) бинарным поиском (листинг 3.2);
3) поиском по сегментам (листинг 3.3).

Листинг 3.1 — Реализация алгоритма поиска слов в словаре полным перебором

```
class BruteForceDictionary:
    "Словарь с поиском ключа перебором"
    def __init__(self):
        self.data = []

    def keys(self):
        return list(self.__iter__())

    def __getitem__(self, key : str) -> int:
        i = self.__get_index_key(key)
        if i > -1:
            return self.data[i][1]
        return None

    def __setitem__(self, key: str, value: int):
        i = self.__get_index_key(key)
        if i < 0:
            self.data.append((key, value))
        else:
            self.data[i] = (key, value)

    def __contains__(self, key: str):
        return self.__get_index_key(key) > -1

```

```
25      def __iter__(self):
26          return iter(map(lambda pair: pair[0], self.data))
27
28      def __get_index_key(self, key: str) -> int:
29          i = 0
30          for i, pair in enumerate(self.data):
31              if pair[0] == key:
32                  return i
33          return -i - 1
```

Листинг 3.2 — Реализация алгоритма двоичного поиска слова в словаре

```
1   class BinarySearchDictionary:
2       "Словарь с двоичным поиском ключа"
3       def __init__(self):
4           self.data = []
5           self.n = 0
6
7       def keys(self):
8           return list(self.__iter__())
9
10      def __getitem__(self, key : str) -> int:
11          i = self.__get_index_key(key)
12          if i >= 0:
13              return self.data[i][1]
14          return None
15
16      def __setitem__(self, key: str, value: int):
17          i = self.__get_index_key(key)
18          if i < 0:
19              self.data.insert(-i-1, (key, value))
20              self.n += 1
21          else:
22              self.data[i] = (key, value)
23
24      def __contains__(self, key: str):
25          return self.__get_index_key(key) >= 0
26
27      def __iter__(self):
28          return iter(map(lambda pair: pair[0], self.data))
29
30      def __get_index_key(self, key: str) -> int:
31          left = 0
32          right = self.n
33
34          while left < right:
35              mid = (left + right) // 2
```

```
            if self.data[mid][0] < key:
                left = mid + 1
            else:
                right = mid
        if left < self.n and self.data[left][0] == key:
            return left
        else:
            return -left - 1
```

Листинг 3.3 — Реализация алгоритма поиска слова в словаре по сегментам

```
class SegmentSearchDictionary:
    "Словарь с поиском ключа по сегментам"
    def __init__(self):
        self.segments = BruteForceDictionary()

    def sort_segments(self, chars):
        def cmp(key):
            i = 0
            for i, char in enumerate(chars):
                if char == key[0]:
                    return i
            return i + 1

        self.segments.data.sort(key=cmp)

    def __getitem__(self, key : str) -> str:
        segment = self.segments[key[0]]
        if segment:
            return segment[key]
        return None

    def __setitem__(self, key: str, value: str):
        segment = self.segments[key[0]]

        if not segment:
            segment = BinarySearchDictionary()
            self.segments[key[0]] = segment
        segment[key] = value

    def __contains__(self, key: str):
        seg = self.segments[key[0]]
        if seg:
            return seg[key]
        else:
            return False

```

```
37      def keys(self):
38          keys = []
39          for key in self.segments:
40              keys += self.segments[key].keys()
41          return keys
42
43      def __iter__(self):
44          iter(self.keys())
```

3.3 Тестирование

В таблице 3.1 отображён возможный набор тестов для тестирования методом чёрного ящика, результаты которого, представленные на рисунке 3.1, подтверждают прохождение программы перечисленных тестов.

Таблица 3.1 — Тесты для проверки корректности программы

Словарь	Слово	Ожидаемый результат
{ }	1	Не найдено
{'1': 2}	1	2
{'2': 1, '1': 2}	1	2
{'2': 1, '1': 2}	3	Не найдено

```
bruteForceDictionary    = { } word:  1 value: Не найдено
binarySearchDictionary  = { } word:  1 value: Не найдено
segmentSearchDictionary = { } word:  1 value: Не найдено

bruteForceDictionary    = {'1': 2 } word:  1 value: 2
binarySearchDictionary  = {'1': 2 } word:  1 value: 2
segmentSearchDictionary = {'1': 2 } word:  1 value: 2

bruteForceDictionary    = {'2':1, '1': 2 } word:  1 value: 2
binarySearchDictionary  = {'2':1, '1': 2 } word:  1 value: 2
segmentSearchDictionary = {'2':1, '1': 2 } word:  1 value: 2

bruteForceDictionary    = {'2':1, '1': 2 } word:  3 value: Не найдено
binarySearchDictionary  = {'2':1, '1': 2 } word:  3 value: Не найдено
segmentSearchDictionary = {'2':1, '1': 2 } word:  3 value: Не найдено
```

Рисунок 3.1 — Результаты тестирования алгоритмов.

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для сравнительного анализа трёх алгоритмов по затрачиваемому процессорному времени в зависимости от индекса слова в словаре. Тестирование проводилось на сервере под управлением Ubuntu Linux (64-bit) с 1 Гб оперативной памяти.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

В рамках данного проекта были проведёны эксперименты по замеру времени работы алгоритмов поиска слова в словаре:

1) полным перебором (график 4.1);
2) двоичным поиском (график 4.2);
3) поиском по сегментам (график 4.3).

4.2 Статистический анализ замеров

По графику 4.1 видно, что преимущественно время поиска линейно, но возможны случайные увеличения времени поиска в связи с выполнением сервером других процессов. В худших случаях алгоритму требуется 42 секунды на поиск последего слова из набора или же несуществующего.

На рисунке 4.4 приведён график плотности распределения времени поиска слова в словаре. В среднем бинарному поиску требуется 0.03577 секунд со среднеквадратичным отклонением в 0.02748 секунд. В то время как алгоритму поиска по сегментам необходимо 0.03097 со среднеквадратичным отклонением в 0.0194 секунд. Такой малый разброс значенией вызван распределением слов по первым буквам в словаре по сегментам (график 4.5). Из-за этого время поиска остаётся примерно одинаковым.

4.3 Вывод

В ходе экспериментов по замеру времени работы было установлено, что самым эффективным и стабильным является поиск по сегментам. Самым долгим является алгоритм полного перебора. Его время возрастает каждый раз из-за того, что слова находятся дальше в словаре, а каждый раз поиск начинается с самого начала. По графикам алгоритма бинарного поиска и поиска по сегментам можно увидеть нормальное распределение времени поиска слова в словаре. На данных тестовых данных

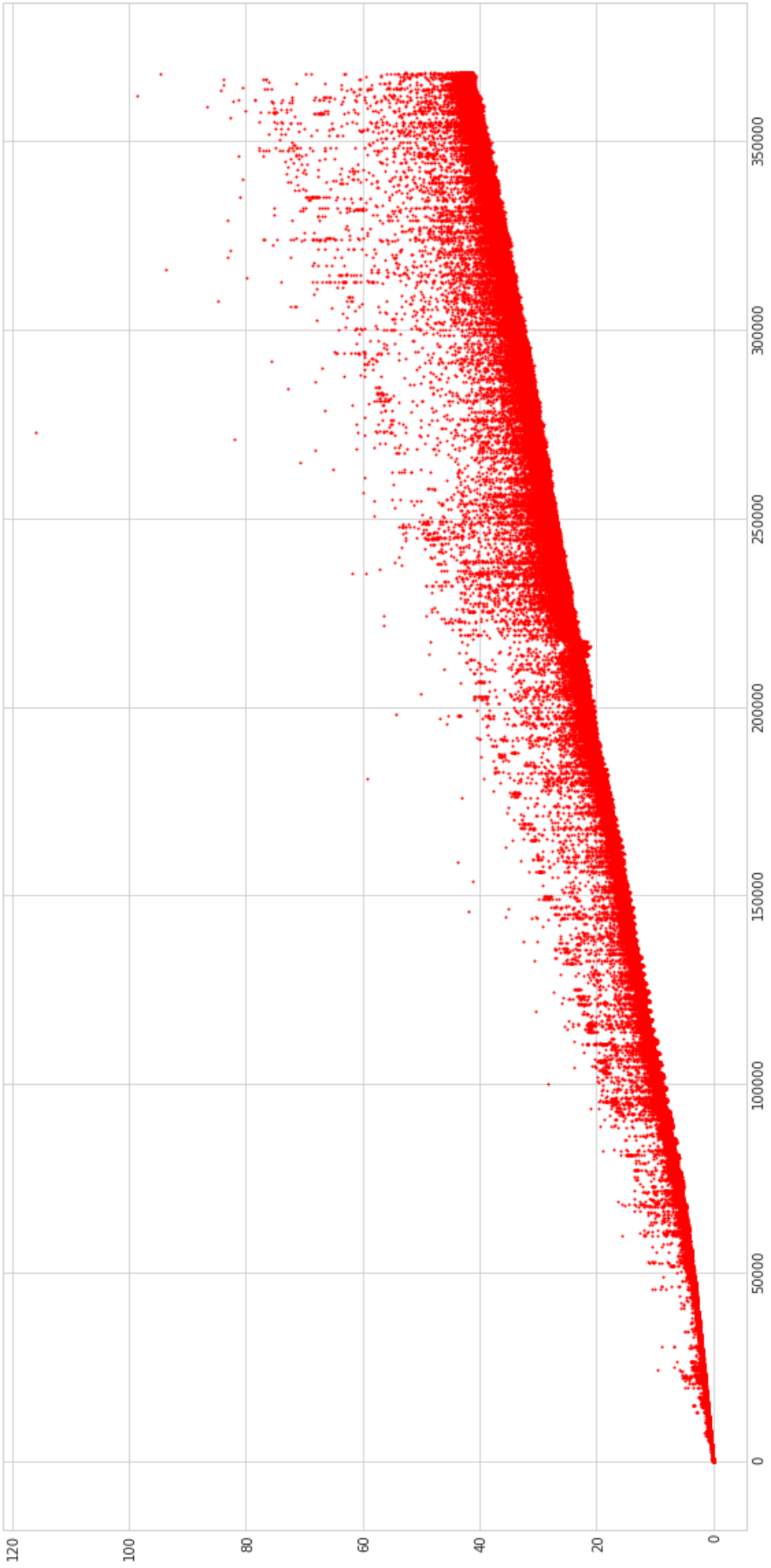

Рисунок 4.1 — Время поиска слова в словаре полным перебором

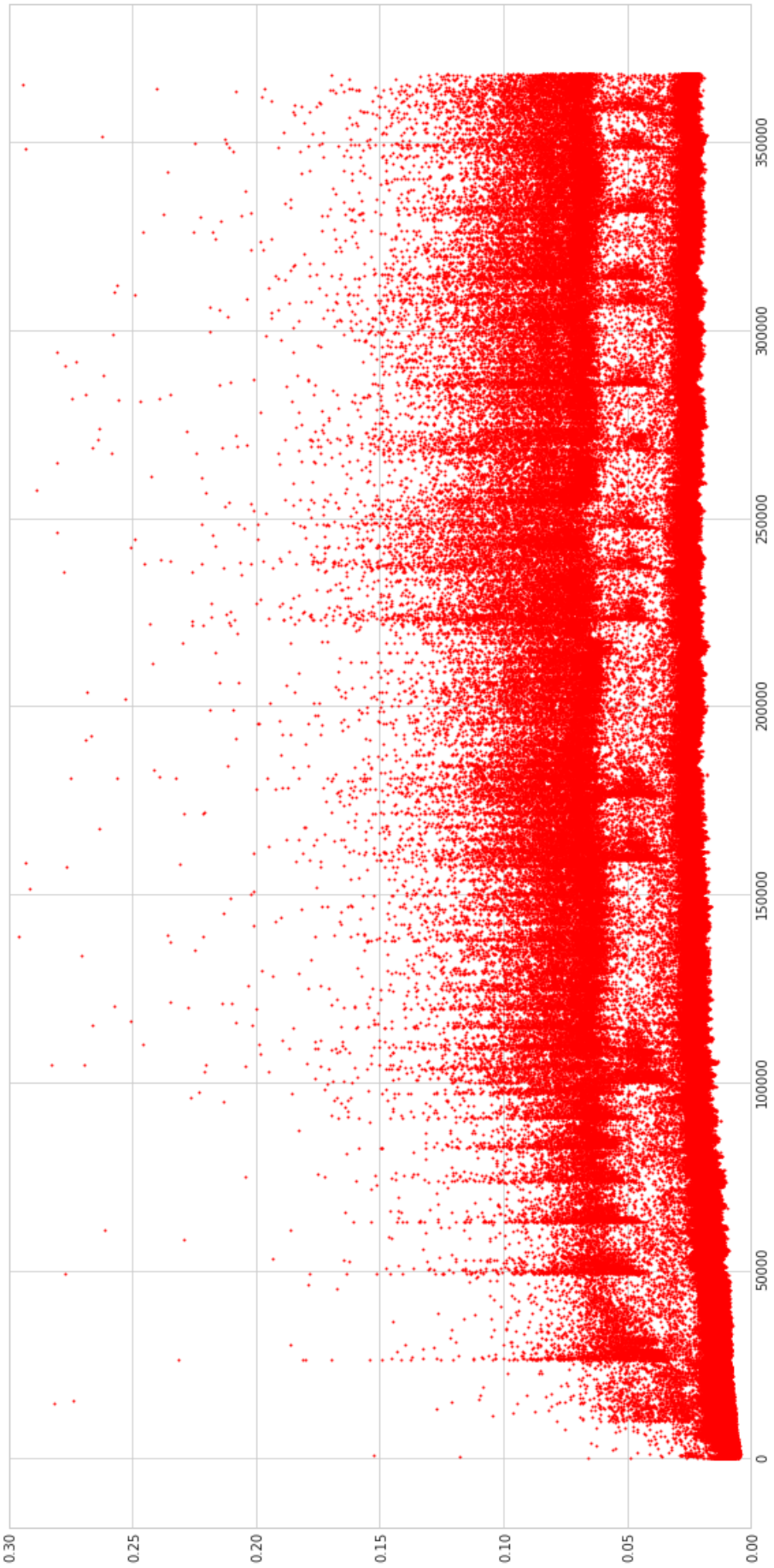

Рисунок 4.2 — Время двоичного поиска слова в словаре

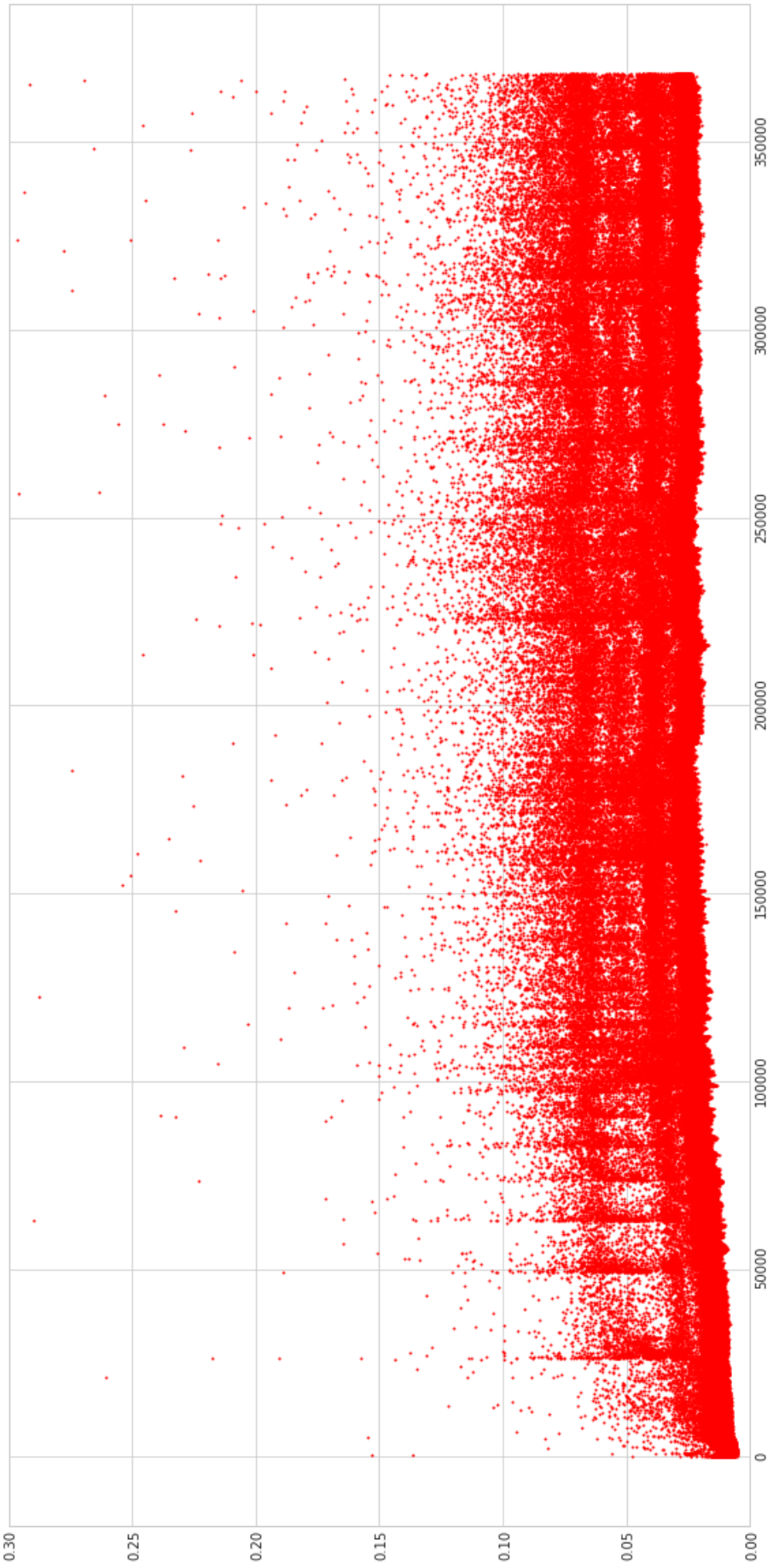

Рисунок 4.3 — Время поиска слова по сегментам в словаре

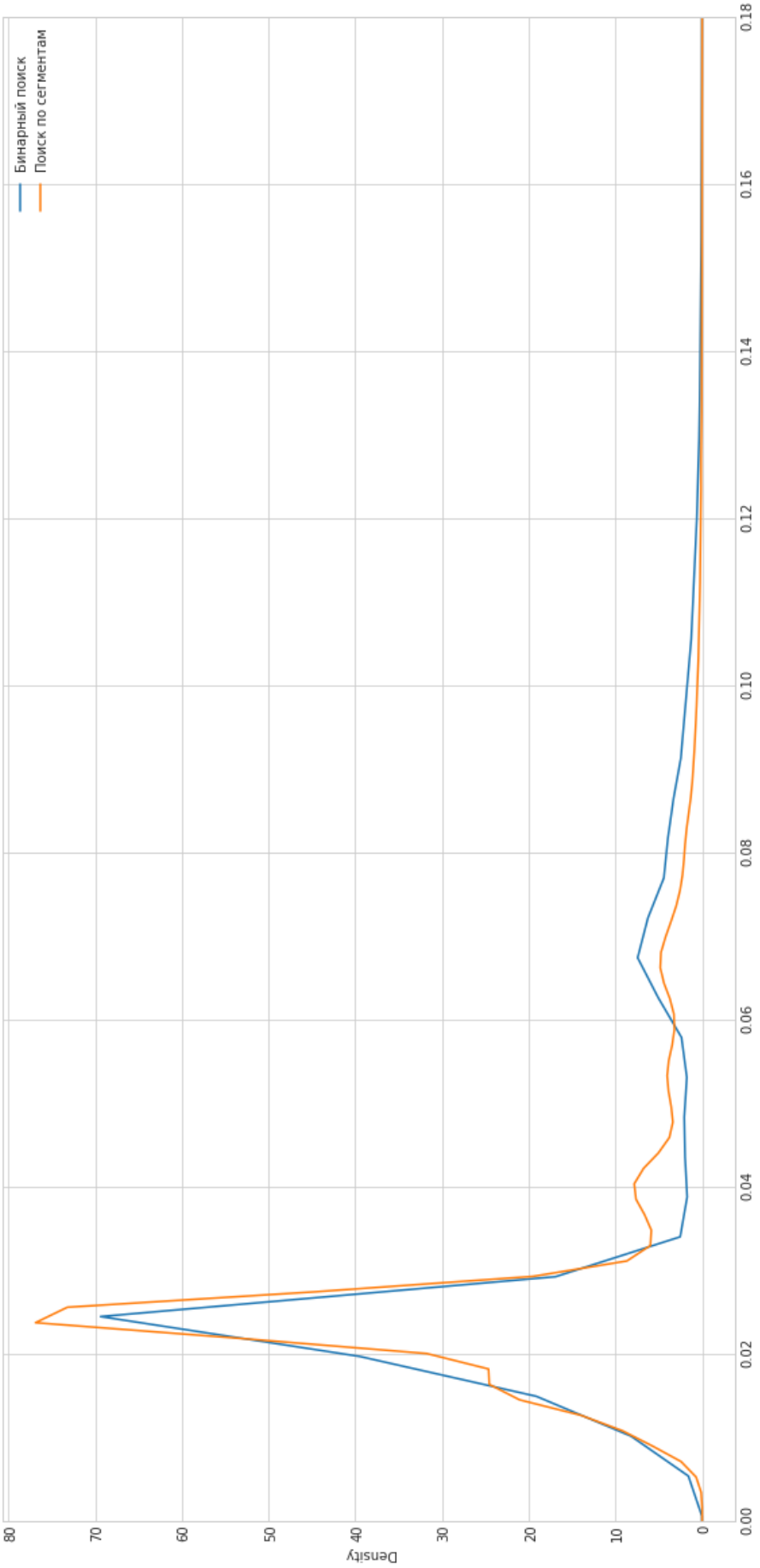

Рисунок 4.4 — Плотности распределения времени поиска слова в словаре

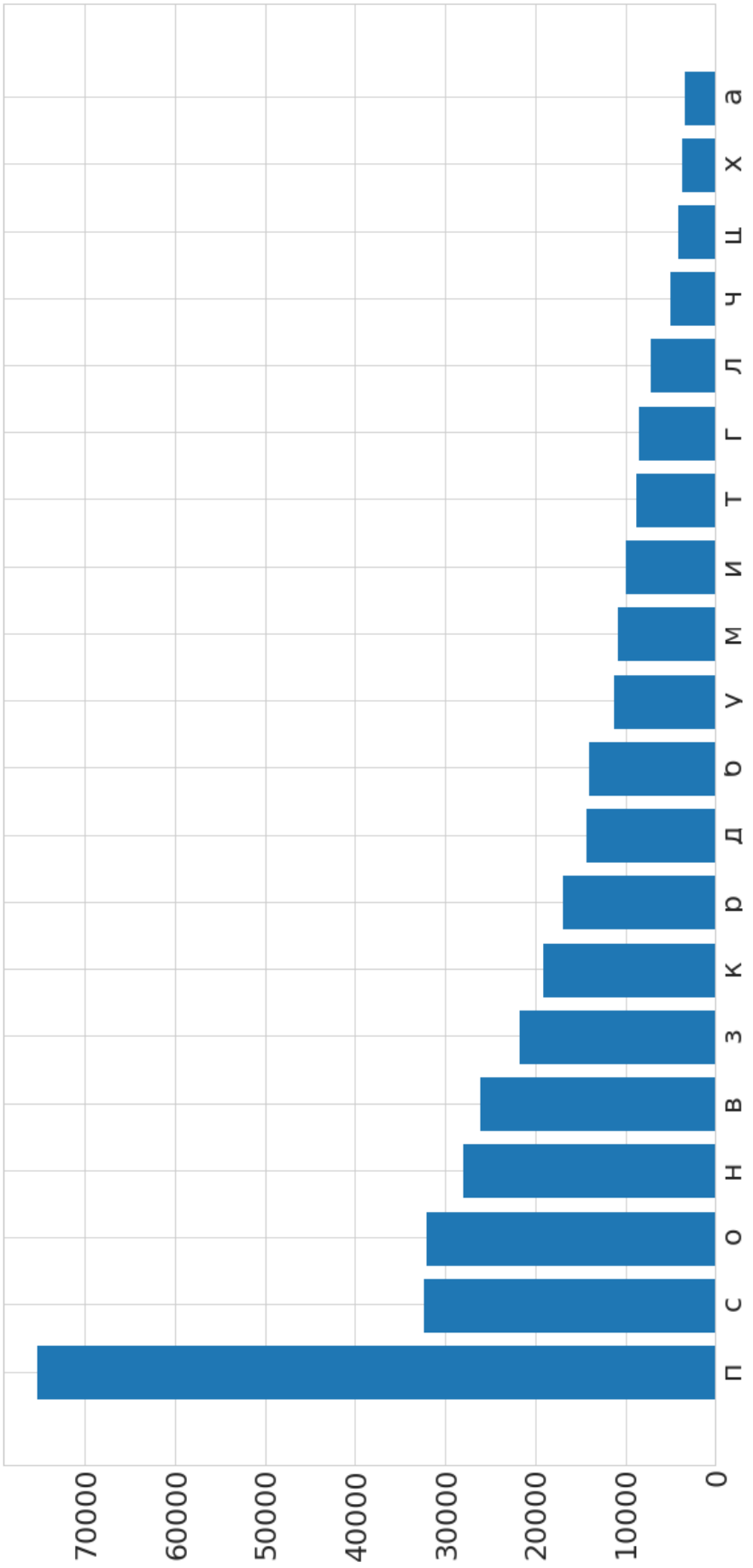

Рисунок 4.5 — Распределение первых 20ти букв слов в словаре по сегментам

Заключение

В ходе выполнения данной лабораторной работы были изучены три алгоритмы поиска по словарю. Были описаны все алгоритмы и реализованы. Также были изучены способы организации слов в словарях:

1) неотсортированный формат;
2) отсортированный формат;
3) неотсортированными сегментами с отсортированными словами.

В ходе экспериментов по замеру времени работы по словарю было установлено, что в худших случаях алгоритму полного перебора требуется 42 секунды на поиск последего слова или же несуществующего, что очень долго. Среднее время бинарного поиска и поиска по сегментам примерно совпадает, но распределение первых букв в данном словаре позволило улучшить среднее время поиска приблизительно в два раза.

Список использованных источников

1. IDE Visual Studio 2019. Режим доступа: (дата обращения: 11.09.2020) Свободный. URL: https://visualstudio.microsoft.com/ru/vs/

2. Python. Режим доступа: (дата обращения: 01.10.2020) Свободный. URL: https://www.python.org/

3. Intel® Core™ i5-7200U Processor [Электронный ресурс]. Режим доступа: (дата обращения: 26.09.2020) Свободный. URL: https://www.intel.com/content/www/us/en/products/processors/core/i5-processors/i5-7200u.html

4. Binary search algorithm [Электронный ресурс]. Режим доступа: (дата обращения: 15.10.2020) Свободный. URL: en.wikipedia.org/wiki/Binary_search_algorithm