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 в словаре

Дисциплина: Анализ алгоритмов

Студент ИУ7-54Б _____ Л.Е.Тартыков

(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова

(Подпись, дата) (И.О. Фамилия)

Москва, 2021

Содержание

Введение

Словарь или ассоциативный массив – 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.

В паре (k, v) значение v называется значением ассоциированным с ключом k. Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

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, строк.

Поддержка ассоциативных массивов есть во многих языках программирования высокого уровня, таких, как `Python`, `JavaScript` и других. Для языков, не имеющих встроенных средств для работы с ассоциативными массивами, существует множество реализаций в виде библиотек.

Целью данной лабораторной работы является изучение способа эффективного по времени и памяти поиска по словарю. Для достижения данной цели необходимо решить следующие задачи:

- исследовать основные алгоритмы поиска по словарю;
- привести схемы рассматриваемых алгоритмов;
- описать использующиеся структуры данных;
- описать структуру разрабатываемого программного обеспечения;
- определить требования к программному обеспечению;
- привести сведения о модулях программы;
- провести тестирование реализованного программного обеспечения;
- провести экспериментальные замеры характеристик сравнения поиска в реализованных алгоритмах.

1 Аналитическая часть

В данном разделе представлены теоретические сведения о рассматриваемых алгоритмах поиска в словаре.

1.1 Теоретические данные

Словари - объекты, которые записываются парой "ключ-значение". Ключи в словаре должны быть уникальными. Поиск необходимой информации в словаре является одной из фундаментальной задачей программирования.

1.2 Описание словаря

В данной лабораторной работе использован отрывок из книги Шерлок Холмс. Словарь представляет собой ключ "слово в словаре значение - частота появления этого слова в отрывке.

1.3 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.

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

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

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

1.4 Алгоритм двоичного поиска

Бинарный поиск выполняется для отсортированных данных. Он позволяет сравнивать ключ со средним элементом словаря; если он меньше, то продолжается поиск в левой части, иначе - в правой.

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 алгоритма сортировки понадобится всего один раз.

1.5 Разбиение словаря на сегменты

Предлагается разбить словарь на сегменты для повышения оптимизации поиска в нем. Критерием для разбиения выбирается первая буква слова. Когда найден сегмент, в котором лежит слово, выполняется бинарный поиск в это сегменте.

1.6 Вывод

В данной работе описана задача реализации поиска в словаре. Были рассмотрены алгоритмы реализации данного поиска.

Входными данными для программного обеспечения являются:

- словарь из записей, вида

$$\{word : string, value : int\}$$

  для поиска по нему;

- ключ для поиска в словаре.

Выходными данными является найденная в словаре запись для каждого из реализуемых алгоритмов из предложенного для пользователя меню.

2 Конструкторский раздел

В данном разделе представлены описание работы алгоритма; описание структур данных, используемых в алгоритме; описание способа тестирования; приведена оценка памяти для хранения данных и структура программного обеспечения.

2.1 Описание работы алгоритмов

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

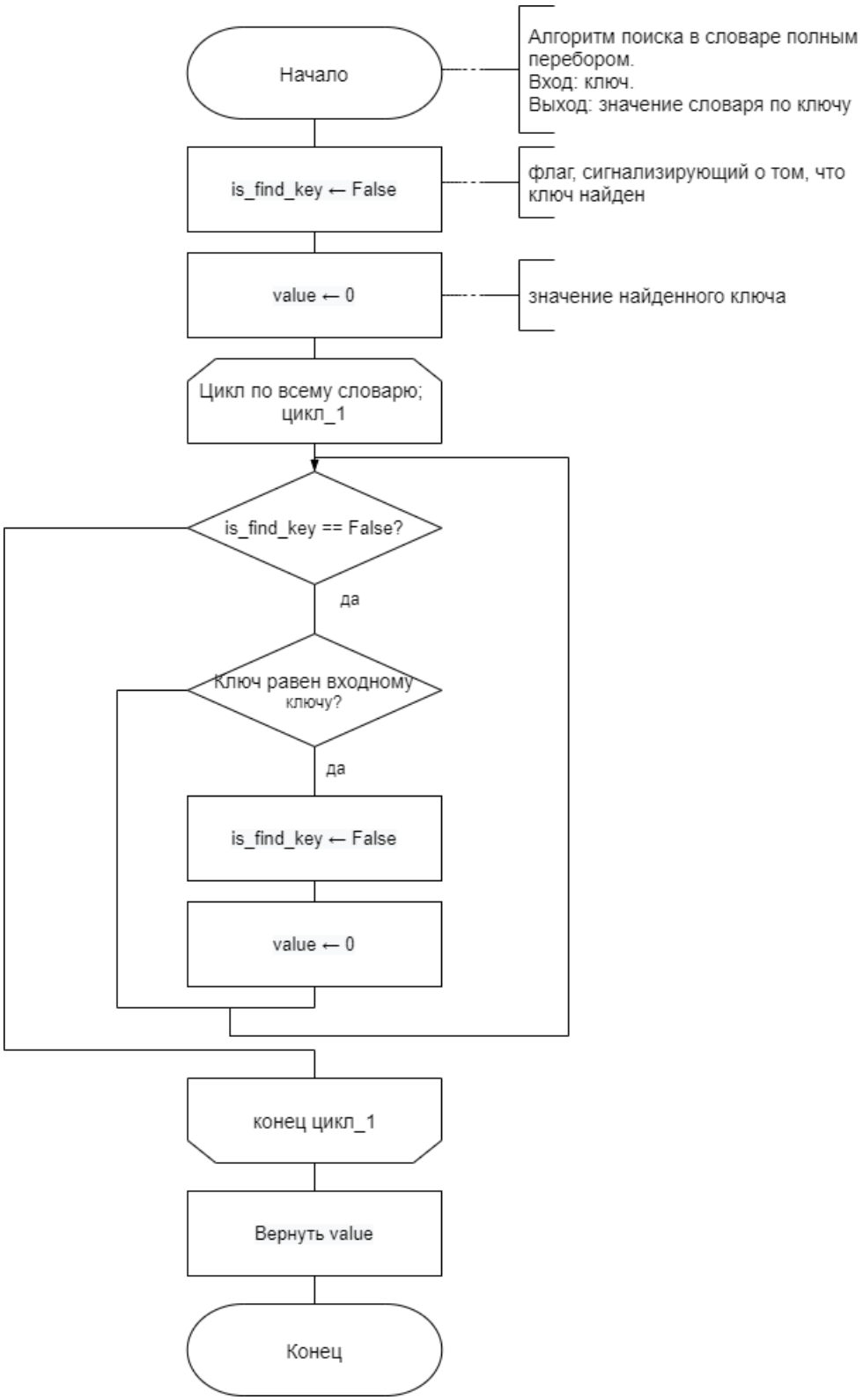

Рисунок 2.1 – Схема алгоритма полного перебора в словаре.

На рисунке 2.2 представлена схема алгоритма бинарного поиска в словаре.

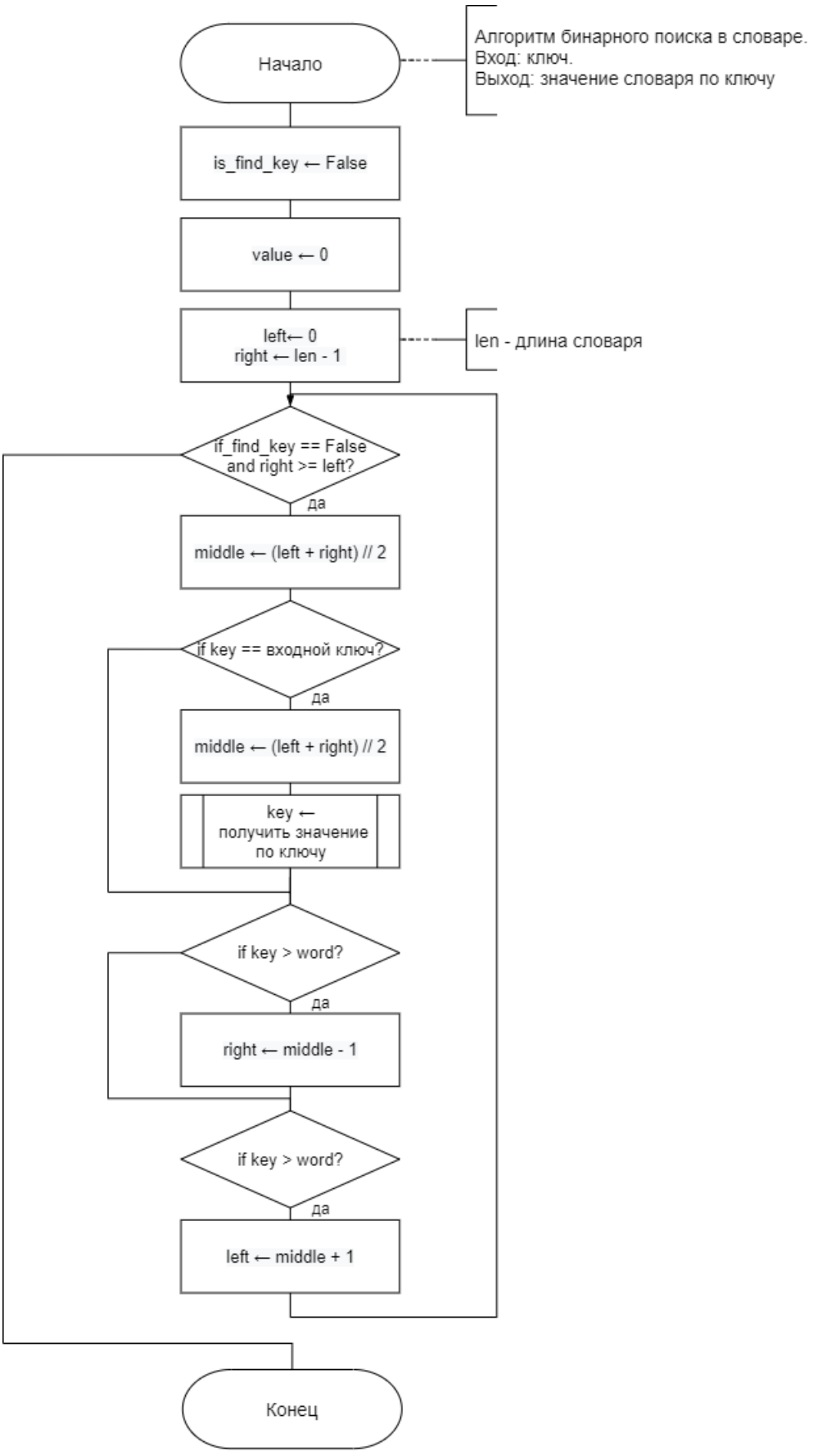

Рисунок 2.2 – Схема алгоритма бинарного поиска в словаре.

На рисунке 2.3 представлена схема алгоритма поиска сегментами с бинарным поиском внутри в словаре.

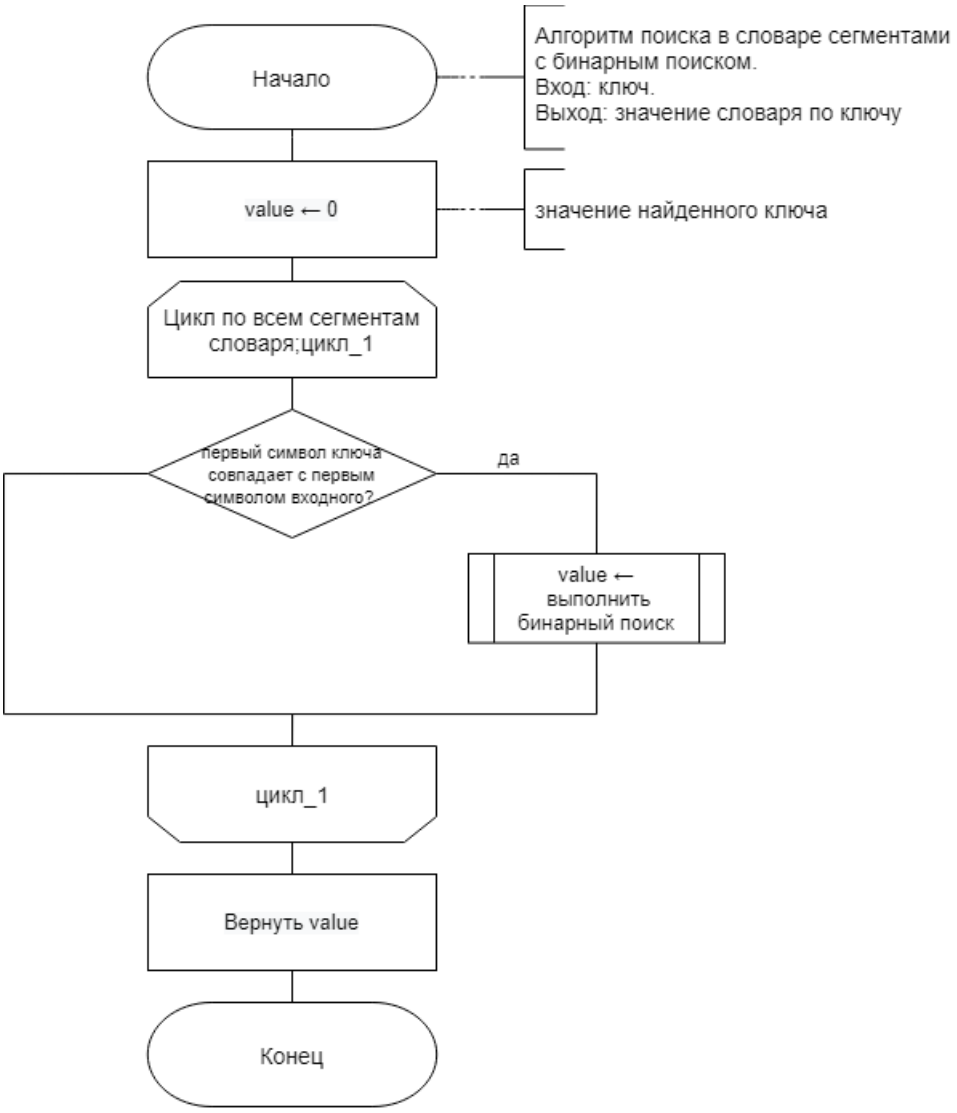

Рисунок 2.3 – Схема алгоритма поиска сегментами в словаре.

2.2 Описание структур данных

Словарь представлен следующей структурой данных:

- Dictionary - тип данных, описывающих ассоциативный массив с ключами типа string.

2.3 Описание способа тестирования и выделенных классов эквивалентности

Тестирование программного обеспечения будет выполнено, используя матрицу смежности, а также набор регулируемых параметров. Матрица является симметричной, диагональные элементы равны нулю.

В качестве классов эквивалентности можно выделить стоимость пути между городами: порядка 50-100 и 500-1000 условных единиц.

2.4 Оценка памяти для хранения данных

Расчет памяти, используемой для хранения словаря, производится по формуле 2.1.

$$M_{dict} = N \cdot (|char| \cdot |key_i| + |int|) \tag{2.1}$$

где N – количество слов в словаре, $|char|$ – размер переменной типа «символ», $|key_i|$ – длина ключа, $|int|$ – размер переменной типа «целое». Расчет памяти, используемой под сегментированный массив, вычисляется по формуле 2.2.

$$M_{seg-dict} = S \cdot M_{dict} \tag{2.2}$$

Где S – количество сегментов.

2.5 Структура программного обеспечения

В качестве парадигмы было использованы структурное программирование в сочетании с объектно-ориентированным. Программное обеспечение состоит из нескольких модулей:

- main - главный модуль, который выполняет вызов функций решения поиска в словаре;
- dictionary - модуль, содержащий необходимые структуры данных и реализацию методов поиска в словаре;

- menu - модуль, содержащий меню для консольного вывода на экран;
- cities - модуль, описывающий необходимые структуры и функции для создания массива городов;
- word - модуль, необходимый для вывода результата найденного значения (или сообщение об ошибке).

2.6 Выделение классов эквивалентности

Для тестирования программного обеспечения выделены следующие случаи:

- искомый ключ присутствует в словаре, в текстовом файле располагается не в первой и не в последней строке;
- искомый ключ присутствует в словаре, в текстовом файле располагается в последней строке;
- искомый ключ присутствует в словаре, в текстовом файле располагается в первой строке;
- искомый ключ не присутствует в словаре.

2.7 Вывод

Были описаны алгоритмы полного перебора и муравьиным методом. Были описаны структуры данных, необходимые для реализации программного обеспечения. Были выделены классы эквивалентности и описан способ тестирования.

3 Технологический раздел

В данном разделе приведены листинги реализованных алгоритмов, требование к программному обеспечению и его тестирование.

3.1 Требования к программному обеспечению

Программа должна удовлетворять следующим требованиям:

- на вход программе подается имя файла, откуда считывается отрывок текста;
- на вход программе подаются корректные данные;
- на выход программа выдает найденное значение по ключу или сообщение об ошибке.

3.2 Выбор средств реализации

Для реализации алгоритмов в данной лабораторной работе был выбран язык программирования Python 3.9.7[**?**]. В качестве среды разработки был использован Visual Studio Code[3], так как в нем присутствуют инструменты для удобства написания и отладки кода.

3.3 Листинги программ

Ниже представлены листинг разработанного модуля метода полного перебора.

Листинг 3.1 – Программный код алгоритма полного перебора.

```
def find_by_full_search(self, word):
    is_find_key = False
    value = 0
    count_compares = 0
    for key in self.dictionary.keys():
        if is_find_key == True:
            break
        if key == word:
            is_find_key = True
            value = self.dictionary[key]
        count_compares += 1
    return value, count_compares
```

Ниже представлены листинг разработанного модуля метода бинарного поиска.

Листинг 3.2 – Программный код алгоритма полного перебора.

```
def find_by_binary_search(self, word, dictionary=None):
    if dictionary == None:
        dictionary = self.sorted_dictionary
        left = 0; right = self.len - 1
    else:
        left = 0; right = len(dictionary) - 1

    is_find = False; value = 0
    list_dictionary = list(dictionary)
    count_compares = 0
    while is_find == False and right >= left:
        middle = (left + right) // 2
        key = list_dictionary[middle]
        if key == word:
            is_find = True
            value = dictionary.get(key)
        elif key > word:
            right = middle - 1
        else:
            left = middle + 1
        count_compares += 1
    return value, count_compares
```

Ниже представлены листинг разработанного модуля сегментного разбиения с бинарным поиском.

Листинг 3.3 – Программный код алгоритма сегментного разбиения с бинарным поиском.

```
def find_by_segment_search(self, word):
    count_compares = 0
    count_segments = len(self.segment_dictionary)
    result = 0; count_companies = 0
    for i in range(count_segments):
        if word[0] == list(self.segment_dictionary[i])[0][0]:
            result, binary_search_compares = self.find_by_binary_search(
                word, self.segment_dictionary[i])
            count_compares += binary_search_compares
            break
    return result, count_compares
```

3.4 Тестирование ПО

Результаты тестирования ПО приведены в таблице 3.1.

Таблица 3.1 – Тестирование ПО

Входные данные	Ожидаемый результат	Результат
and	331	331
together	1	1
1	2	2
wednesday?	not found	not found

3.5 Вывод

Было написано и протестировано программное обеспечение для решения поставленной задачи.

4 Исследовательский раздел

В данном разделе приведены технические характеристики устройства; классы данных, на которых были проведены эксперименты; результаты параметризации и выборка из нее наилучших результатов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Ubuntu 21.10;
- память: 8 GiB;
- процессор: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz.

Тестирование проводилось на ноутбуке, который был подключен к сети питания. Во время проведения тестирования ноутбук был нагружен только встроенными приложениями окружения, самим окружением и системой тестирования.

4.2 Постановка эксперимента

Эксперимент проведен на данных типа "строка". Количество элементов в словаре фиксировано и равно 2241. Проведенный эксперимент устанавливает зависимость количество сравнений при поиске от позиции элемента в словаре.

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования. Оптимизация компилятора была отключена.

4.3 Результаты эксперимента

Результаты эксперимента приведены в приложениях А, Б, В для полного перебора, бинарного поиска и бинарного поиска в сегментированном словаре соответственно.

В среднем, при поиске полным перебором для каждого ключа осуществляется в 131 раз больше сравнений, чем для бинарного поиска и в 206 раз больше сравнений, чем для поиска полным перебором.

4.4 Вывод

Качественная оценка работы алгоритма зависит от количества сравнений с ключами при поиске. В среднем, при поиске полным перебором для каждого ключа осуществляется в 131 раз больше сравнений, чем для бинарного поиска и в 206 раз больше сравнений, чем для поиска полным перебором. Исходя из результатов эксперимента, можно сделать вывод, что самым оптимальным алгоритмом поиска из трех предложенных является алгоритм бинарного поиска с предварительной сегментацией словаря.

Заключение

В данной лабораторной работе были рассмотрены основополагающие материалы, которые потребовались для решения задачи поиска в словаре полным перебором, бинарного поиска и бинарного поиска с предварительной сегментацией словаря. Однако, он требует дополнительных вычислительных затрат на сегментацию словаря и дополнительный объем памяти на хранение выделенных сегментов, что отражает формула 2.2. Были разработано и реализовано программное обеспечение, проведены соответствующие эксперименты.

Опираясь на проведенное исследование, можно сделать вывод, что самым оптимальным подходом к поиску является разделение словаря на сегменты и осуществление бинарного поиска в каждом сегменте, особенно в случаях, когда словарь, подающийся на вход алгоритму, уже сегментирован.

Список литературы

[1] Кнут Д.Э. Искусство программирования. Том 3. Сортировка и поиск. Вильямс, 2001.

[2] Python 3.9 Documentation. [электронный ресурс], режим доступа: https://docs.python.org/3.9/ (дата обращения 16.12.2021)

[3] Documentation for Visual Studio Code. [электронный ресурс], режим доступа: https://code.visualstudio.com/docs (дата обращения: 19.10.2021)

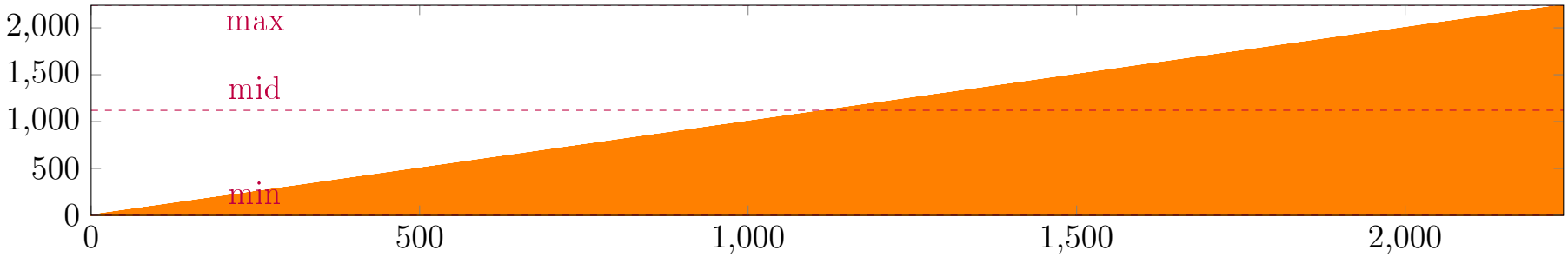

Рисунок 1 – Гистограмма количества сравнений на каждый ключ

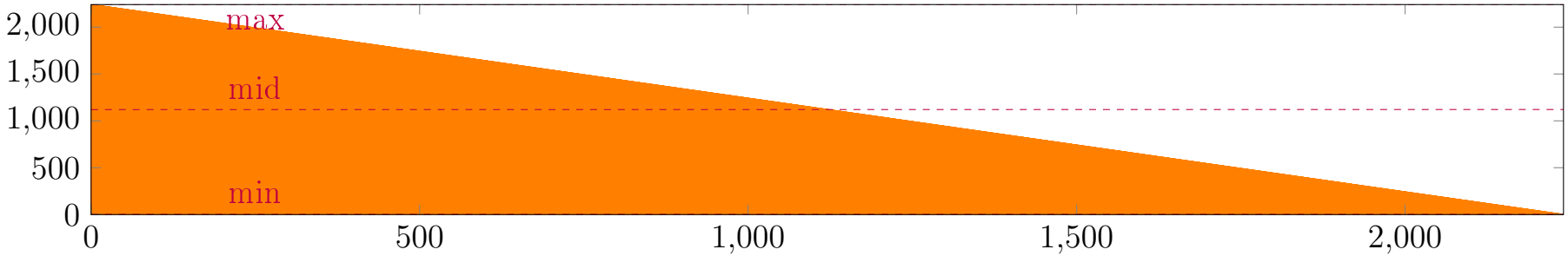

Рисунок 2 – Гистограмма количества сравнений на каждый ключ по убыванию.

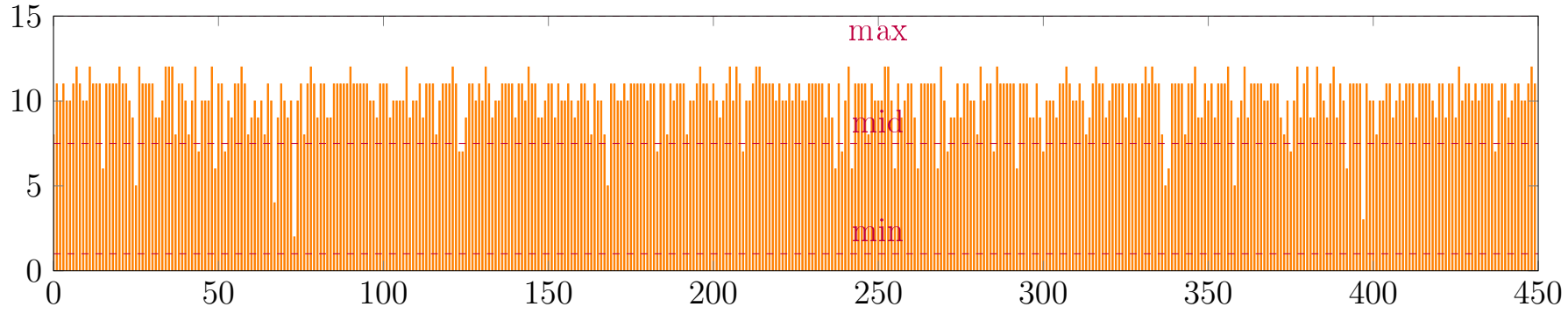

Рисунок 3 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 0 – 450)

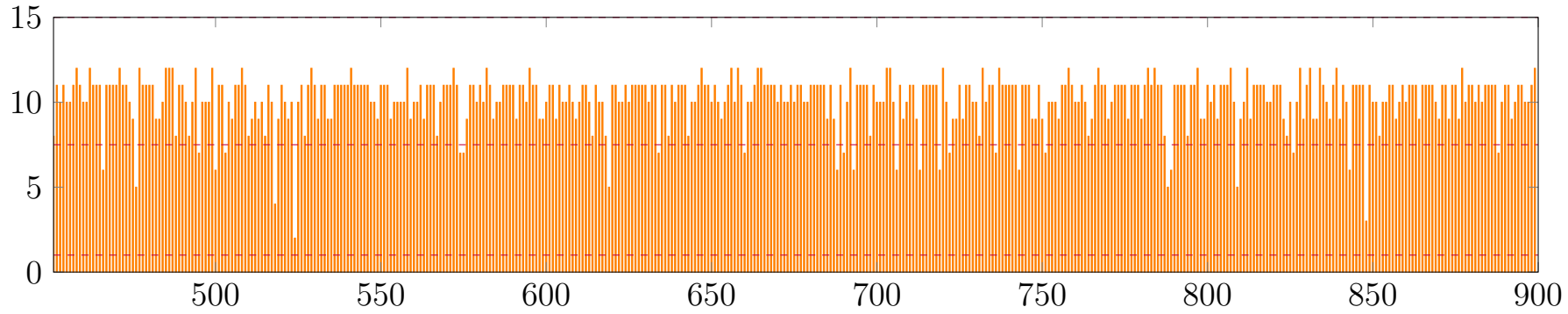

Рисунок 4 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 451 – 900)

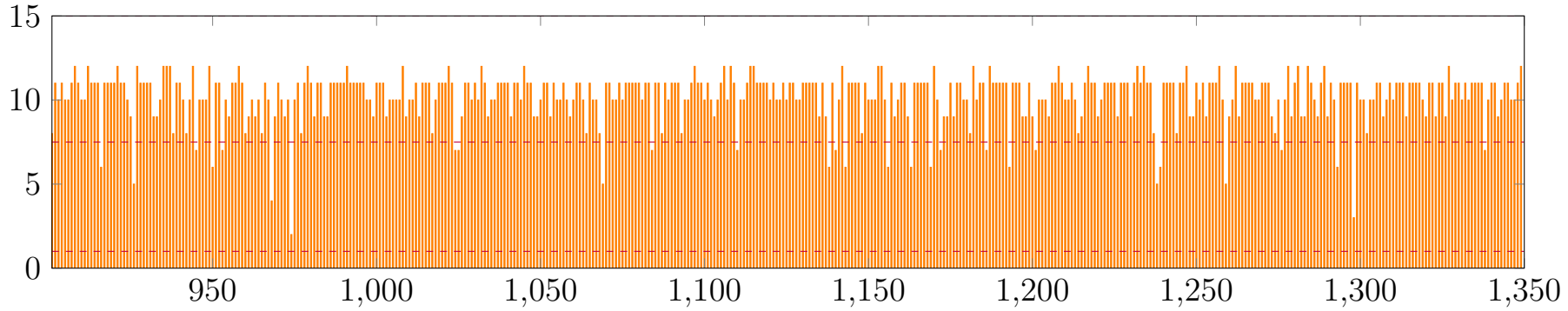

Рисунок 5 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 901 – 1350)

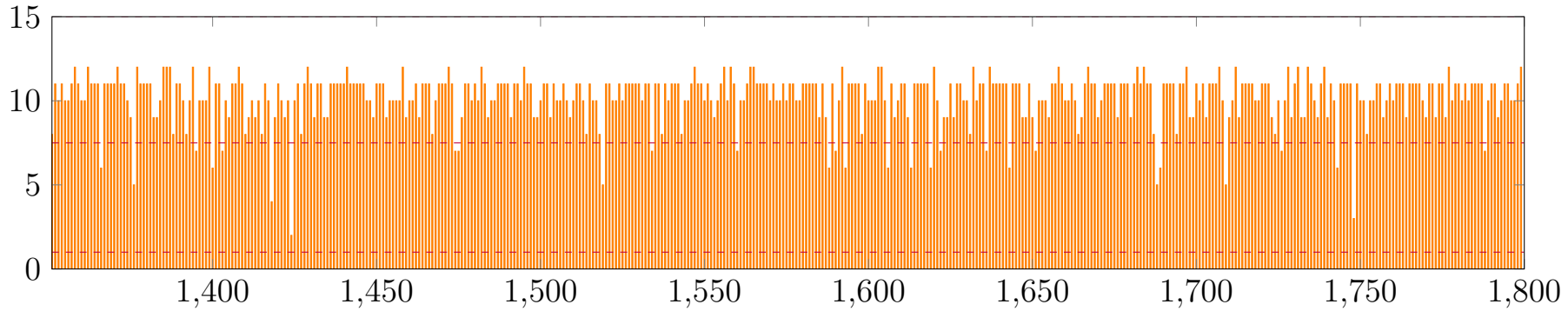

Рисунок 6 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1351 – 1800)

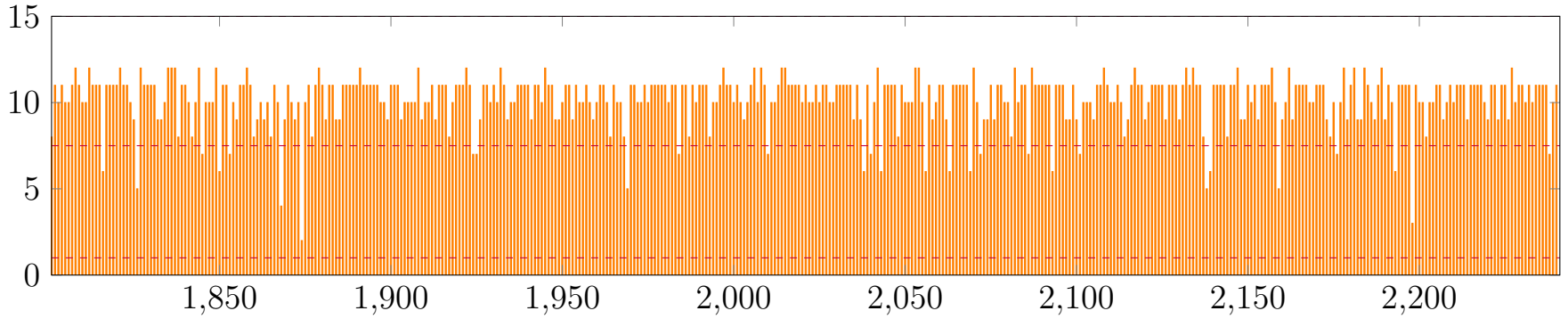

Рисунок 7 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1801 – 2241)

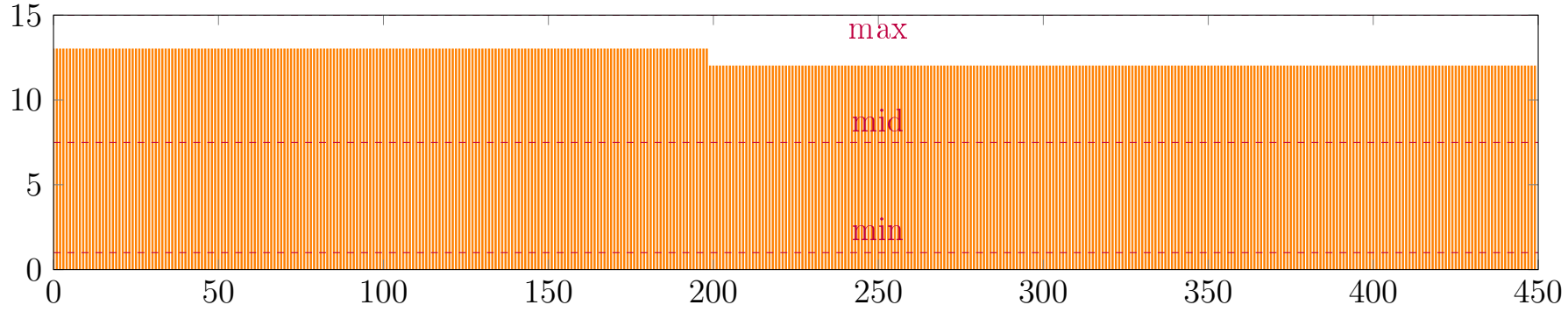

Рисунок 8 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 0 – 450)

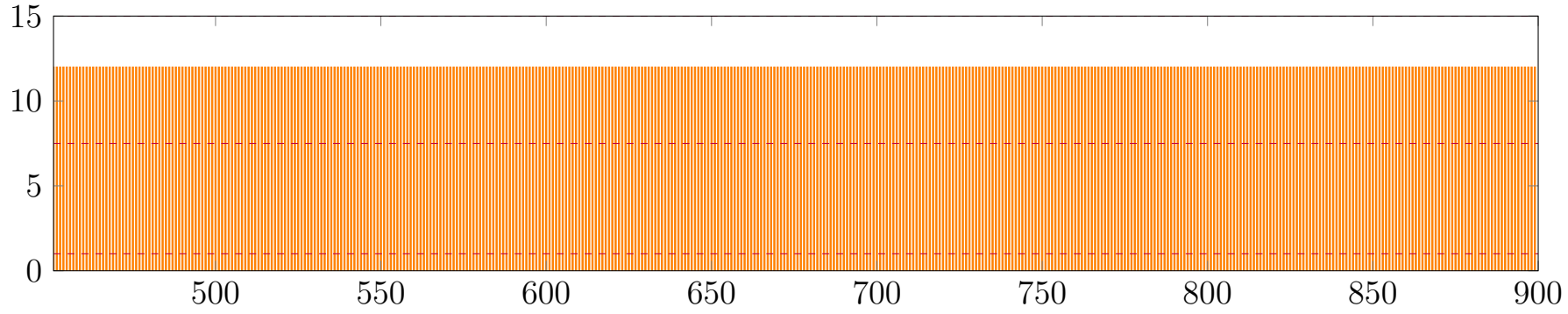

Рисунок 9 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 451 – 900)

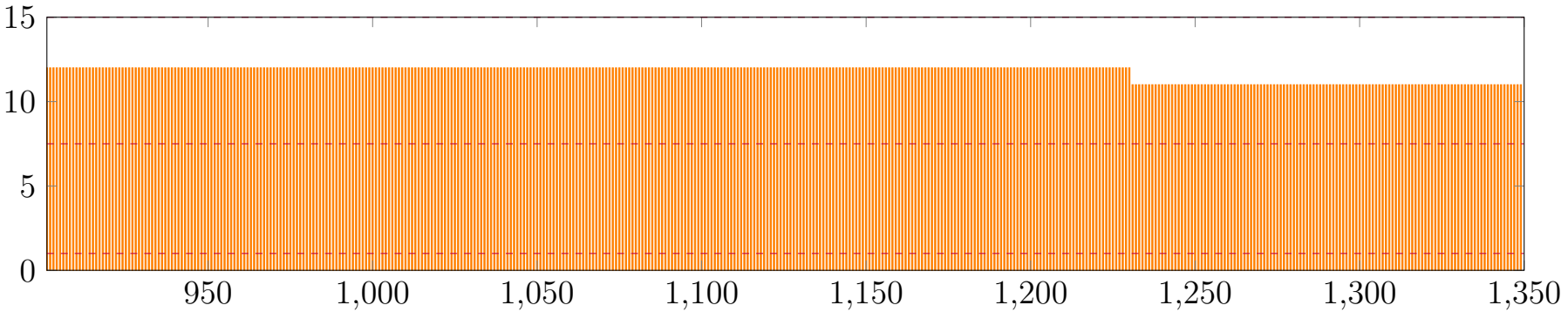

Рисунок 10 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 901 – 1350)

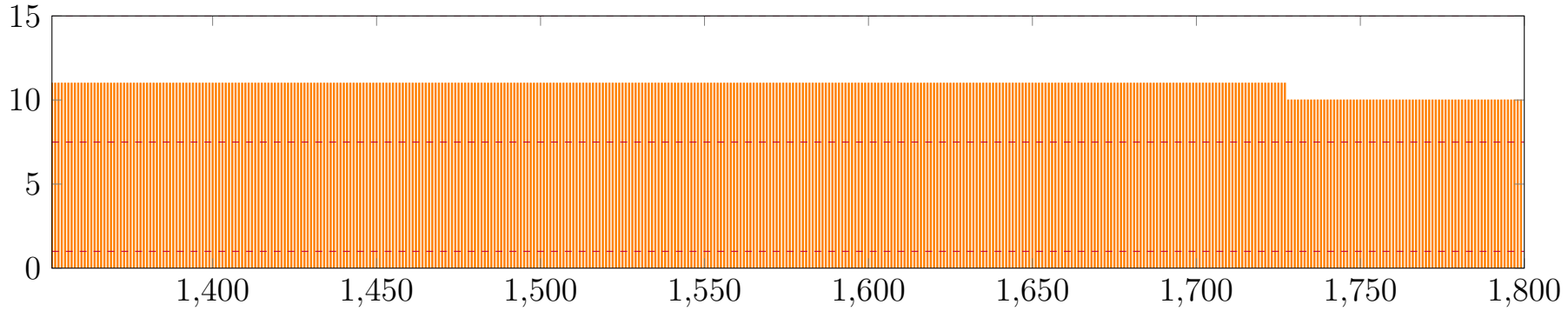

Рисунок 11 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1351 – 1800)

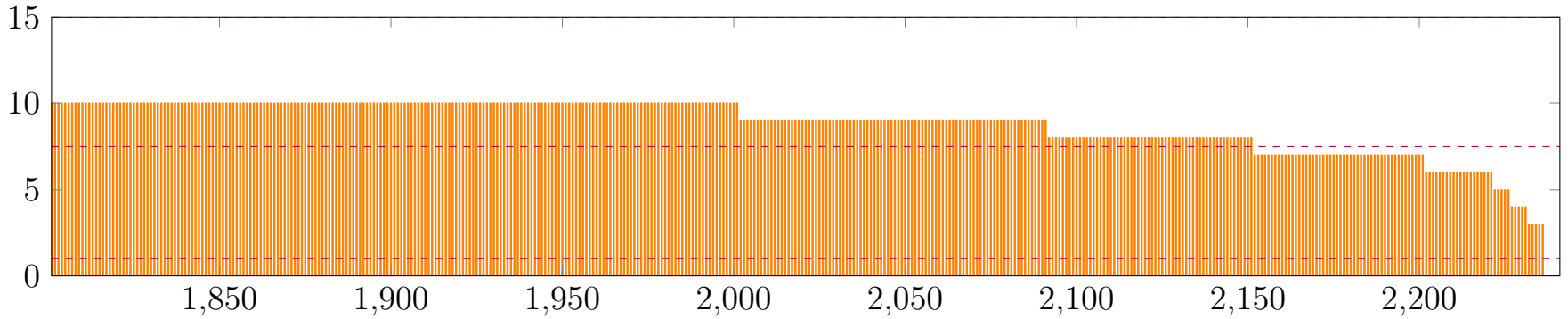

Рисунок 12 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1801 – 2241)

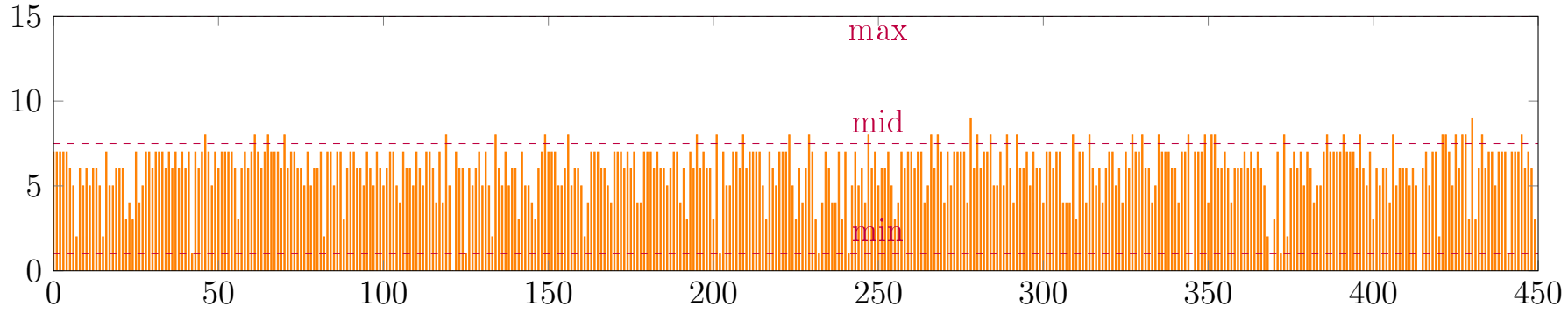

Рисунок 13 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 0 – 450)

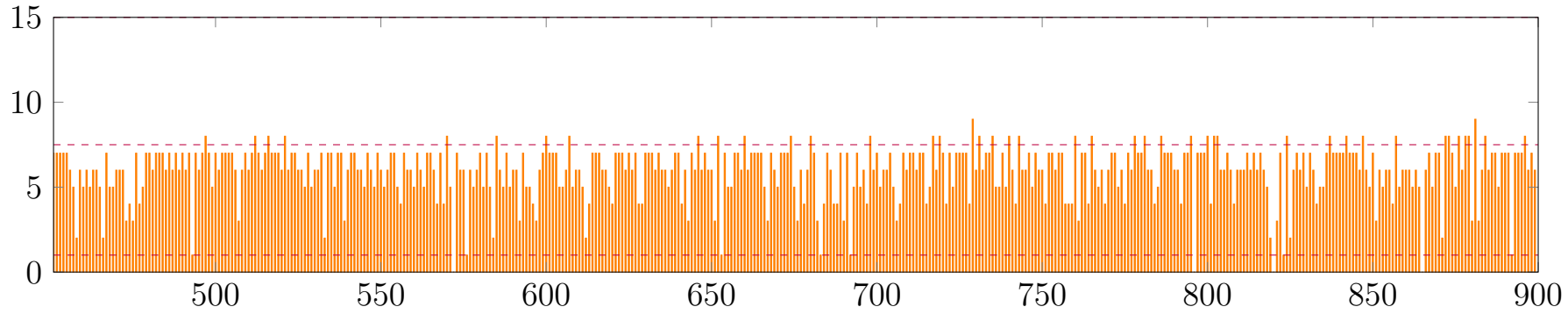

Рисунок 14 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 451 – 900)

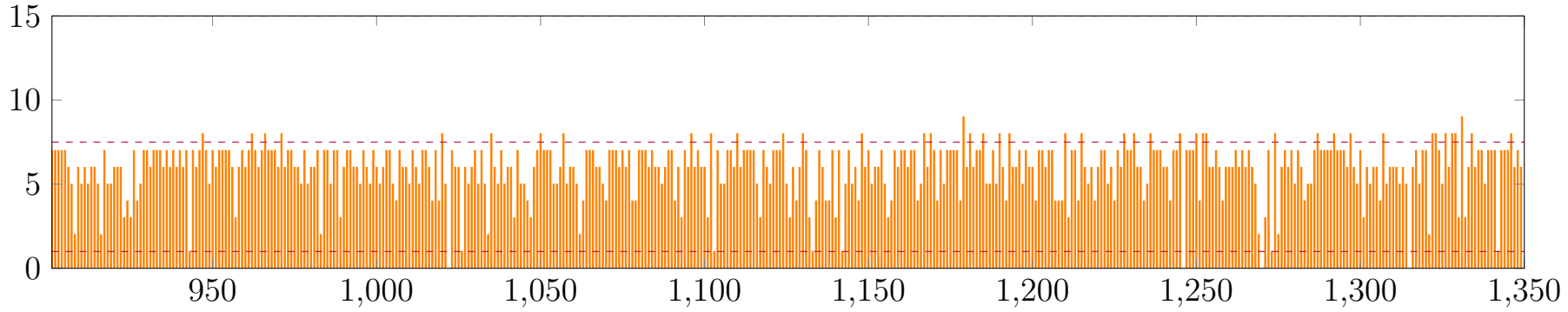

Рисунок 15 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 901 – 1350)

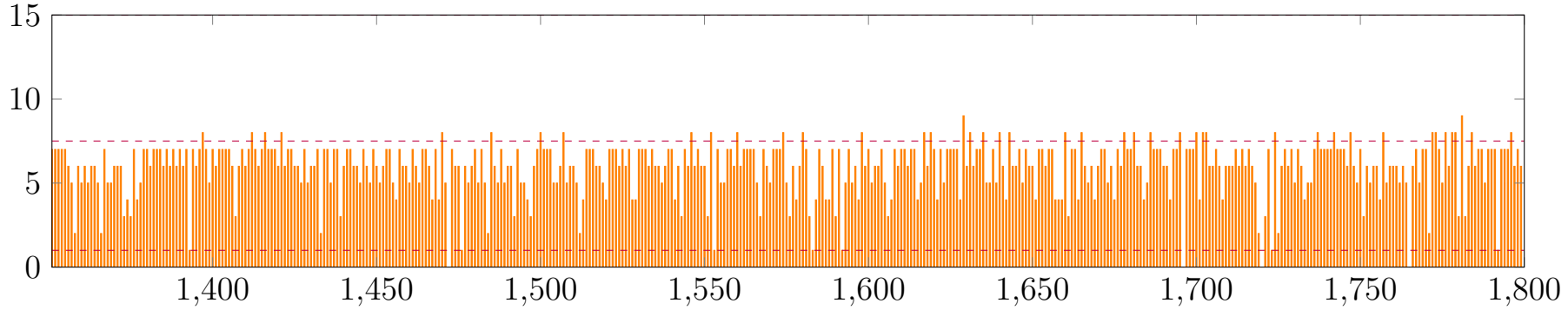

Рисунок 16 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1351 – 1800)

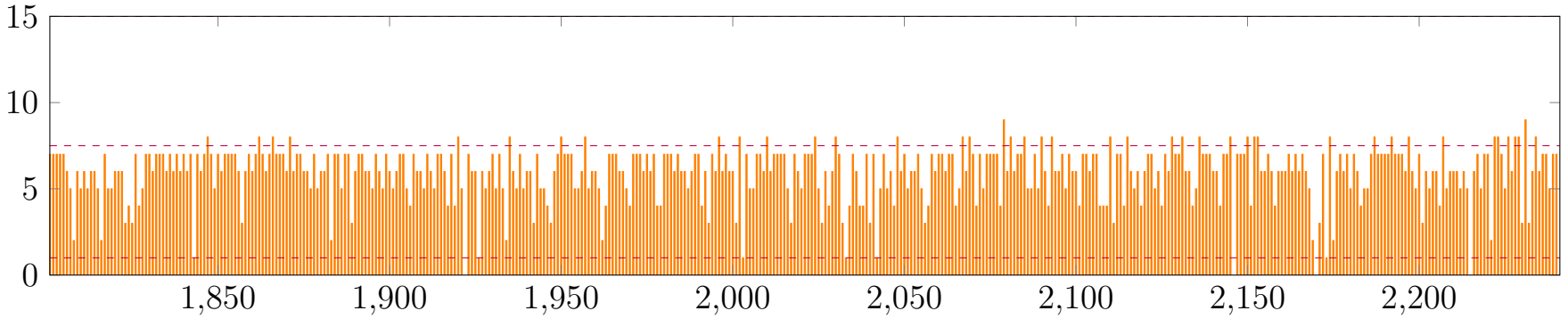

Рисунок 17 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1801 – 2241)

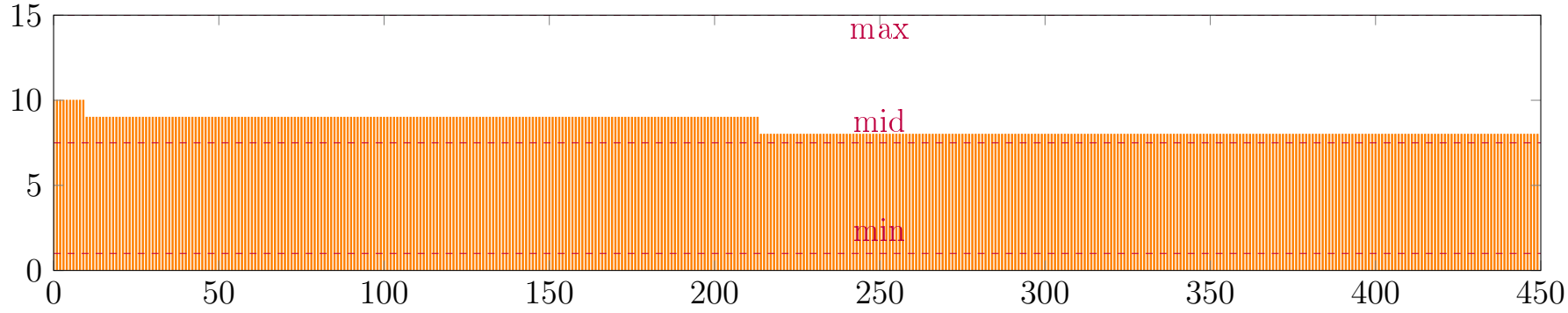

Рисунок 18 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 0 – 450)

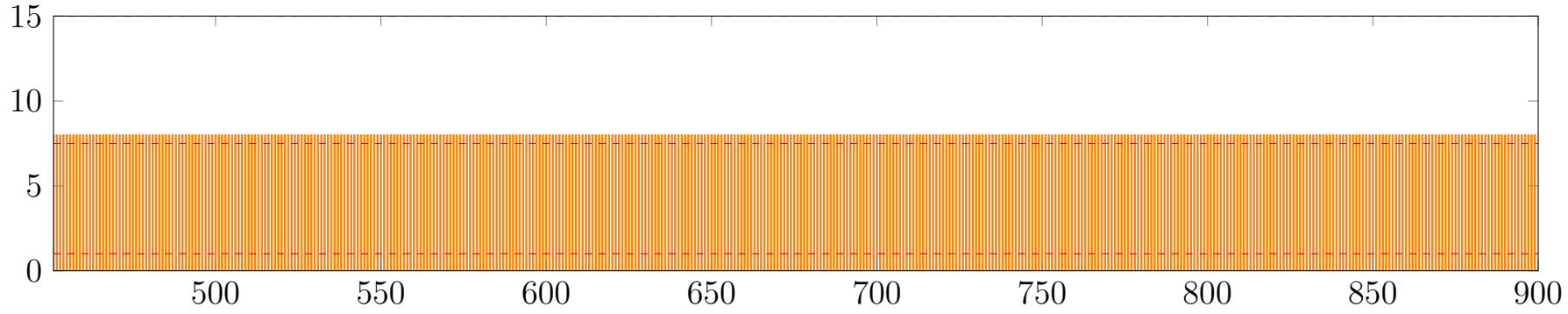

Рисунок 19 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 451 – 900)

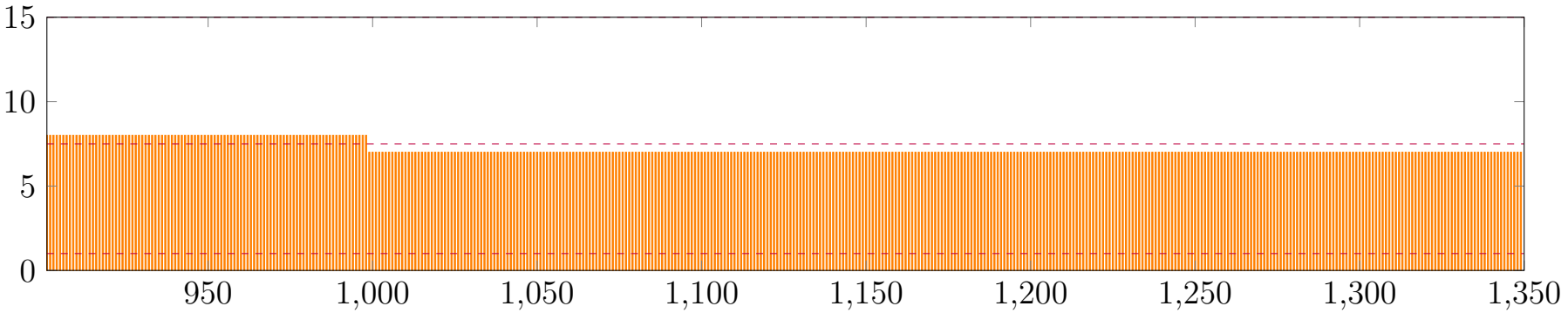

Рисунок 20 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 901 – 1350)

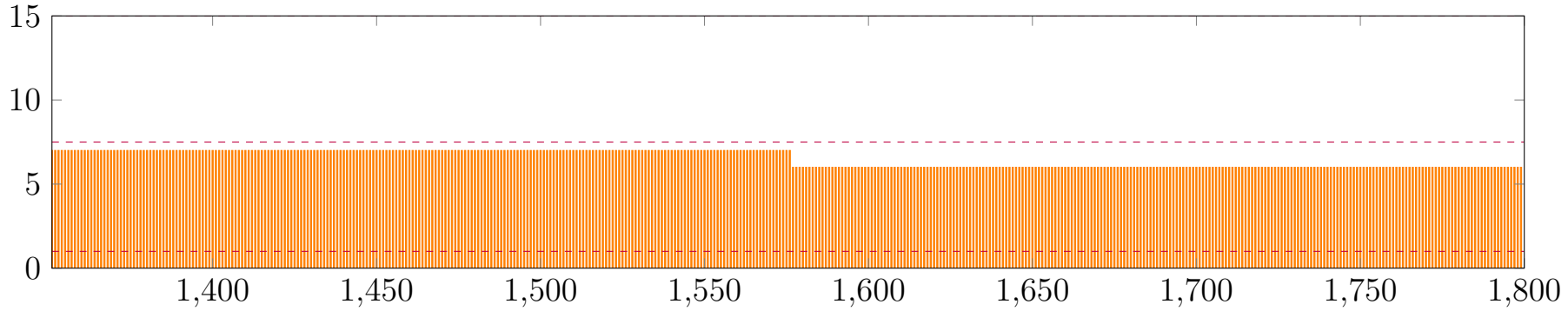

Рисунок 21 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1351 – 1800)

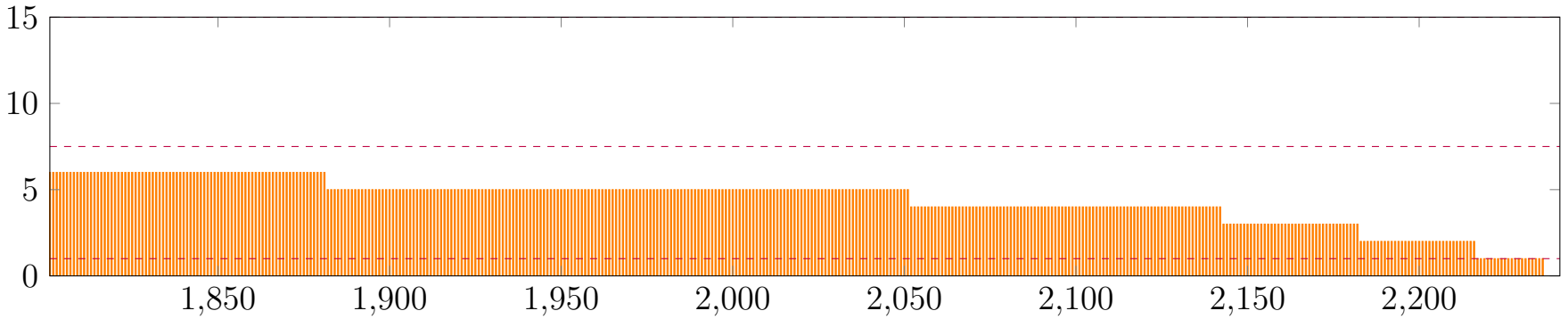

Рисунок 22 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1801 – 2241)