



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 Алгоритмы поиска в словаре

Студент Зайцева А.А.

Группа ИУ7-52Б

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Алгоритм полного перебора	4
1.2 Алгоритм бинарного поиска	4
1.3 Алгоритм поиска в сегментированном словаре	5
1.4 Вывод из аналитической части	6
2 Конструкторская часть	7
2.1 Структура импользуемого словаря	7
2.2 Схема алгоритма полного перебора	7
2.3 Схема алгоритма бинарного поиска	9
2.4 Схема алгоритма поиска в сегментированном словаре	10
2.5 Вывод из конструкторской части	13
3 Технологическая часть	14
3.1 Требования к ПО	14
3.2 Выбор средств реализации	14
3.3 Листинги кода	14
3.4 Тестирование	17
4 Исследовательская часть	18
4.1 Технические характеристики	18
4.2 Пример работы программы	18
4.3 Сравнение трудоемкостей реализаций	19
4.4 Сравнение времени выполнения реализаций алгоритмов	20
4.5 Количество сравнений	22
4.6 Вывод из исследовательской части	29
Заключение	30
Список использованной литературы	31

Введение

Словарь (англ. dictionary, map) — абстрактный тип данных, позволяющий хранить набор значений, обращение к которым происходит по ключам. Ключи должны допускать сравнение друг с другом. Примеры словарей достаточно разнообразны. Например, обычный толковый словарь хранит определения слов (являющиеся значениями), сопоставленные с самими словами (являющимися ключами), а банковская база данных может хранить данные клиентов, сопоставленные с номерами счетов.

Одной из основных операций в словаре является поиск значения по ключу. Словари могут содержать большое количество элементов, поэтому вопрос скорости поиска в них является важным [1].

Целью данной работы является разработка эффективного алгоритма поиска по словаре.

В рамках выполнения работы необходимо решить следующие задачи:

- 1) изучить три алгоритма поиска в словаре: алгоритм полного перебора, алгоритм бинарного поиска и алгоритм поиска в сегментированном словаре;
- 2) разработать и реализовать изученные алгоритмы;
- 3) провести сравнительный анализ трудоёмкости реализаций алгоритмов на основе теоретических расчетов (в среднем; в лучшем и худшем случаях);
- 4) провести сравнительный анализ процессорного времени выполнения реализаций алгоритмов на основе экспериментальных данных;
- 5) провести сравнительный анализ количества сравнений с заданным ключом, необходимых для поиска каждым алгоритмом значения по ключу и для определения отсутствия заданного ключа в словаре.

1 Аналитическая часть

В данном разделе будет приведена теория, необходимая для разработки и реализации трех алгоритмов поиска в словаре: алгоритма полного перебора, алгоритма бинарного поиска и алгоритма поиска в сегментированном словаре.

1.1 Алгоритм полного перебора

Алгоритм полного перебора сводится к последовательному прохождению по всем ключам словаря и их сравнению с заданным ключом.

Этот алгоритм считается методом «грубой силы» [2]. Зато он может производиться в неотсортированном словаре, а добавление новых элементов в такой словарь не вызывает затруднений – их можно добавлять на любую позицию. Дополнительных затрат по памяти также не требуется.

1.2 Алгоритм бинарного поиска

Двоичный поиск (бинарный поиск) [3] — алгоритм поиска объекта (в нашем случае - значения) по заданному признаку (ключу) в множестве объектов, упорядоченных по тому же самому признаку (по ключу), принцип работы которого заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где находится искомый объект.

Данный алгоритм работает за логарифмическое время, дополнительных затрат по памяти не требует. Однако алгоритм бинарного поиска подразумевает, что массив ключей отсортирован. Это вызывает трудности при добавлении новых элементов в словарь, особенно если словарь хранится в постоянной памяти (например, на диске).

Алгоритм вставки элемента в упорядоченный массив заключается в последовательном анализе элементов массива, начиная с последнего и, при необходимости, сдвиге анализируемого элемента вправо на одну позицию, для того, чтобы освободить место для вставляемого элемента. Сдвиги про-

водятся до тех пор, пока не будет найдено место для вставляемого элемента, соответствующее его значению [4].

1.3 Алгоритм поиска в сегментированном словаре

Словарь можно разбить на сегменты так, чтобы все элементы с некоторым общим признаком попадали в один сегмент. Признаком для строк может быть первая буква, для чисел – остаток от деления. Элементы внутри одного сегмента можно также разбивать на подсегменты и так далее.

Затем сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементам с наибольшей частотной характеристикой был самый быстрый доступ (в случае поиска полным перебором они должны быть ближе к началу). Такой характеристикой может послужить, например, размер сегмента.

Вероятность обращения к определенному сегменту равна сумме вероятностей обращений к его ключам, то есть $P_i = \sum_j p_j$, 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: $P_i = N \cdot p$, где N - количество элементов в i -ом сегменте, а p - вероятность обращения к произвольному ключу.

Таким образом, для поиска ключа в словаре необходимо найти сегмент, которому принадлежит этот ключ (например, полным перебором), а затем осуществить поиск ключа в пределах найденного сегмента. Чтобы поиск в пределах сегмента осуществлялся эффективнее, можно упорядочить ключи в каждом сегменте и реализовать бинарный поиск.

Сегментированный словарь позволяет уменьшить количество сравнений, необходимых для поиска в словаре. Однако он требует предобработки произвольного словаря для превращения его в сегментированный, а также дополнительной памяти под такую организацию. Объем дополнительной памяти определяется количеством выделяемых сегментов и подсегментов.

Также из-за необходимости поддержания структуры словаря усложня-

ется операция добавления в словарь новых элементов. Необходимо определить, к какому сегменту относится новый ключ, если сегмент разбит на подсегменты – соответствующий подсегмент, а затем найти позицию внутри сегмента, в которую этот ключ необходимо вставить для сохранения упорядоченности. Также после добавления нового ключа сегмент становится больше, поэтому необходимо проверить упорядоченность сегментов по частотной характеристике и пересортировать их в случае необходимости.

1.4 Вывод из аналитической части

Были рассмотрены идеи и материалы, необходимые для разработки и реализации трех алгоритмов поиска в словаре: алгоритма полного перебора, алгоритма бинарного поиска и алгоритма поиска в сегментированном словаре.

2 Конструкторская часть

В данном разделе будут приведены структура используемого словаря и схемы рассмотренных в предыдущем разделе алгоритмов поиска в словаре.

2.1 Структура импользуемого словаря

Словарь состоит из пар вида $\langle \text{en} - \text{rus} \rangle$, где en - слово на английском языке, rus - его перевод на русский.

2.2 Схема алгоритма полного перебора

На рисунке 2.1 приведена схема алгоритма полного перебора поиска в словаре.

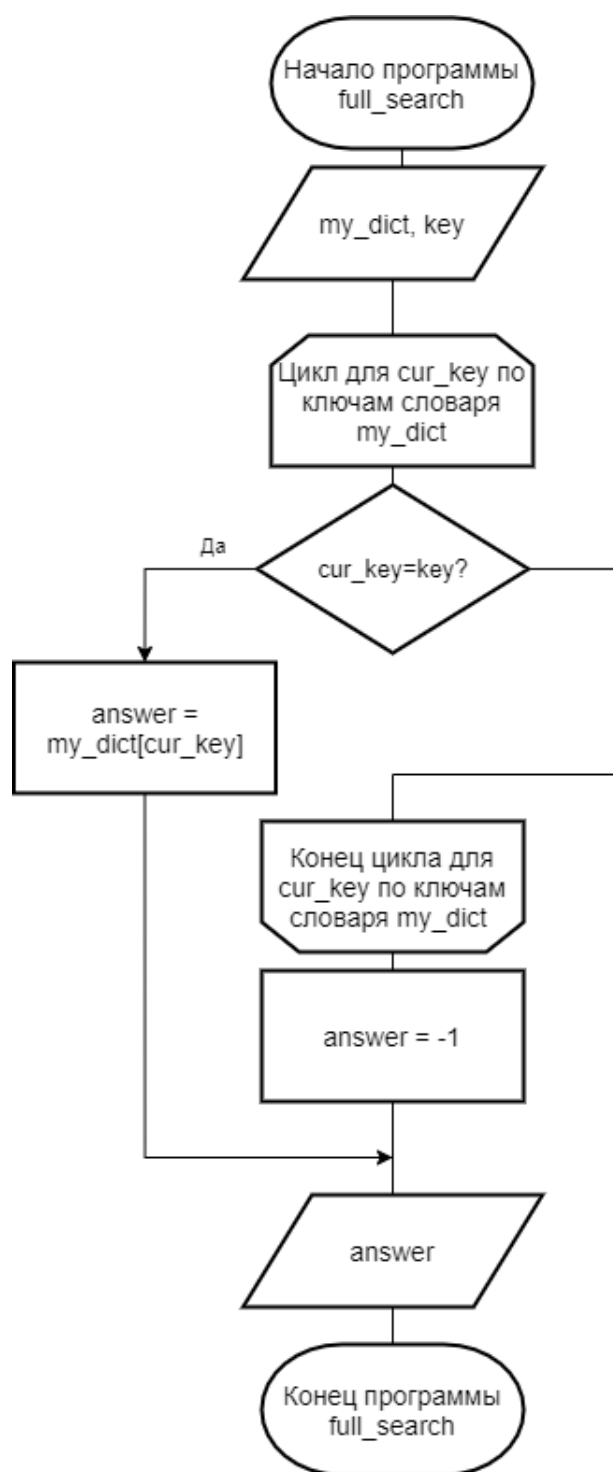


Рисунок 2.1 – Схема алгоритма полного перебора

2.3 Схема алгоритма бинарного поиска

На рисунке 2.2 приведена схема алгоритма бинарного поиска в словаре. Предполагается, что передаваемый словарь `my_dict` лексикографически упорядочен по возрастанию ключей.

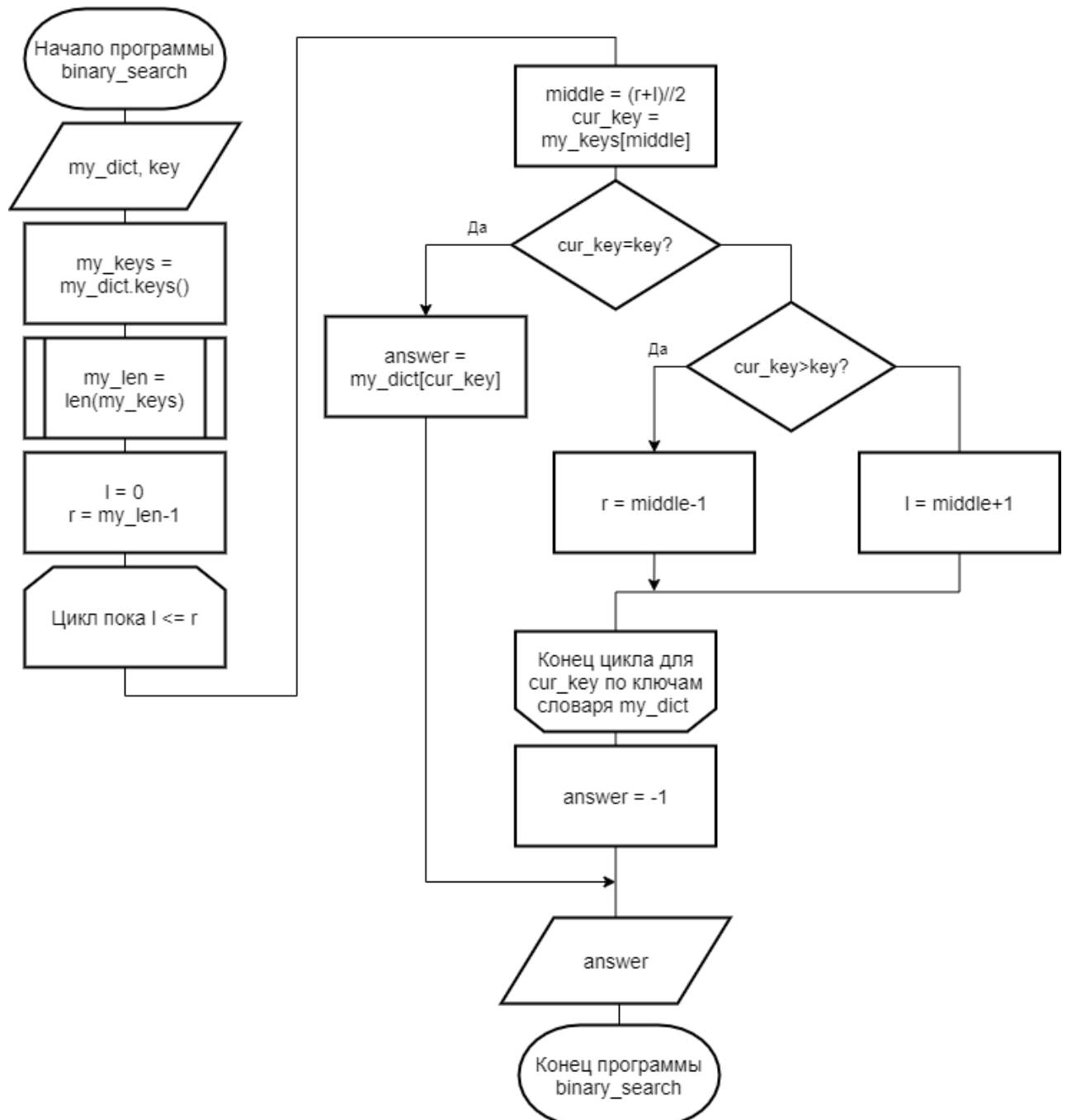


Рисунок 2.2 – Схема алгоритма бинарного поиска

2.4 Схема алгоритма поиска в сегментированном словаре

На рисунке 2.3 приведена схема создания словаря, сегментированного по первым буквам ключей. Такой подход к сегментации выбран, так как он является наиболее нативным для используемого словаря. Предполагается, что обращения ко всем ключам равновероятны, поэтому для ускорения поиска нужного сегмента полным перебором сегментированный словарь упорядочен по убыванию количества пар (ключ: значени) в нем. Пары внутри одого сегмента упорядочены лексикографически по возрастанию ключей для возможности реализации бинарного поиска в пределах сегмента.

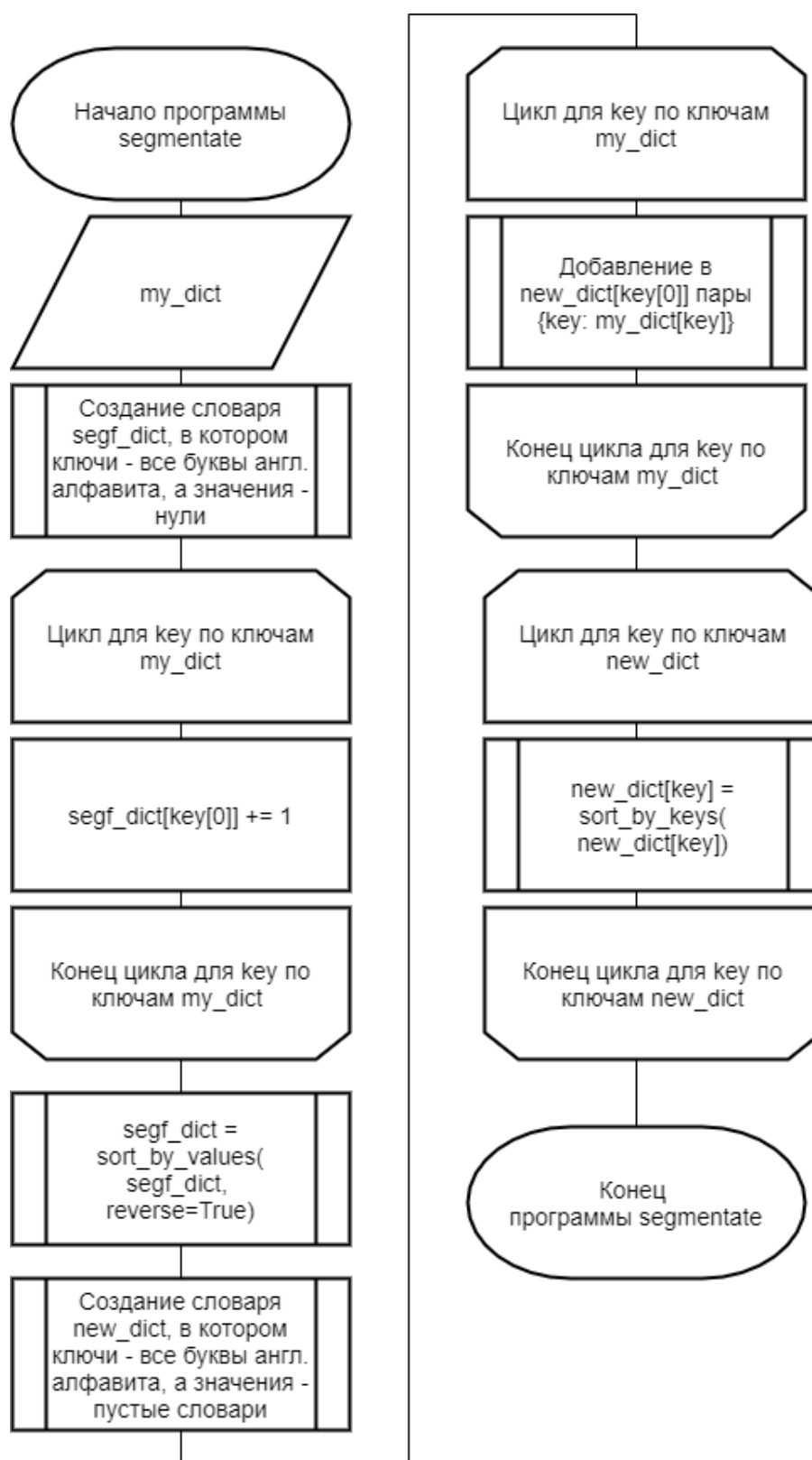


Рисунок 2.3 – Схема создания сегментированного словаря

На рисунке 2.4 приведена схема поиска в созданном сегментированном словаре. Сначала полным перебором осуществляется поиск нужного сегмента, а затем – бинарный поиск ключа в выбранном сегменте.

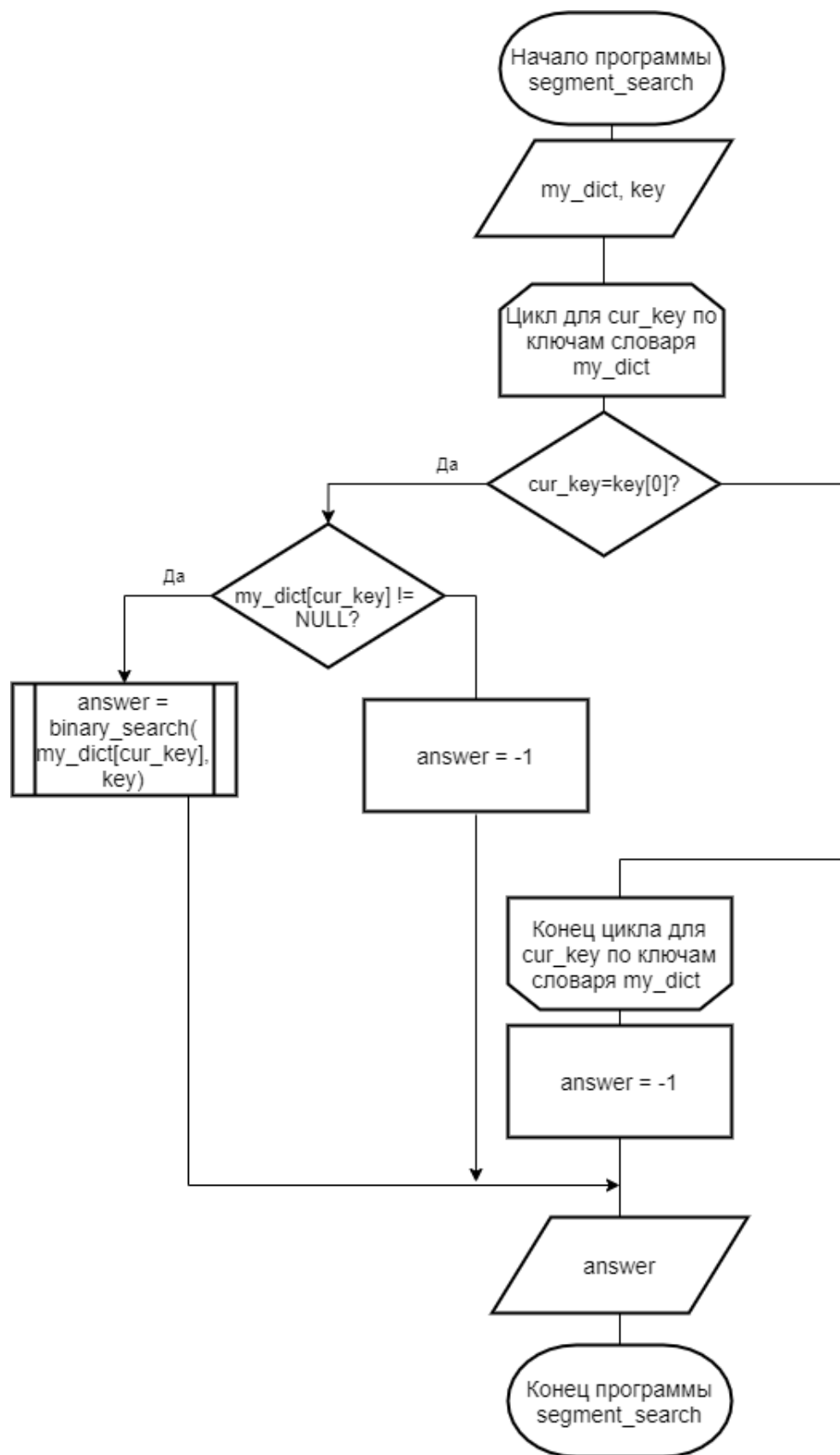


Рисунок 2.4 – Схема поиска в сегментированном словаре

2.5 Вывод из конструкторской части

Были приведены схемы разрабатываемых алгоритмов поиска в словаре: алгоритма полного перебора, алгоритма бинарного поиска и алгоритма поиска в сегментированном слолваре.

3 Технологическая часть

В данном разделе производится выбор средств реализации, приводятся требования к ПО, листинги реализованных алгоритмов поиска в словаре, а также результаты их тестирования.

3.1 Требования к ПО

На вход программе подается ключ, на выходе ожидается значение, найденное в словаре по заданному ключу каждым реализованным алгоритмом, или значение -1, если такого ключа в словаре нет.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Python [5]. Он позволяет быстро реализовывать различные алгоритмы без выделения большого времени на проектирование структуры программы и выбор типов данных.

Кроме того, в Python есть библиотека time, которая предоставляет функцию process_time для замера процессорного времени [6].

В качестве среды разработки выбран PyCharm. Он является кросс-платформенным, а также предоставляет удобный и функциональный отладчик и средства для рефакторинга кода, что позволяет быстро находить и исправлять ошибки [7].

3.3 Листинги кода

В листинге 3.1 представлена реализация алгоритма полного перебора поиска в словаре.

Листинг 3.1 – Алгоритм полного перебора

```
1 def full_search(my_dict, key):  
2     for cur_key in my_dict.keys():  
3         if cur_key == key:  
4             return my_dict[cur_key]  
5     return -1
```

В листинге 3.2 представлена реализация алгоритма бинарного поиска в словаре.

Листинг 3.2 – Алгоритм бинарного поиска в словаре

```
1 def binary_search(my_dict, key):  
2     my_keys = list(my_dict.keys())  
3     my_len = len(my_keys)  
4     l = 0  
5     r = my_len - 1  
6  
7     while l <= r:  
8         middle = (r + l) // 2  
9         cur_key = my_keys[middle]  
10  
11        if cur_key == key:  
12            return my_dict[cur_key]  
13        elif cur_key > key:  
14            r = middle - 1  
15        else:  
16            l = middle + 1  
17  
18    return -1
```

В листинге 3.3 представлена реализация функции для создания сегментированного словаря.

Листинг 3.3 – Функция для создания сегментированного словаря

```
1
2 def segmentate(my_dict):
3     seg_frequency_dict = {letter: 0 for letter in "
4         abcdefghijklmnopqrstuvwxyz"}
5     for key in my_dict.keys():
6         seg_frequency_dict[key[0]] += 1
7
8     seg_frequency_dict = sort_by_values(seg_frequency_dict, reverse=
9         True)
10
11     new_dict = {letter: dict() for letter in seg_frequency_dict.keys
12         ()}
13     for key in my_dict.keys():
14         new_dict[key[0]].update({key: my_dict[key]})
15     for key in new_dict:
16         new_dict[key] = sort_by_keys(new_dict[key])
17
18     return new_dict
```

В листинге 3.4 представлена реализация алгоритма поиска в сегментированном словаре.

Листинг 3.4 – Алгоритм поиска в сегментированном словаре

```
1 def segment_search(my_dict, key):
2     for k in my_dict:
3         if key[0] == k:
4             if my_dict[k]:
5                 return binary_search(my_dict[k], key)
6             else:
7                 return -1
8     return -1
```


3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов поиска в словаре. Все тесты пройдены успешно этим каждым алгоритмом.

Ключ	Словарь	Ожидание	Результат
"few"	{}	-1	-1
"few"	{"few": "неск.", "little": "немн."}	"неск."	"неск."
"much"	{"few": "неск.", "little": "немн."}	-1	-1

Таблица 3.1 – Функциональные тесты.

Вывод

Был произведен выбор средств реализации, приведены требования к ПО, реализованы и протестированы алгоритмы поиска в словаре.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows 10;
- оперативная память: 16 Гб;
- процессор: Intel® Core™ i5-8259U;
- количество ядер: 4;
- количество логических процессоров: 8.

Во время тестирования ноутбук был включен в сеть питания и нагружен только встроенными приложениями окружения и системой тестирования.

4.2 Пример работы программы

На рисунке 4.1 приведен пример работы программы.

```

Loaded 2 items
Словарь:
{'few': 'несколько', 'little': 'немного'}
Input key: few
Python search: несколько
Full search: несколько
Binary search: несколько
Segment search: несколько

Process finished with exit code 0

```

Рисунок 4.1 – Пример работы программы

4.3 Сравнение трудоемкостей реализаций

полный перебор Трудоемкость алгоритма зависит от того, присутствует ли искомый ключ в словаре. Если такого ключа нет, то она определяется количеством ключей в словаре, если есть – его удаленностью от начала массива ключей.

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 трудоемкость такого случая равно трудоемкости случая с ключом на последней позиции. Средняя трудоемкость может быть рассчитана как математическое ожидание по формуле (4.1), где Ω – множество всех возможных случаев.

$$\begin{aligned}
\sum_{i \in \Omega} p_i \cdot f_i &= (k_0 + k_1) \cdot \frac{1}{N+1} + (k_0 + 2 \cdot k_1) \cdot \frac{1}{N+1} + \\
&+ (k_0 + 3 \cdot k_1) \cdot \frac{1}{N+1} + (k_0 + N k_1) \frac{1}{N+1} + (k_0 + N \cdot k_1) \cdot \frac{1}{N+1} = \\
&= k_0 \frac{N+1}{N+1} + k_1 + \frac{1+2+\dots+N+N}{N+1} = \\
&= k_0 + k_1 \cdot \left(\frac{N}{N+1} + \frac{N}{2} \right) = k_0 + k_1 \cdot \left(1 + \frac{N}{2} - \frac{1}{N+1} \right)
\end{aligned} \tag{4.1}$$

Средняя трудоёмкость при множестве всех возможных случаев Ω может быть рассчитана по формуле (4.2).

$$\sum_{i \in \Omega} (f_{\text{выбор сегмента } i\text{-ого элемента}} + f_{\text{бинарный поиск } i\text{-ого элемента}}) \cdot p_i \quad (4.2)$$

Возможны два случая определения отсутствия ключа в словаре. Если в словаре нет ключей, начинающихся на ту же букву, то поиск значения по ключу закончится на этапе поиска нужного сегмента. Иначе понадобится осуществить еще и поиск внутри сегмента.

4.4 Сравнение времени выполнения реализаций алгоритмов

Сравнивалось процессорное время работы реализаций алгоритмов поиска в словаре по ключу: алгоритма полного перебора, алгоритма бинарного поиска и алгоритма поиска в сегментированном словаре. Эти реализации сравнивались по времени работы при количестве ключей в словаре 100, 1000, 5000, 10000 и 40000.

Для каждого алгоритма и каждого количества заявок проводилось 2 замера времени. Первый был связан с поиском ключей, которые на самом деле есть в словаре, причем осуществлялся поиск каждого ключа, что позволило симитировать равновозможность выбора ключа. Второй замер связан с поиском ключей, которых на самом деле нет в словаре. Таких «несуществующих» ключей генерировалось столько, сколько всего ключей было в словаре.

Так как некоторые задачи выполняются достаточно быстро, а замеры времени имеют некоторую погрешность, они для каждой реализации, каждого объема словаря и каждого ключа выполнялись 10 раз, а затем вычислялось среднее время работы.

На рисунке 4.2 приведены результаты сравнения времени выполнения реализаций алгоритмов. На графике введены следующие обозначения: full_x - алгоритм полного перебора, binary_x - алгоритм бинарного поиска, segment_x

- алгоритм поиска в сегментированном словаре, $x=e$ (exists) - время работы соответствующего алгоритма при поиске существующих ключей, $x=n$ (not exist) – при поиске несуществующих ключей.

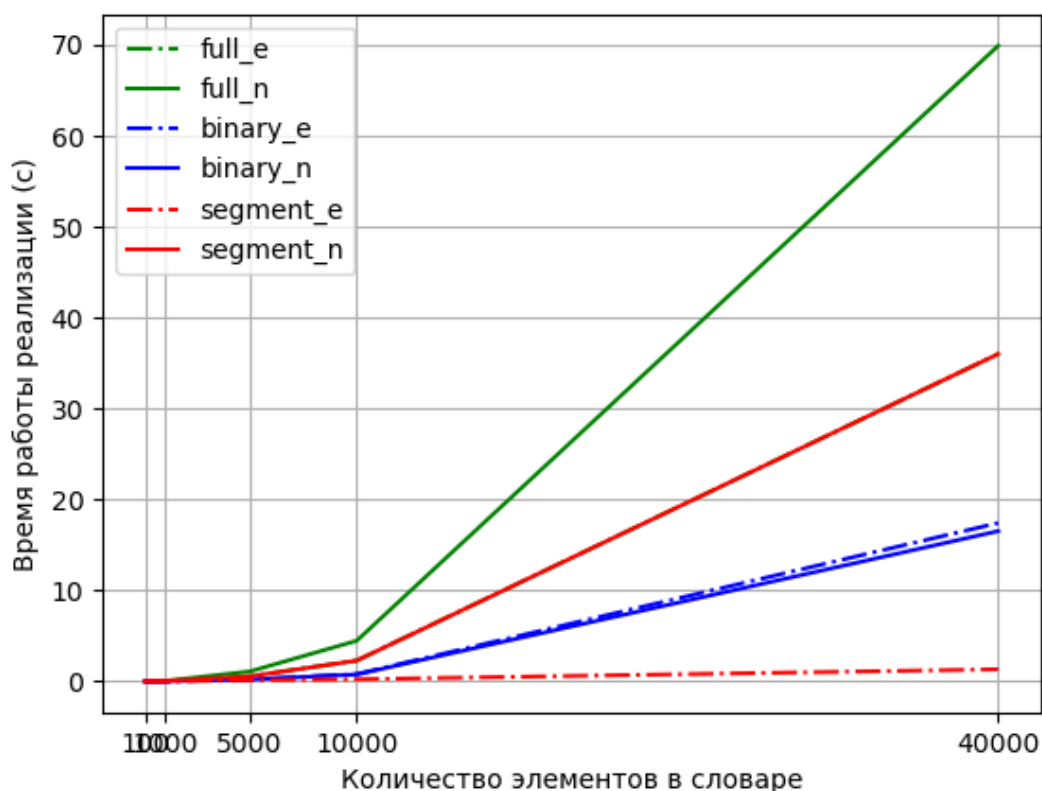


Рисунок 4.2 – Сравнение времени работы реализаций в зависимости от объема словаря

Как видно из графиков, для поиска существующего ключа наибольшее количество процессорного времени требуется алгоритму полного перебора, наименьшее - поиску в сегментированном словаре. Для определения того, что в словаре нет заданного ключа, наибольшее количество процессорного времени требуется опять же алгоритму полного перебора, наименьшее - бинарному поиску. Это различие в «наилучшем» алгоритме появляется в результате того, что при поиске несуществующего ключа в сегментированном словаре на бинарный поиск, являющийся его частью, накладывается еще и поиск нужного сегмента.

С увеличением размеров словаря разница во времени работы алгоритмов увеличивается.

4.5 Количество сравнений

На рисунках 4.3-4.8 приведены гистограммы с количеством сравнений (ось y), которые потребовались рассматриваемым реализациям алгоритмов поиска по ключам в словаре для того, чтобы найти нужный ключ (ось x). Все замеры производились в случае словаря, состоящего из 31 элемента. Для каждой реализации приведены две гистограммы, построенные по одним и тем же данным, но в первой ключи на оси x расположены в том порядке, в котором они хранились в словаре, а во второй – упорядочены по убыванию потребовавшегося количества сравнений.

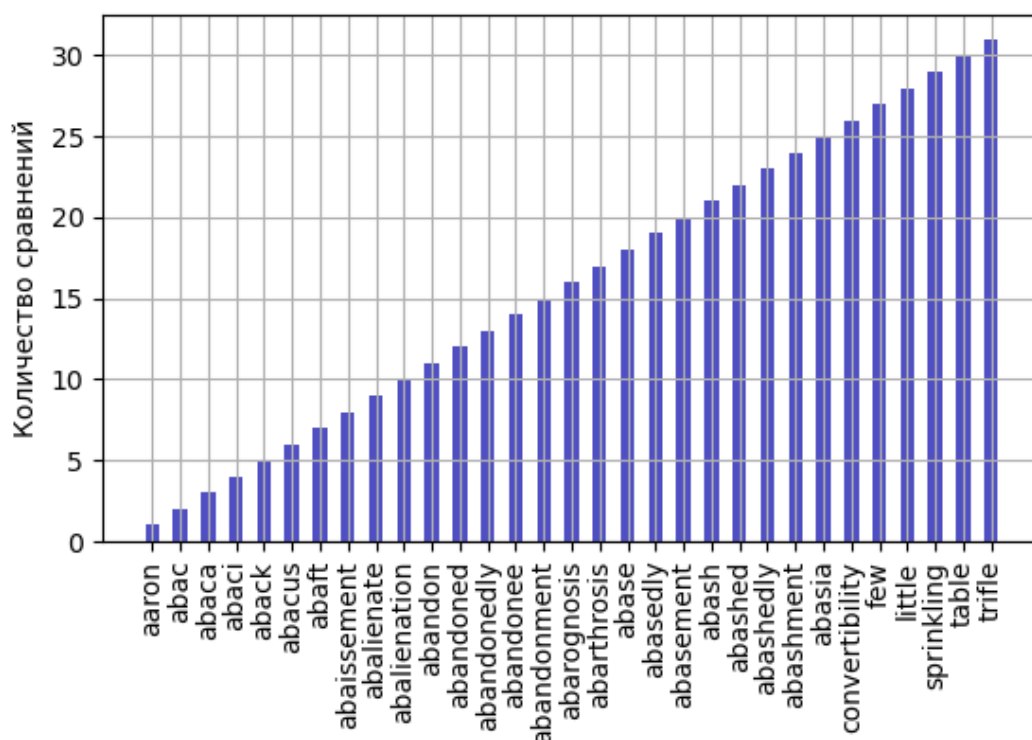


Рисунок 4.3 – Количество сравнений: полный перебор, сортировка по расположению

Из рисунков 4.3 и 4.4 видно, что количество сравнений, необходимых для поиска данного ключа полным перебором определяется только позицией ключа в словаре и равно этой позиции. Наименьшее количество сравнений (1) потребуется для самого первого ключа, наибольшее (N-количество элементов в словаре) – для последнего. Для определения того, что ключа

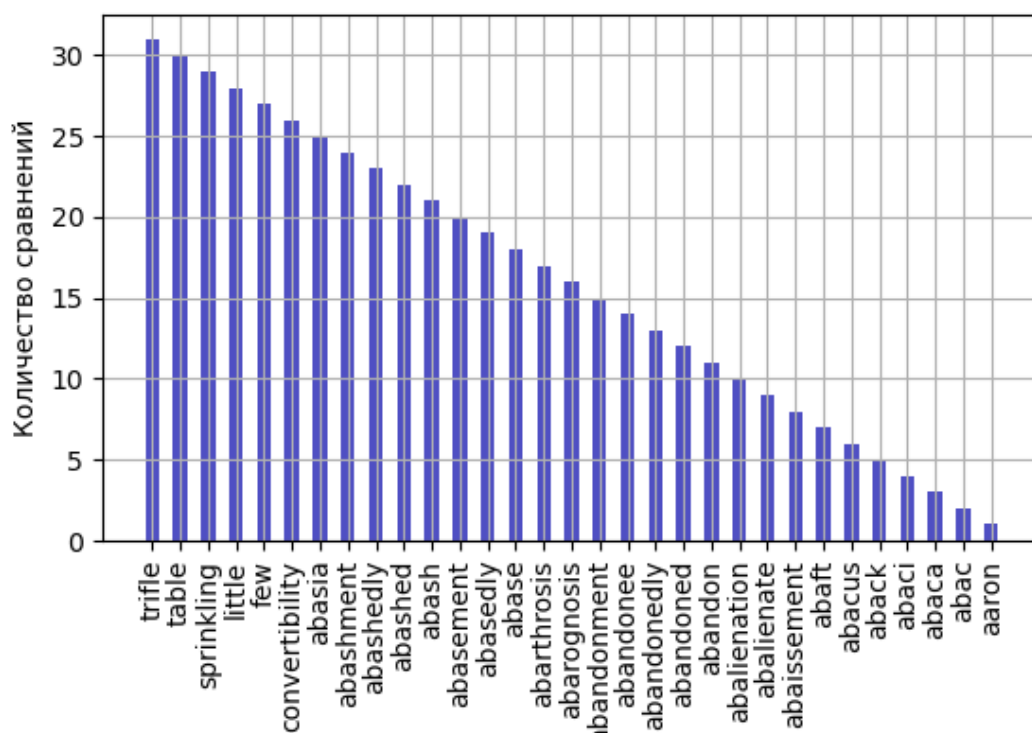


Рисунок 4.4 – Количество сравнений: полный перебор, сортировка по сравнениям

нет в словаре, потребуется столько же сравнений, сколько и для худшего случая.

Из рисунков 4.5 и 4.6 видно, что количество сравнений, необходимых для поиска данного ключа бинарным поиском также определяется позицией ключа в словаре, но здесь зависимость более сложная, так как надо учитывать еще и ход разбиения исходного словаря. Наименьшее количество сравнений (1) потребуется для срединного ключа, далее 2 сравнения для элементов на границе $1/4$ и $3/4$, а наибольшее - $2 * \log N + 1 - 1$ (в данном примере для нечетных). Для определения того, что ключа нет в словаре, потребуется столько же сравнений, сколько и для худшего случая.

Из рисунков 4.7 и 4.8 видно, что количество сравнений, необходимых для поиска данного ключа в сегментированном словаре определяется не только позицией ключа в своем сегменте, но и размером этого сегмента (так как наибольшие сегменты размещены ближе к началу). Наименьшее количество сравнений (2) потребуется для поиска самого первого ключа в

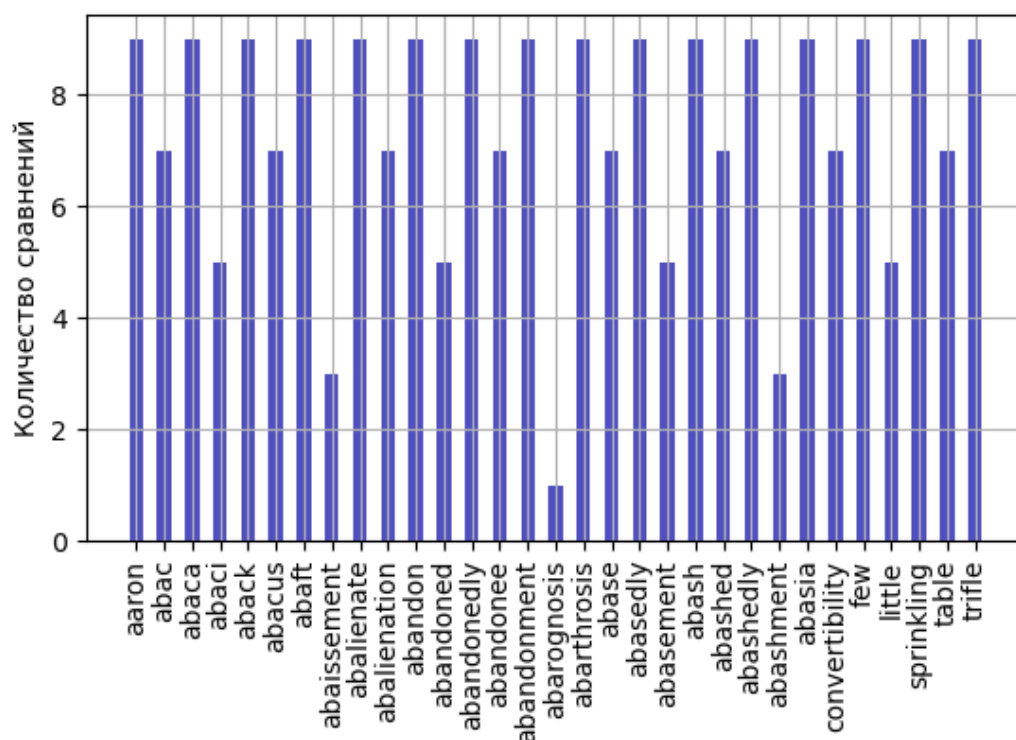


Рисунок 4.5 – Количество сравнений: бинарный поиск, сортировка по расположению

самом большом сегмента, а наибольшее в общем случае определить нельзя: таковым может оказаться, например, ключ на «неудачной» позиции в самом большом сегменте или единственный ключ из самого маленького сегмента. Для определения того, что ключа нет в словаре потребуется столько же сравнений, сколько сегментов в словаре, если других ключей, начинающихся на ту же букву нет, а в противном случае – сумма сравнений для нахождения потенциально подходящего сегмента и определения, что в нем нет искомого ключа.

Таким образом, в общем случае наибольшее количество сравнений требуется в алгоритме полного перебора. Соотношение количества сравнений в бинарном поиске и поиске в сегментированном словаре определяется размером словаря и количеством и размером сегментов в сегментированном словаре.

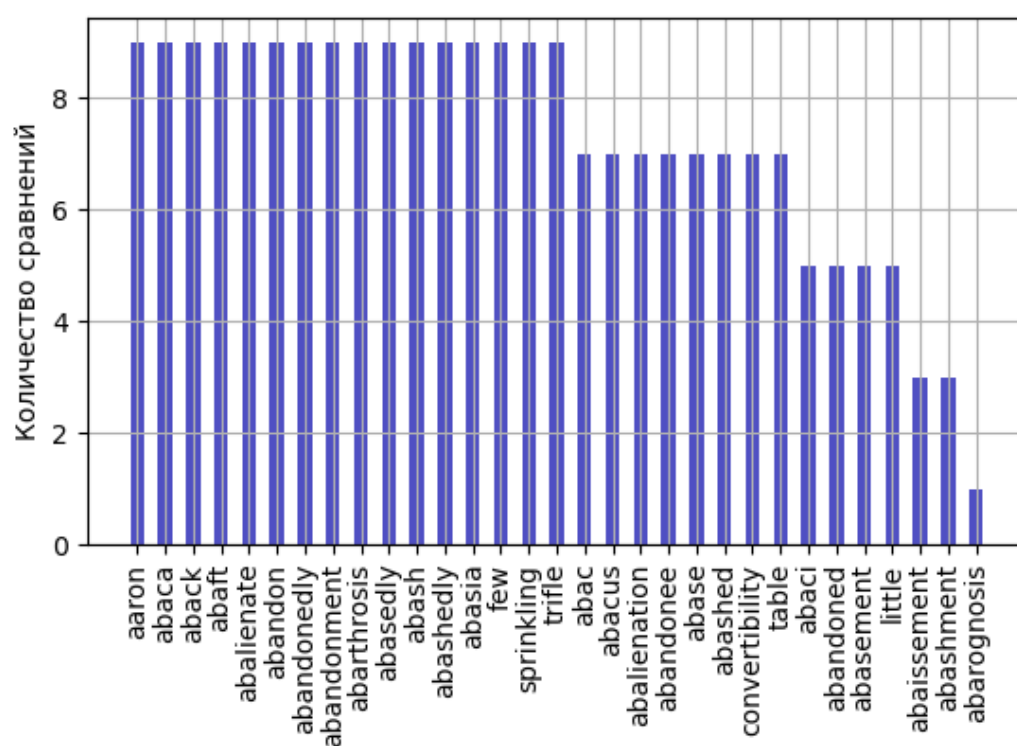


Рисунок 4.6 – Количество сравнений: бинарный поиск, сортировка по сравнениям

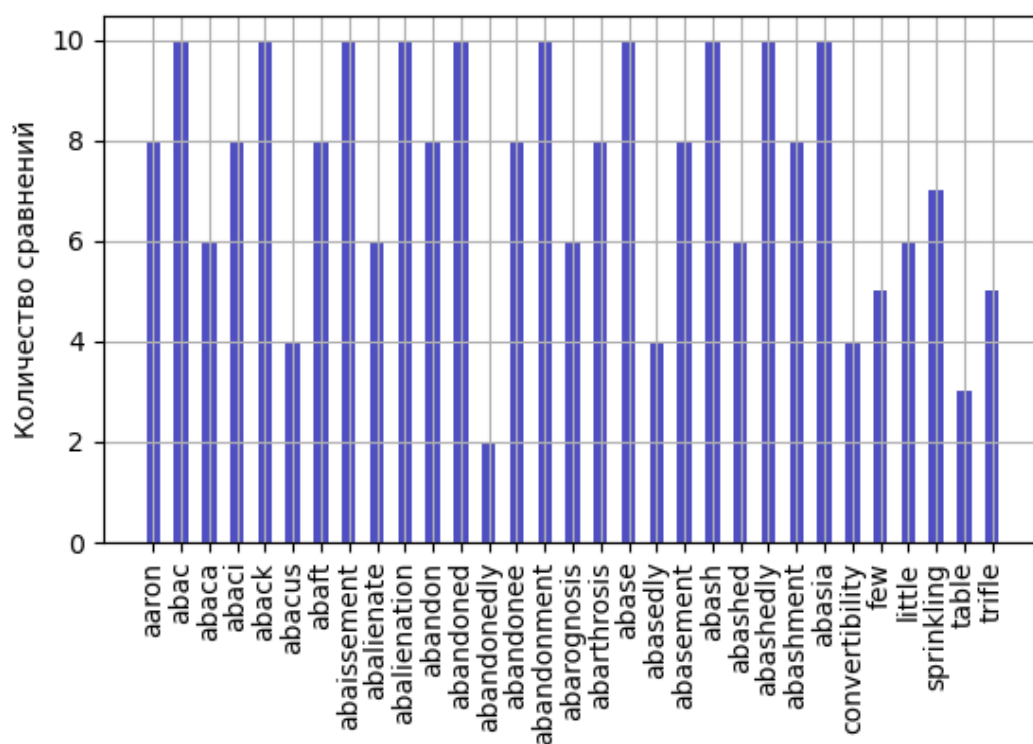


Рисунок 4.7 – Количество сравнений: поиск в сегментированном словаре, сортировка по расположению

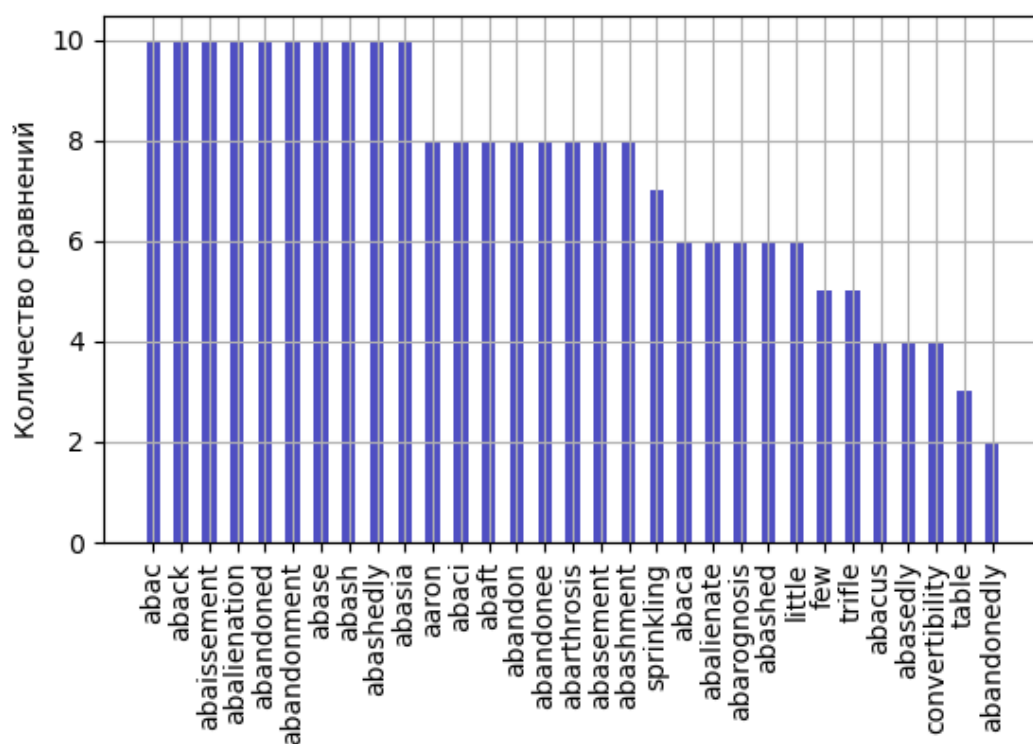


Рисунок 4.8 – Количество сравнений: поиск в сегментированном словаре, сортировка по сравнениям

4.6 Вывод из исследовательской части

Таким образом, в ситуациях, когда количество городов велико (более 8, например), а задача коммивояжера не требует абсолютно точного ответа, для ее решения можно использовать муравьиный алгоритм вместо алгоритма полного перебора, что позволит сэкономить процессорное время. При этом заранее проведенная параметризация может помочь настроить первый алгоритм так, что и он будет в большинстве случаев выдавать точный ответ.

При небольшом же размере графа (примерно до размерности 8×8) муравьиный алгоритм с большой вероятностью даст точный ответ, однако в этом случае он не дает выигрыша по времени перед алгоритмом полного перебора.

Заключение

В результате выполнения лабораторной работы была достигнута поставленная цель: был реализован муравьиный алгоритм для решения задачи коммивояжера и приобретены навыки параметризации алгоритмов.

В рамках выполнения работы были выполнены следующие задачи:

- 1) реализован алгоритм полного перебора для решения задачи коммивояжера;
- 2) изучен и реализован муравьиный алгоритм для решения задачи коммивояжера;
- 3) проведена параметризация муравьиного алгоритма на трех классах данных и подобраны оптимальные параметры;
- 4) проведен сравнительный анализ времени выполнения и трудоемкости реализаций.

Литература

- [1] Множество и словарь [Электронный ресурс]. Режим доступа: https://acm.khpnets.info/w/index.php?title=\T2A\CYRM\T2A\cyrn\T2A\cyro\T2A\cyrrzh\T2A\cyre\T2A\cyrs\T2A\cyrt\T2A\cyrv\T2A\cyro_\T2A\cyri_\T2A\cyrs\T2A\cyr1\T2A\cyro\T2A\cyrv\T2A\cyra\T2A\cyrr\T2A\cyrsftsn._\T2A\CYRR\T2A\cyre\T2A\cyra\T2A\cyr1\T2A\cyri\T2A\cyrz\T2A\cyra\T2A\cyrc\T2A\cyri\T2A\cyrya_\T2A\cyrn\T2A\cyra_\T2A\cyrd\T2A\cyre\T2A\cyrr\T2A\cyre\T2A\cyrv\T2A\cyrsftsn\T2A\cyrya\T2A\cyrh_\T2A\cyrp\T2A\cyro\T2A\cyri\T2A\cyrs\T2A\cyrk\T2A\cyra (дата обращения: 28.11.2021).
- [2] АЛГОРИТМИЧЕСКИЕ СТРАТЕГИИ [Электронный ресурс]. Режим доступа: https://portal.tpu.ru/f_ic/files/school/materials/ppt/2.pdf (дата обращения: 28.11.2021).
- [3] Целочисленный двоичный поиск [Электронный ресурс]. Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=\T2A\CYRC\T2A\cyre\T2A\cyr1\T2A\cyro\T2A\cyrrch\T2A\cyri\T2A\cyrs\T2A\cyr1\T2A\cyre\T2A\cyrn\T2A\cyrn\T2A\cyrrery\T2A\cyrrishrt_\T2A\cyrd\T2A\cyrv\T2A\cyro\T2A\cyri\T2A\cyrrch\T2A\cyrn\T2A\cyrrery\T2A\cyrrishrt_\T2A\cyrp\T2A\cyro\T2A\cyri\T2A\cyrs\T2A\cyrk (дата обращения: 28.11.2021).
- [4] Вставка элемента в отсортированный массив [Электронный ресурс]. Режим доступа: <https://infopedia.su/17x10dd5.html> (дата обращения: 28.11.2021).
- [5] Лутц Марк. Изучаем Python, том 1, 5-е изд. Пер. с англ. — СПб.: ООО “Диалектика”, 2019. Т. 832.
- [6] time — Time access and conversions [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/library/time.html> (дата обращения: 05.09.2021).
- [7] Python и Pycharm [Электронный ресурс]. Режим доступа: <https://py-charm.blogspot.com/2017/09/pycharm.html> (дата обращения: 05.09.2021).