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 ИУ, Информатика и системы управления

КАФЕДРА ИУ7, Программное обеспечение ЭВМ и информационные технологии

ЛАБОРАТОРНАЯ РАБОТА №7

ПО ДИСЦИПЛИНЕ

“Анализ алгоритмов”

Студент ИУ7-54Б _____ **А.А. Андреев**
(Группа) (Подпись, дата) (И.О.Фамилия)

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

2021 г.

Содержание

Введение **3**

1. Аналитическая часть **4**

1.1 Алгоритм полного перебора 4

1.2 Алгоритм двоичного поиска 5

Вывод 6

2. Конструкторская часть. **7**

2.1 Структура словаря 7

2.2 Схема алгоритмов 7

Вывод 9

3. Технологическая часть. **10**

3.1 Требования к программному обеспечению 10

3.2 Средства реализации 10

3.3 Реализация алгоритмов 10

Вывод 14

4. Исследовательская часть. **15**

4.1 Демонстрация работы программы 15

4.2 Технические характеристики 15

4.3 Время выполнения алгоритмов 16

Вывод 17

Заключение **18**

Список литературы **19**

Введение

Данная лабораторная работа посвящена исследованию алгоритмов поиска в словаре.

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.

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 [1]. В данной лабораторной работе будут рассмотренны и реализованы такие алгоритмы поиска в словаре как:

1) полный перебор;

2) бинарный поиск;

3) поиск по сегментам.

Цель данной работы является: изучение и применение на практике алгоритмов поиска по словарю.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Описать и реализовать алгоритмы полного перебора, двоичного поиска и поиска по сегментам;
2. Провести сравнительный анализ работы алгоритмов поиска по словарю;
3. Сделать выводы и применимости алгоритмов к решению задачи поиска по словарю.

1. Аналитическая часть

В данном разделе будет описана теоретическая основа алгоритмов поиска по словарю.

1.1 Алгоритм полного перебора

Алгоритмом полного перебора называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты исходного набора данных [2]. 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 решения может стремиться к экспоненциальному времени работы.

Пусть алгоритм нашёл элемент на первом сравнении (лучший случай), тогда будет затра- чено $k0 + k1$ операций, на втором - $k0 + 2 \cdot k1$, на последнем (худший случай) - $k0 + N \cdot k1$. 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 послед- ней позиции. Средняя трудоёмкость может быть рассчитана как математическое ожидание по формуле (1.1), где Ω – множество всех возможных случаев.

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

1.2 Алгоритм двоичного поиска

Алгоритм двоичного поиска применяется к заранее упорядоченному словарю. При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в словаре. Если они равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой или правой частях словаря [3].

На каждом шаге осуществляется поиск середины отрезка по формуле (1.2).

$$mid = \frac{left + right}{2} \quad (1.2)$$

Если искомый элемент равен элементу с индексом mid, поиск завершается. В случае ес- ли искомый элемент меньше элемента с индексом mid, на место mid перемещается правая граница рассматриваемого отрезка, в противном случае — левая граница.

Поиск в словаре с использованием данного алгоритма в худшем случае (необходимо спуститься по двоичному дереву от корня до листа) будет иметь трудоемкость $O(log2N)$, что быстрее поиска при помощи алгоритма полного перебора. Но стоит учитывать тот факт, что данный алгоритм работает только для заранее упорядоченного словаря. В случае большого объема данных и обратного порядка сортировки может произойти так, что алгоритм полного перебора будет эффективнее по времени, чем алгоритм двоичного поиска.

1.3 Алгоритм частотного анализа

Алгоритм на вход получает словарь и на его основе составляется частотный анализ. По полученным значениям словарь разбивается на сегменты так, что все элементы с некоторым общим признаком попадают в один сегмент (для строк это может быть первая буква, для чисел - остаток от деления) [4].

Сегменты упорядочиваются по значению частотной характеристики так, чтобы к элемен- там с наибольшей частотной характеристикой был самый быстрый доступ. Такой характери- стикой может послужить, например, размер сегмента. Вероятность обращения к определен- ному сегменту равна сумме вероятностей обращений к его ключам, то есть $Pi = \sum j\ pj = N \cdot p$, где Pi - вероятность обращения к i-ому сегменту, pj - вероятность обращения к j-ому элемен- ту, 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 упорядочиваются по значению. Это необходимо для реа- лизации бинарного поиска, который обеспечит эффективный поиск со сложностью $O(\log 2\ n)$, где n - 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 (1.3).

$$\sum_{i \in \Omega} \left(f_{\text{выбор сегмента } i\text{—го элемента}} + f_{\text{бинарный поиск } i\text{—го элемента}} \right) \cdot p_i \tag{1.3}$$

Вывод

В аналитическом разделе была описана теоретическая основа алгоритмов поиска по словарю, изучены их преимуществами и недостатками.

2. Конструкторская часть.

В данном разделе будет приведены блок-схемы алгоритмов, описанных в аналитическом разделе.

2.1 Структура словаря

Словарь состоит из пар вида <number - type>, где number - номер договора.

2.2 Схема алгоритмов

На рисунке 2.1 приведена обобщенная схема полного перебора.

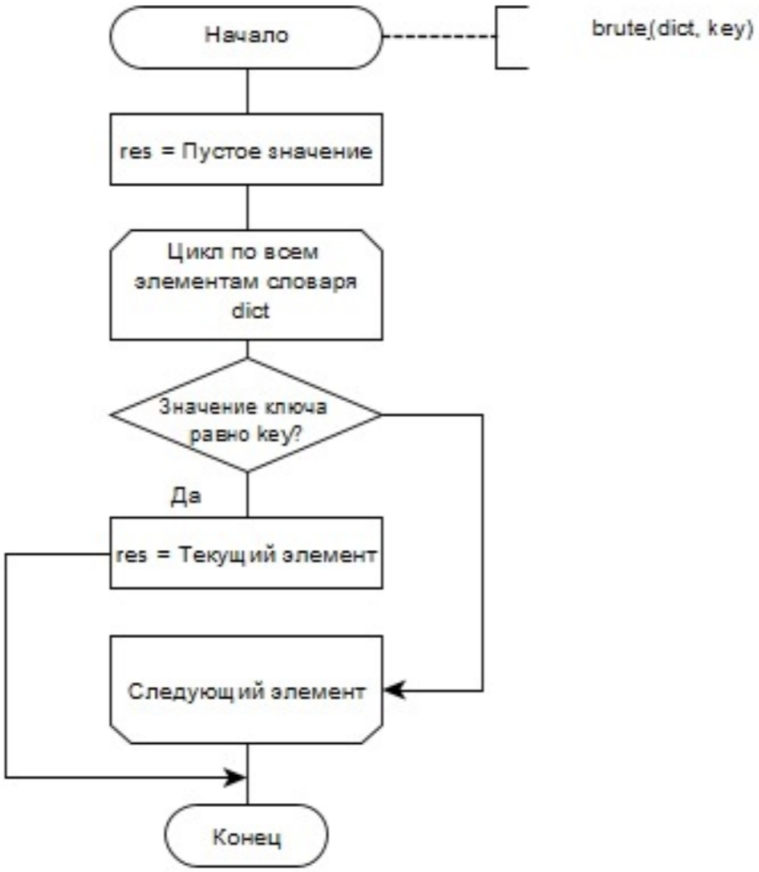

Рисунок 2.1: Схема алгоритма полного перебора

На рисунке 2.2 приведена обобщенная схема двоичного поиска.

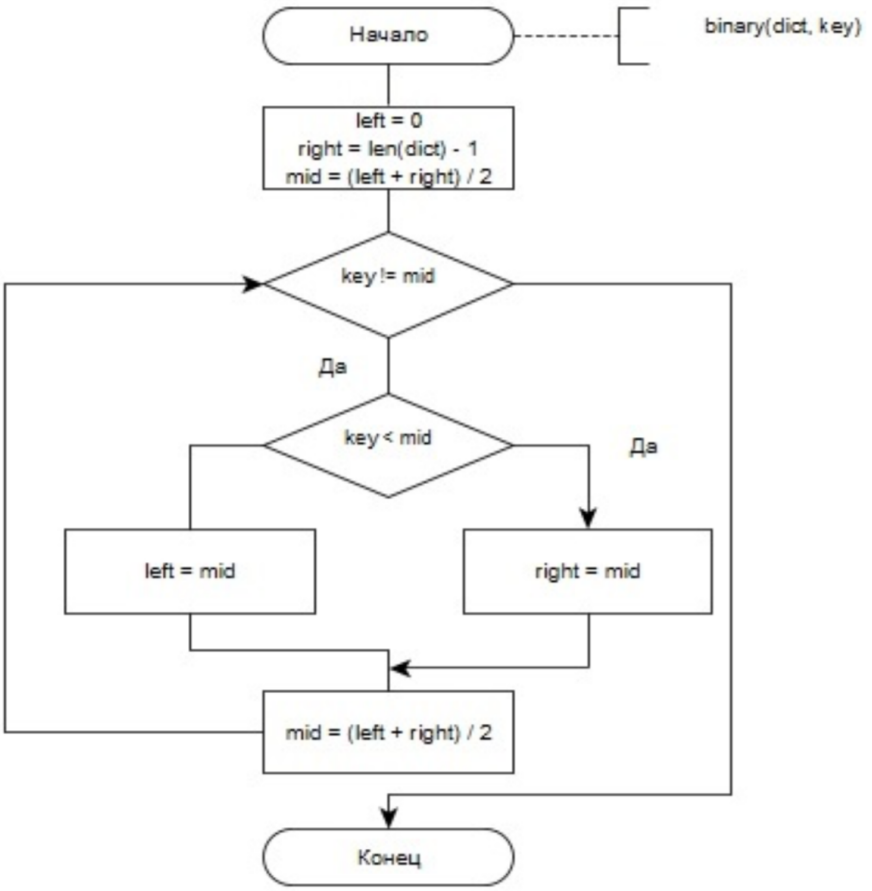

Рисунок 2.2: Схема алгоритма двоичного поиска

На рисунке 2.3 приведена обобщенная схема частотного анализа.

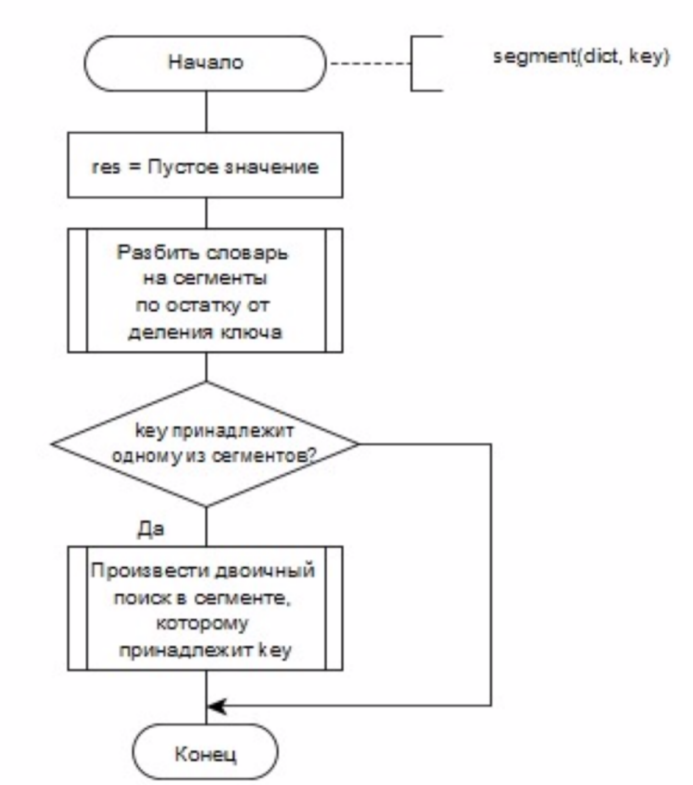

Рисунок 2.3: Схема алгоритма с использованием частотного анализа

Вывод

В данном разделе были рассмотрены блок-схемы, которые позволяют перейти к технологической части.

3. Технологическая часть.

В данном разделе будут рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач.

3.1 Требования к программному обеспечению

Программное обеспечение должно реализовывать поставленную на лабораторную работу задачу. Интерфейс для взаимодействия с программой - командная строка. Пользователь должен иметь возможность вводить номер контракта для поиска в словаре. Словарь должен генерироваться перед

3.2 Средства реализации

Для разработки данной программы применён язык Python [2] с библиотекой time() для вычисления времени работы процессора, чтобы расширить знания в области данного языка программирования.

3.3 Реализация алгоритмов

В листингах 3.1-3.8 приведена реализация работы программы поиска.

Листинг 3.1: Базовый класс Search

```
1. class Search:
2.
3.     _dictionary, _number, _record = None, None, None
4.
5.     def __init__(self, dictionary, number):
6.         self._dictionary = dictionary
7.         self._number = number
8.
9.     @property
10.        def dict(self):
11.            return self._dictionary
12.
13.        @property
14.        def number(self):
15.            return self._number
16.
17.        @property
18.        def record(self):
19.            return self._record
```

Листинг 3.2: Класс BruteSearch

```
1. class BruteSearch(Search):
2.
3.     def __init__(self, dictionary, number):
4.         super().__init__(dictionary, number)
5.         self.find()
6.
7.     def find(self):
8.         for record in self.dict:
9.             if record['number'] == self.number:
10.                 self._record = record
11.                 return config.BS
12.         self._record = None
13.         return config.BS
```

Листинг 3.3: Класс BinarySearch

```
1. class BinarySearch(Search):
2.
3.     def __init__(self, dictionary, number):
4.         super().__init__(dictionary, number)
5.         self.find()
6.
7.     def find(self):
8.         left, right = 0, len(self.dict) - 1
9.
10.         if self.dict[left]['number'] > self.number or
    self.dict[right]['number'] < self.number:
11.             self._record = None
12.             return config.BS
13.
14.         if self.dict[left]['number'] == self.number:
15.             self._record = self.dict[left]
16.             return config.BS
17.
18.         if self.dict[right]['number'] == self.number:
19.             self._record = self.dict[right]
20.             return config.BS
21.
22.         mid = (left + right) // 2
23.         res = self.dict[mid]['number']
24.         while self.number != res:
25.             if self.number < res:
26.                 right = mid
27.             elif self.number > res:
28.                 left = mid
29.             mid = (left + right) // 2
30.             res = self.dict[mid]['number']
31.         self._record = self.dict[mid]
32.         return config.BS
```

Листинг 3.4: Класс SegmentSearch

```
1. class SegmentSearch(Search):
2.
3.     _segment_list = None
4.
5.     def __init__(self, dictionary, number, segment_list):
6.         super().__init__(dictionary, number)
7.         self._segment_list = segment_list
8.         self.find()
9.
10.        def find(self):
11.            if len(self._segment_list) == 0:
12.                self._record = None
13.                return config.BS
14.
15.            self._record =
    BinarySearch(self._segment_list[self.number %
    len(self._segment_list)], self.number).record
16.            return config.BS
```

Листинг 3.5: Основные функции

```
1. def divide_dict(dictionary, segment_count):
2.     segment_list = [[] for _ in range(segment_count)]
3.     for record in dictionary:
4.         segment_list[record['number'] %
    segment_count].append(record)
5.     return segment_list
6.
7.
8. def get_time(iteration, dictionary, s_type, segment_count):
9.     segment_list = divide_dict(dictionary, segment_count)
10.
11.        time = 0
12.        for i in range(iteration):
13.            t1 = process_time()
14.            for j in range(1000):
15.                if s_type == config.BR_TYPE:
16.                    BruteSearch(dictionary, j + 1)
17.                elif s_type == config.BI_TYPE:
18.                    BinarySearch(dictionary, j + 1)
19.                elif s_type == config.SE_TYPE:
20.                    SegmentSearch(dictionary, j + 1,
    segment_list)
21.            t2 = process_time()
22.            time += (t2 - t1) / 1000
23.
24.        return time / iteration
```

Листинг 3.6: Основной файл программы

```
import i_o

import config
import search, dictionary

def main(iteration=100, segment_count=2,
type_contract=config.type_contract):
    data = dictionary.Dictionary(iteration, type_contract,
show_generation=True)

    number = int(input("Введите номер договора: "))

        br_word, bi_word, se_word =
search.BinarySearch(data.data, number).record,
search.BinarySearch(data.data, number).record, \

search.SegmentSearch(data.data, number,
search.divide_dict(data.data, segment_count)).record

        br_time, bi_time, se_time = search.get_time(iteration,
data.data, config.BR_TYPE, segment_count), \
                                    search.get_time(iteration,
data.data, config.BI_TYPE, segment_count), \
                                    search.get_time(iteration,
data.data, config.SE_TYPE, segment_count)

        i_o.IO(
            br_word, bi_word, se_word,
            br_time, bi_time, se_time
        ).out_searching_results()


if __name__ == '__main__':
    main()
```

Вывод

В данном разделе были рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач.

4. Исследовательская часть.

4.1 Демонстрация работы программы

Пример работы программы представлен на рисунке 4.1.

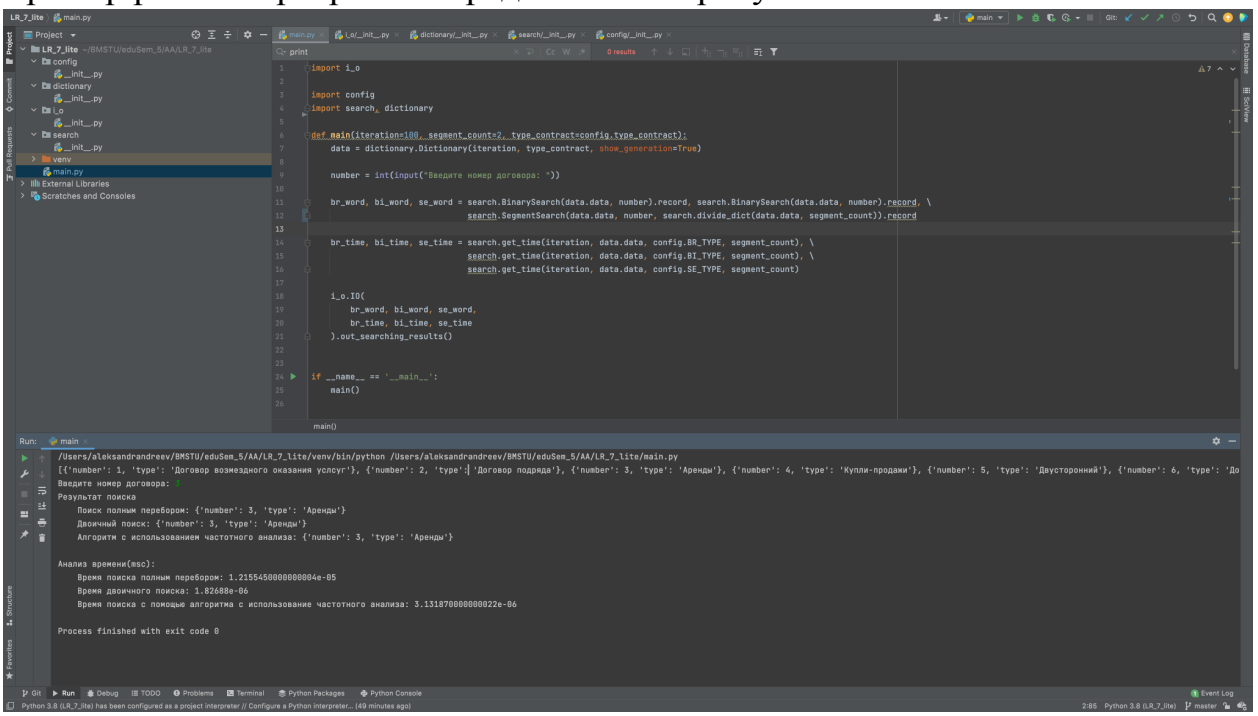

Рисунок 4.1: Демонстрация работы программы

4.2 Технические характеристики

В Таблице 4.2 приведены технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения.

Таблица 4.2: Технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения

ОС	Mac OS Mojave 64-bit
ОЗУ	8 Gb 2133 MHz LPDDR3
Процессор	2,3 GHz Intel Core i5

4.3 Время выполнения алгоритмов

Для произведения замеров времени выполнения реализации алгоритмов будет использована формула:

$$t = NT \tag{4.3}$$

где t — среднее время выполнения алгоритма, N — количество замеров, T — время выполнения N замеров. Неоднократное измерение времени необходимо для получения более точного результата. Количество замеров взято равным 100. Для сравнения времени работы алгоритмов использовался словарь, состоящий из 1751 элемента. На рисунке 4.3 представлены графики времени работы алгоритмов поиска по словарю. Индекс ключа указан на горизонтальной оси.

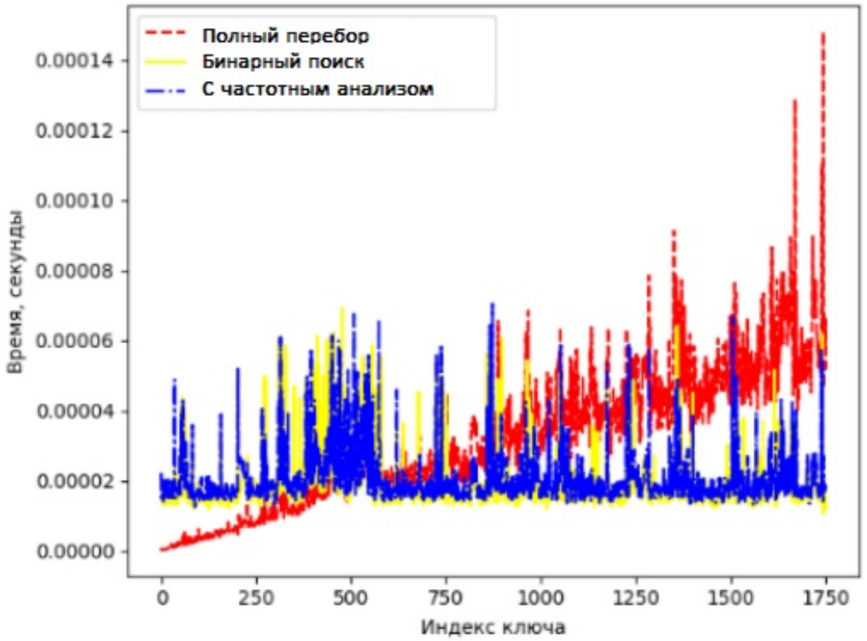

Рисунок 4.3: Время выполнения алгоритмов

Как можно наблюдать на графике, самый медленный алгоритм - алгоритм полного перебора. Время на в нём растет линейно и увеличивается с увеличением индекса элемента словаря. Алгоритм бинарного поиска и алгоритм с использованием частотного анализа тратят примерно одинаковое количество времени, однако стоит учитывать, что они требуют дополнительных расходов времени на подготовку данных к работе с алгоритмом. Однако в лучшем случае алгоритм полного перебора работает в 4 раза быстрее алгоритма двоичного поиска и алгоритма с использованием частотного анализа.

Вывод

В данном разделе был проведен сравнительный анализ работы реализованных алгоритмов полного поиска полным перебором, бинарным поиском и бинарным поиском методом частотного анализа. Самый медленный алгоритм - алгоритм полного перебора. Время на в нём растет линейно и увеличивается с увеличением индекса элемента словаря. Алгоритм бинарного поиска и алгоритм с использованием частотного анализа тратят примерно одинаковое количество времени, однако стоит учитывать, что они требуют дополнительных расходов времени на подготовку данных к работе с алгоритмом. Однако в лучшем случае алгоритм полного перебора работает в 4 раза быстрее алгоритма двоичного поиска и алгоритма с использованием частотного анализа.

Заключение

В ходе выполнения данной лабораторной работы были изучены принципы работы реализованных алгоритмов полного поиска полным перебором, бинарным поиском и бинарным поиском методом частотного анализа. Было проведено исследование работы алгоритма при различных параметрах, из которого можно сделать вывод, самый медленный алгоритм - алгоритм полного перебора. Время на в нём растет линейно и увеличивается с увеличением индекса элемента словаря. Алгоритм бинарного поиска и алгоритм с использованием частотного анализа тратят примерно одинаковое количество времени, однако стоит учитывать, что они требуют дополнительных расходов времени на подготовку данных к работе с алгоритмом. Однако в лучшем случае алгоритм полного перебора работает в 4 раза быстрее алгоритма двоичного поиска и алгоритма с использованием частотного анализа.

Список литературы

[1] Кормен Т.Х., Лейзерсон Ч.И., Алгоритмы: Построение и анализ, год выпуска 2019, тираж 1328, 700 страниц.

[2] ISO/IEC JTC1 SC22 WG21 N 3690 «Programming Languages — C++» [Электронный ресурс]. https://devdocs.io/cpp/ Дата обращения: 13.09.2021

[3] Гасфилд, Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, год выпуска 2003, типаж 900, 653 страницы.

[4] Вычисление процессорного времени выполенения программы [Электронный ресурс] Режим доступа: https://www.tutorialspoint.com/python/time_clock.htm. Дата обращения: 13.09.2021