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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №7

Название: Поиск в словаре

Дисциплина: Анализ алгоритмов

Студент	ИУ7-56Б		Т. А. Казаева
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение

С непрерывным ростом количества доступной текстовой информации появляется потребность определенной организации ее хранения, удобной для поиска. Если текстовая информация представляет собой некоторое количество пар, то ее удобно хранить в словаре.

Словарь(ассоциативный массив) – это абстрактный тип данных, состоящий из коллекции элементов вида "ключ – значение".

Словари могут содержать достаточно большие объемы данных, поэтому задача оптимизации поиска в словаре остается актуальной. В предложенной работе представлен анализ трех алгоритмов поиска в словаре: полный перебор, бинарный поиск и максимально эффективный поиск с разбиением словаря ключей на сегменты.

Цель лабораторной работы – анализ предложенных алгоритмов поиска в словаре. Для достижения поставленной цели необходимо выполнить следующие задачи:

- провести анализ алгоритмов полного перебора, бинарного поиска и максимально эффективного поиска с разбиением словаря ключей на сегменты;
- оценить объем памяти для хранения данных;
- разработать и протестировать ПО, реализующее три предложенных алгоритма;
- исследовать зависимость количества сравнений при поиске от способа реализации поиска.

Результаты сравнительного анализа будут приведены в виде гистограмм, из которых можно будет сделать вывод об эффективности каждого из предложенных алгоритмов.

1. Аналитический раздел

Данный раздел содержит информацию о словаре, в котором будет осуществлен поиск, сведения об организации словаря, подходу к сегментированию и описание алгоритмов полного перебора и бинарного поиска.

1.1 Словарь оценочной лексики

В текстах на естественном языке тональность выражена лексическими средствами[1] (словами и выражениями), поэтому для определения тональности естественного текста необходим словарь оценочной лексики.

Словарь предложенный в данной работе, представлен в виде списка слов с атрибутом оценки. Атрибут в данном словаре – это метка полярности («позитивный», «негативный»). Ключом в этом случае является слово, а значением – оценка его тональности. Пример организации словаря:

«annoying» – «*негативный*»
«fidelity» – «*позитивный*»
«award» – «*позитивный*»

При анализе естественного текста в таком словаре осуществляется поиск значения.

1.2 Полный перебор

В алгоритме полного перебора искомый ключ сравнивается со всеми существующими ключами в словаре до тех пор, пока не будет обнаружено совпадение. Время, затрачиваемое на полный перебор зависит от позиции искомого ключа в словаре. Если объем словаря достаточно велик, то полный перебор может потребовать экспоненциального времени работы.

Метод является универсальным. Он прост в исполнении, не требует дополнительных операций и проверок. При небольшом объеме словаря самым оптимальным решением может являться полный перебор.

1.3 Бинарный поиск

При бинарном поиске искомый ключ сравнивается со средним ключом в словаре, в результате этого определяется, в какой половине словаря находится искомый ключ, и снова применяется та же процедура уже к половине словаря[2].

Определение середины словаря на каждой итерации осуществляется по формуле 1.1:

$$mid = \frac{|segment|}{2} \tag{1.1}$$

где часть словаря *segment* была получена в результате применения формулы 1.1 на предыдущей итерации. На первой итерации *segment* содержит весь словарь.

Алгоритм требует меньшего времени работы, чем полный перебор – его трудоемкость $O(\log_2 N)$. Однако, требуется провести дополнительный этап подготовки – словарь следует отсортировать по ключу.

1.4 Разделение словаря на сегменты

Для большей оптимизации поиска в словаре, предлагается разбить его на сегменты и отсортировать сегменты по их размеру. Критерием для разбиения предлагается выбрать первую букву слова. Сегменты при таком разбиении будут существенно варьироваться в размере, что подтверждает статистика на 2012 год [3]. При таком разбиении повышается вероятность того, что позиция искомого слова будет ближе к началу словаря, что значительно ускорит поиск.

1.5 Вывод

Программное обеспечение, решающее поставленную задачу, может работать следующим образом. На вход алгоритму подается файл со словарем и искомый ключ, пользователь выбирает алгоритм поиска. Программа возвращает значение по искомому ключу и количество сравнений при поиске.

2. Конструкторский раздел

Раздел содержит описание работы алгоритмов и обоснование структур данных, выбранных при их реализации, оценку используемой памяти и описание системы тестирования программного обеспечения.

2.1 Описание структур данных

Словарь представлен в качестве массива с прямой адресацией, содержащий пары «ключ» – «значение». Выбор структуры данных «массив» обусловлен константной вычислительной сложностью доступа к элементу, что упростит реализацию алгоритма бинарного поиска.

Сегментированный словарь представлен в памяти как массив ассоциативных массивов, содержащих пары «ключ» – «значение». Причины выбора данной структуры аналогичны причинам описанным выше.

Метка полярности в словаре представлена как целое число, -1 характеризует отрицательную полярность, 1 – положительную.

2.2 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.

2.3 Схемы алгоритмов

На рисунке 2.1 представлена схема работы алгоритма полного перебора. Обращение к элементам массивов на схеме алгоритмов обозначено подстрочными индексами. Операция присваивания обозначена как «←», операции «равно» и «не равно» как «=» и «#» соответственно.

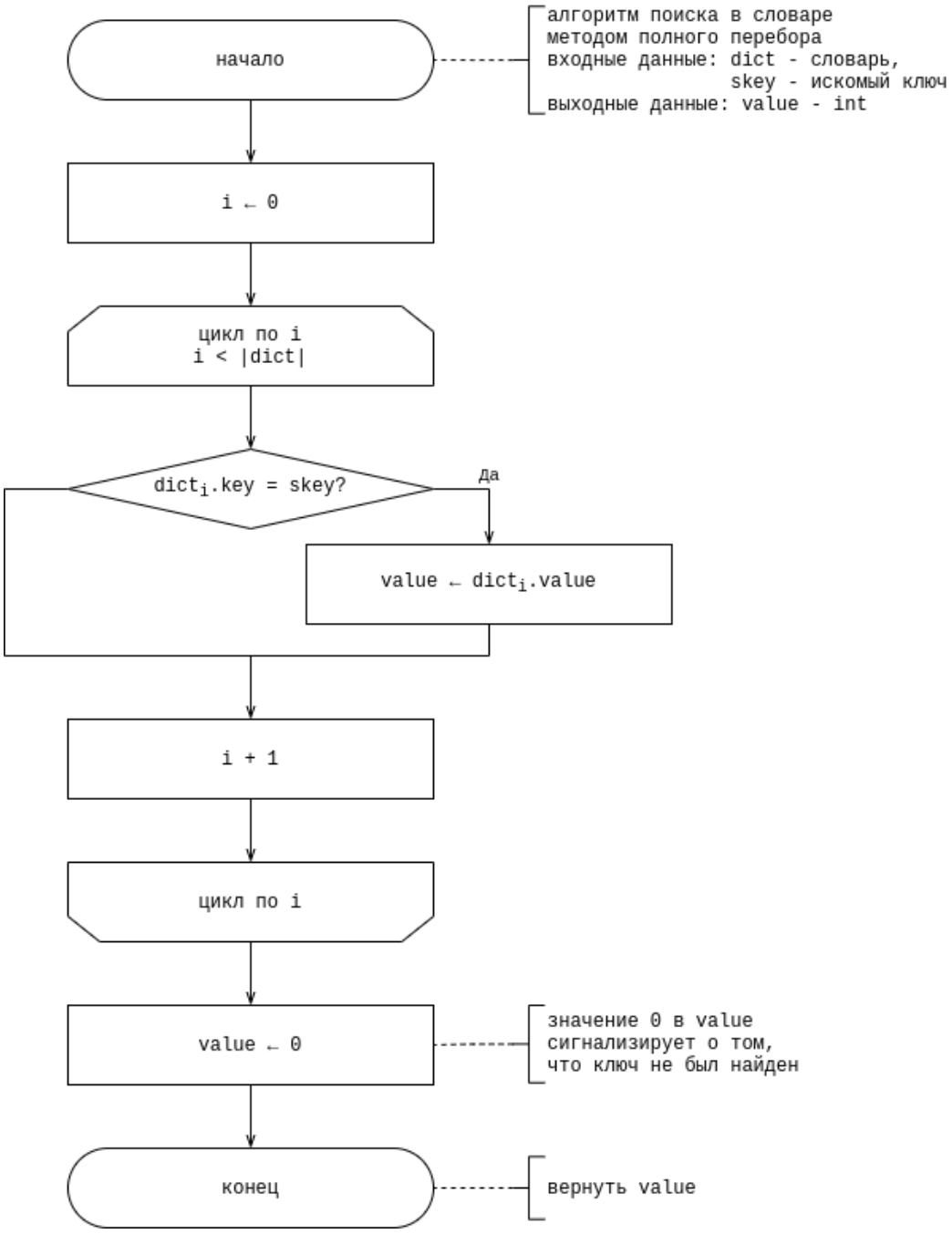

Рисунок 2.1 – Схема работы алгоритма полного перебора

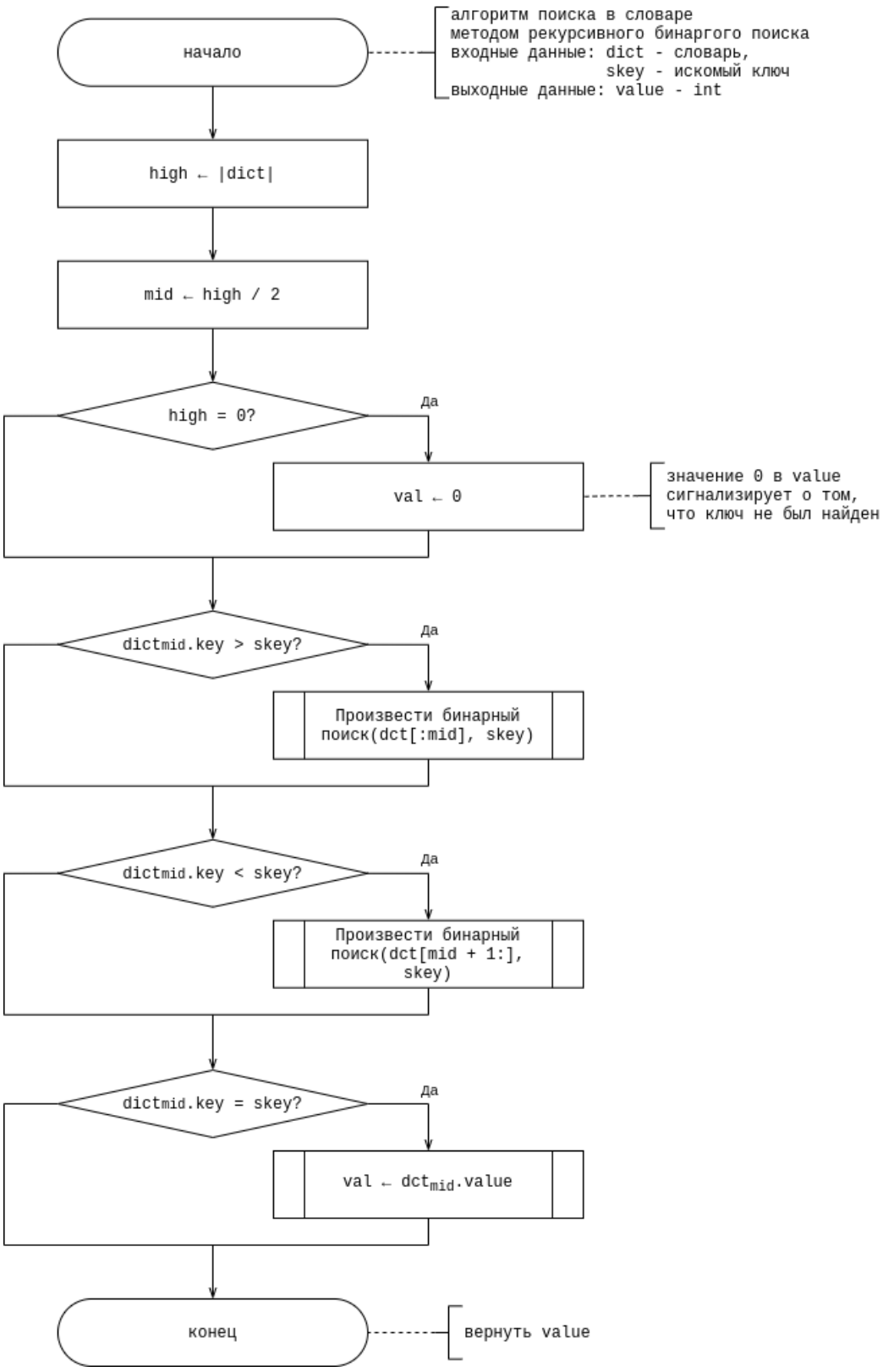

Рисунок 2.2 – Схема работы алгоритма бинарного поиска

На рисунке 2.2 представлена схема работы алгоритма бинарного поиска. Операция взятия среза обозначена как «[:]».

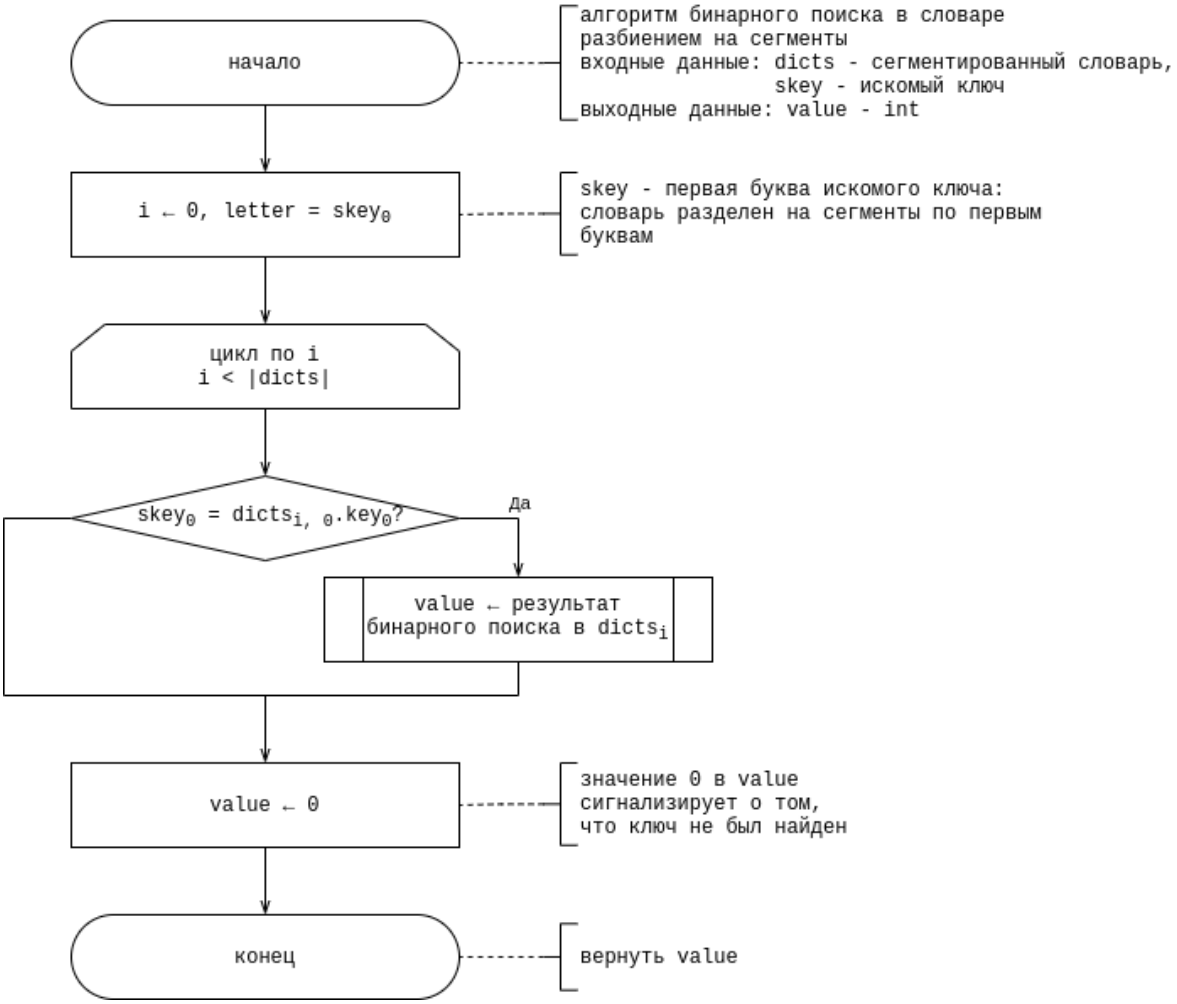

Рисунок 2.3 – Схема работы алгоритма бинарного поиска для словаря, поделенного на сегменты

На рисунке 2.3 представлена схема работы алгоритма бинарного поиска для словаря, поделенного на сегменты. Вызываемая в теле условного оператора функция представлена на рисунке 2.2.

2.4 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.

2.5 Структура ПО

ПО имеет один модуль *dictry*, содержащий функции реализации поиска в словаре и функцию чтения словаря из внешнего ресурса.

2.6 Вывод

Были разработаны схемы алгоритмов, необходимых для решения задачи. Получено достаточно теоретической информации для написания программного обеспечения.

3. Технологический раздел

Раздел содержит листинги реализованных алгоритмов, требование к ПО и тестирование реализованного программного обеспечения.

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- программа получает на вход имя текстового файла со словарем и искомый ключ;
- программа выдает значение по искомому ключу. Если ключа в словаре не нашлось, программа выдает текстовую строку *NOT FOUND*.

3.2 Средства реализации

Для реализации ПО был выбран компилируемый многопоточный язык программирования Golang[4], поскольку язык отличается автоматическим управлением памяти. В качестве среды разработки была выбрана среда VS Code, написание сценариев осуществлялось утилитой make.

3.3 Листинги кода

Листинг 3.1 демонстрирует реализацию типа «словарь» и «сегментированный словарь». Словарь содержит пары значений, описанных структурой «Entry».

Листинг 3.1 – Структуры данных

```
type Entry struct {
    Key   string
    Value int
}

type Dictionary struct {
    Size    int
    entries []Entry
}

type SgtDictionary struct {
    Size    int
    dicts   []Dictionary
}
```

Листинг 3.2 демонстрирует реализацию алгоритма полного перебора.

Листинг 3.2 – Полный перебор

```
func (dct Dictionary) BruteForce(skey string) (int, int, error) {
    compars := 0

    for _, entry := range(dct.entries) {
        compars++;
        if (entry.Key == skey) {
            return entry.Value, compars, nil;
        }
    }
    return 0, compars, errors.New("NOT FOUND")
}
```

Функция 3.3 является «оберткой» для бинарного поиска, вызываемой из пользовательского интерфейса:

Листинг 3.3 – Бинарный поиск(функция - обертка)

```
func (dct Dictionary) BinarySearch(skey string) (int, int, error) {
    dct.sortAlphab()

    r, compars, err := dct.__binaryHelper(skey)
    return r, compars, err
}
```

Сам бинарный поиск осуществляет функция, демонстрируемая на листинге 3.4.

Листинг 3.4 – Бинарный поиск

```
func (dct Dictionary) __binaryHelper(skey string)
                     (result int, compars int, err error) {
    var (
        high    int = len(dct.entries)
        mid     int = high / 2
    )
    switch {
        case high == 0:
        result = 0
        err    = errors.New("NOT FOUND")
        case dct.entries[mid].Key > skey:
        dct.entries = dct.entries[:mid]
        result, compars, err = dct.__binaryHelper(skey)
        case dct.entries[mid].Key < skey:
        dct.entries = dct.entries[mid + 1:]
        result, compars, err = dct.__binaryHelper(skey)
        default:
        result = dct.entries[mid].Value
    }
    compars++
    return
}
```

Поиск в сегментированном словаре осуществляется с помощью функции, представленной на листинге 3.5.

Листинг 3.5 – Бинарный поиск в сегментированном словаре

```
func (dct Dictionary) SectionedBinary(skey string) (int, int, error) {
    var (
        sec_compars int = 0
        sletter byte = skey[0]
        rvalue  int
        err     error
        compars int
    )
    sdct := dct.SegmentByAlphabet()
    rvalue = 0
    for _, d := range sdct.dicts {
        sec_compars++
        if sletter == d.entries[0].Key[0] {
            rvalue, compars, err = d.BinarySearch(skey)
            break
        }
    }
    return rvalue, compars + sec_compars, err
}
```

3.4 Тестирование ПО

Результаты тестирования ПО приведены в таблице 3.1.

Таблица 3.1 – Тестирование ПО

Входные данные	Результат	Ожидаемый результат
wanna	1	1
grab	1	1
coffee	-1	-1
wednesday?	NOT FOUND	NOT FOUND

3.5 Вывод

Было написано и протестировано программное обеспечение для решения поставленной задачи.

4. Исследовательский раздел

Раздел содержит технические характеристики устройства, на котором проведен эксперимент. Также раздел содержит результаты проведенного эксперимента.

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- операционная система Ubuntu 20.04.1 LTS;
- память 7 GiB;
- процессор Intel(R) Core(TM) i3-8145U[5] CPU @ 2.10GHz.

4.2 Постановка эксперимента

Эксперимент проведен на данных типа "строка". Количество элементов в словаре фиксировано и равно 2883. 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 и В для полного перебора, бинарного поиска и бинарного поиска в сегментированном словаре соответственно.

Медиана подсчитанных количеств сравнений для метода полного перебора равна 1442, для бинарного поиска равна 11 и для бинарного поиска в сегментированном словаре равна 7.

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

При повсеместном использовании словарей растет потребность в оптимизации алгоритмов поиска в словаре. В ходе лабораторной работы был проведен анализ алгоритмов поиска полным перебором, бинарного поиска и бинарного поиска с предварительной сегментацией словаря. Результат эксперимента показал, что наиболее качественно задачу решает алгоритм 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. 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] Клышинский Э.С. Ефремова Н.Э. *Автоматическая обработка текстов на естественном языке и анализ данных.*

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

[3] *English Letter Frequency Counts.* URL: `http://norvig.com/mayzner.html` (дата обр. 26.11.2021).

[4] *Go.* URL: `https://go.dev/` (дата обр. 26.11.2021).

[5] *Процессор Intel® Core™ i3-8145U.* URL: `https://ark.intel.com/content/www/ru/ru/ark/products/149090/intel-core-i3-8145u-processor-4m-cache-up-to-3-90-ghz.html` (дата обр. 26.11.2021).

Приложение А

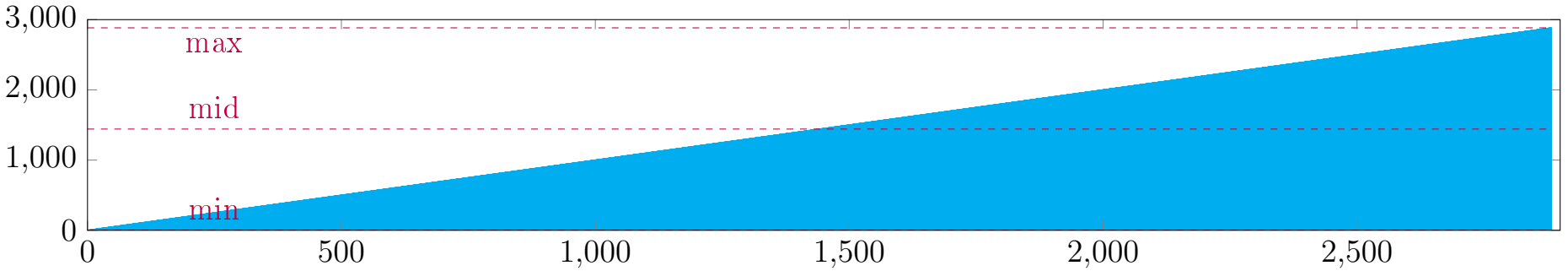

Рисунок 1 – Гистограмма количества сравнений на каждый ключ

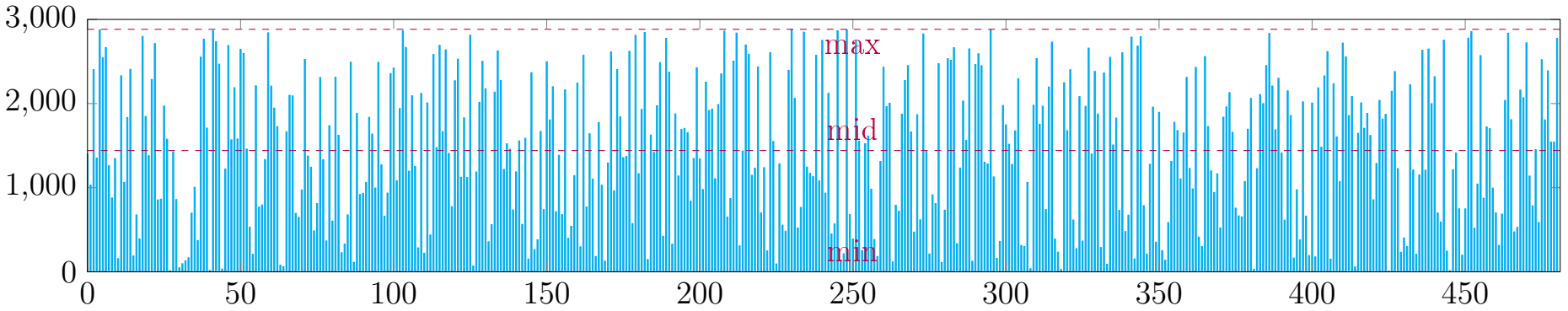

Рисунок 2 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 0 – 481)

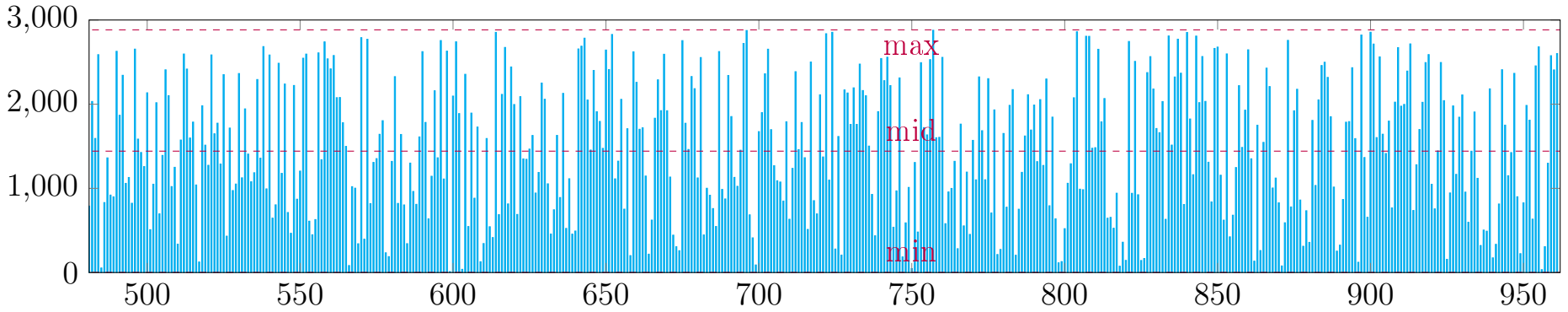

Рисунок 3 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 482 – 962)

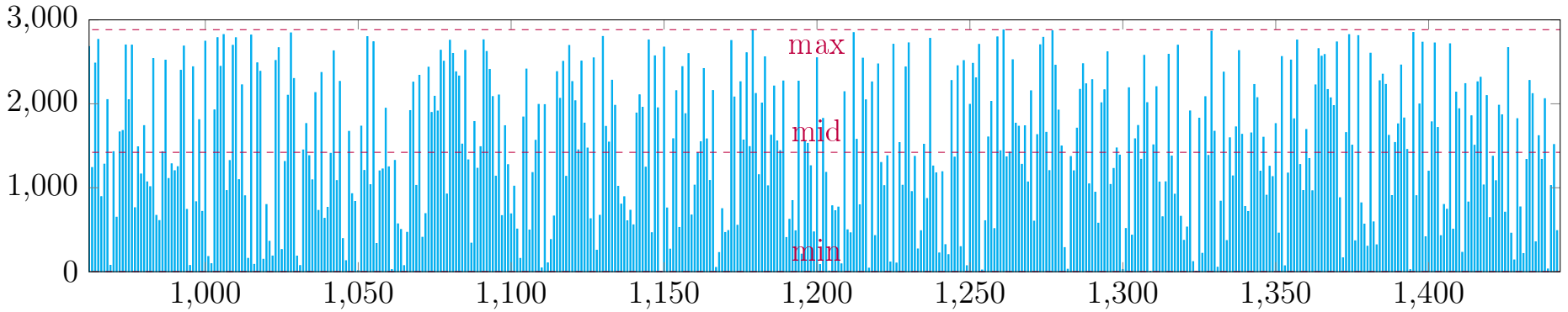

Рисунок 4 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 963 – 1443)

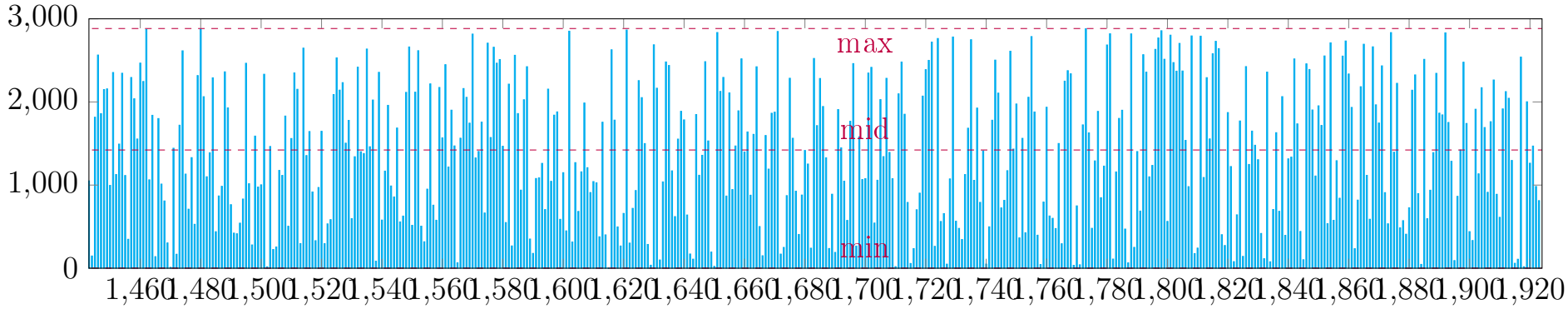

Рисунок 5 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1444 – 1924)

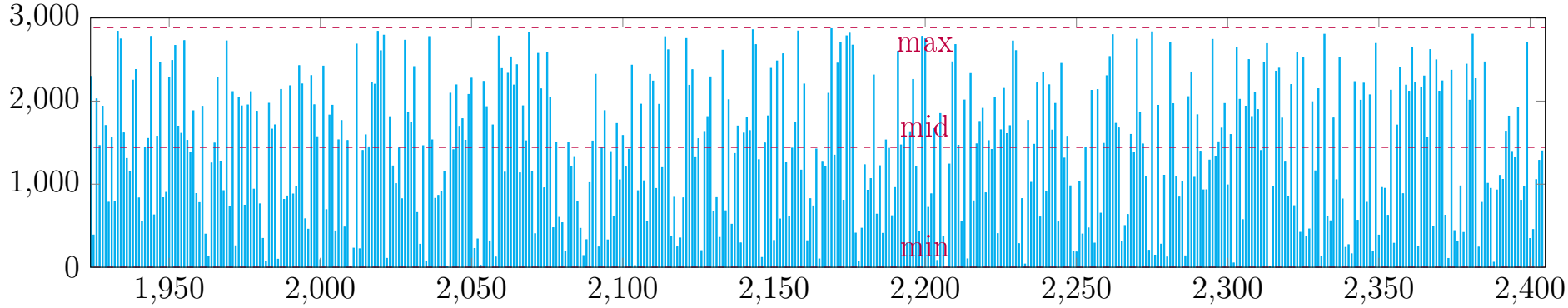

Рисунок 6 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1925 – 2405)

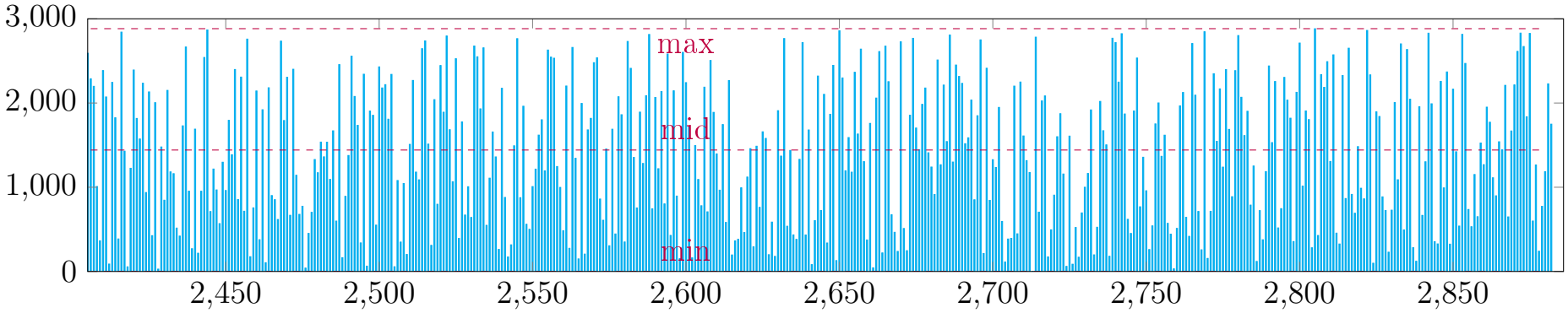

Рисунок 7 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи от 2046)

Приложение Б

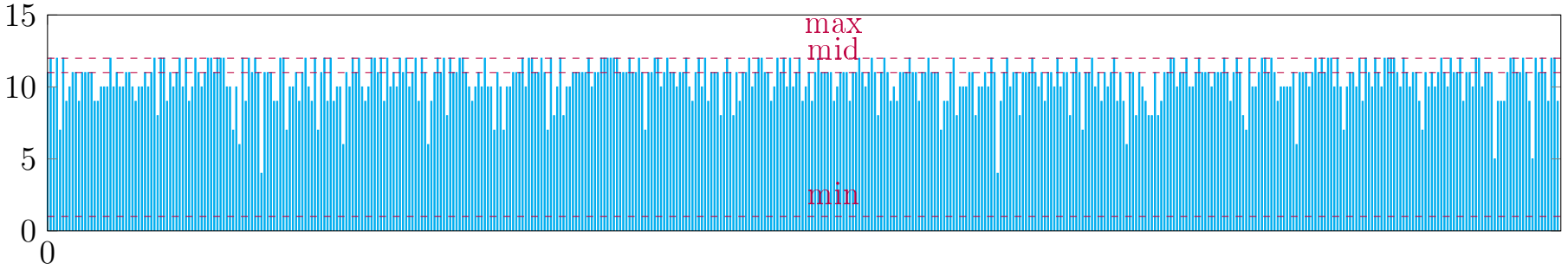

Рисунок 2 – Гистограмма количества сравнений на каждый ключ (ключи 0 – 481)

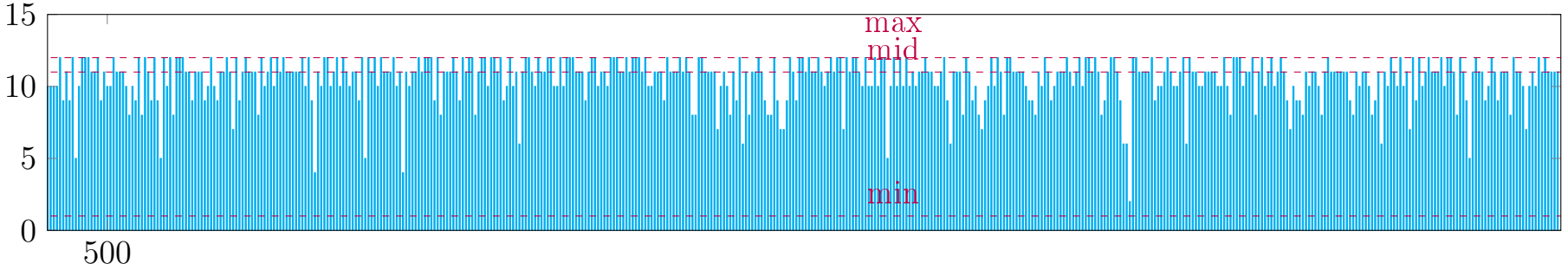

Рисунок 3 – Гистограмма количества сравнений на каждый ключ (ключи 482 – 962)

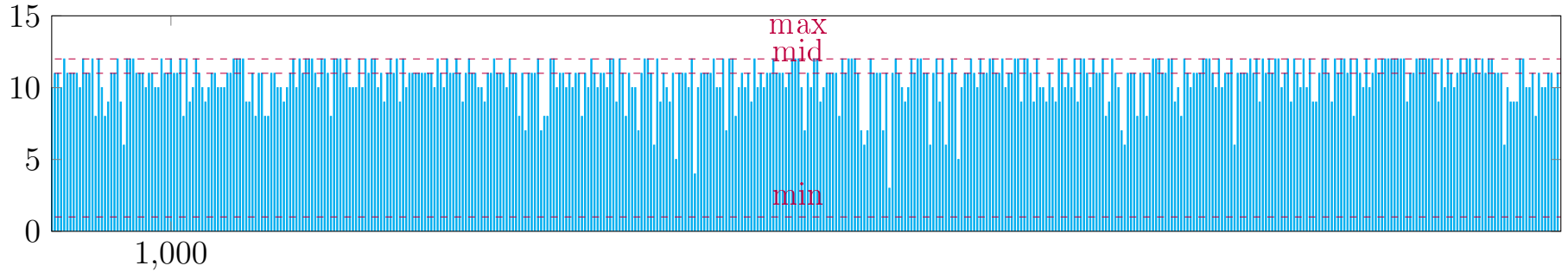

Рисунок 4 – Гистограмма количества сравнений на каждый ключ (ключи 963 – 1443)

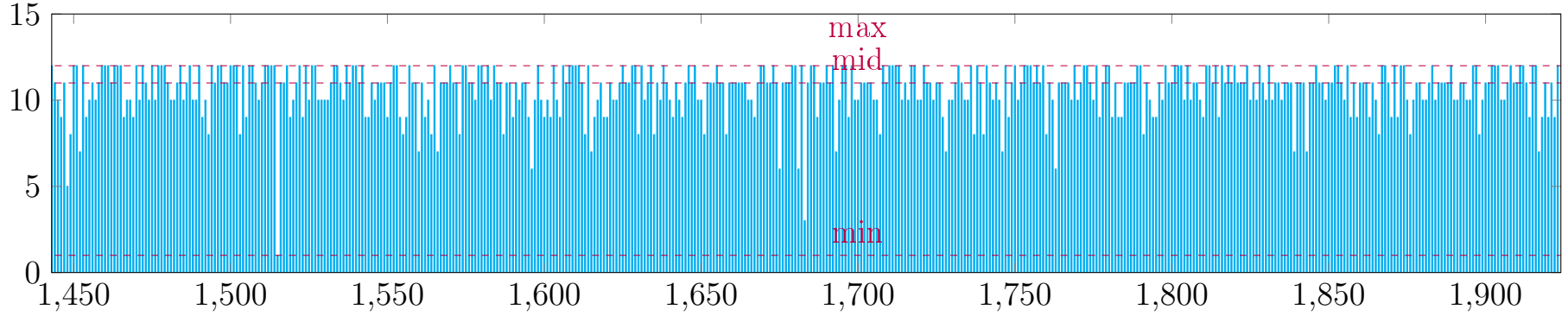

Рисунок 5 – Гистограмма количества сравнений на каждый ключ (ключи 1444 – 1924)

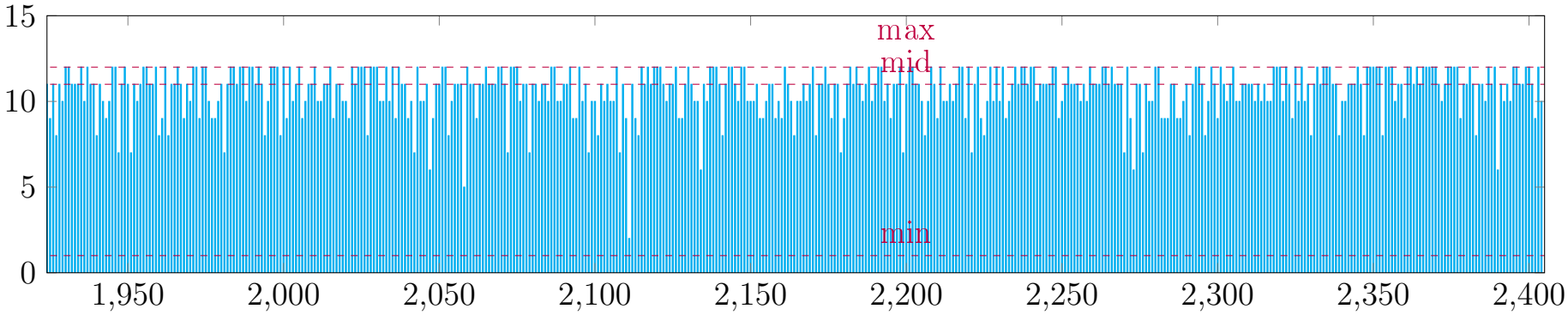

Рисунок 6 – Гистограмма количества сравнений на каждый ключ (ключи 1925 – 2405)

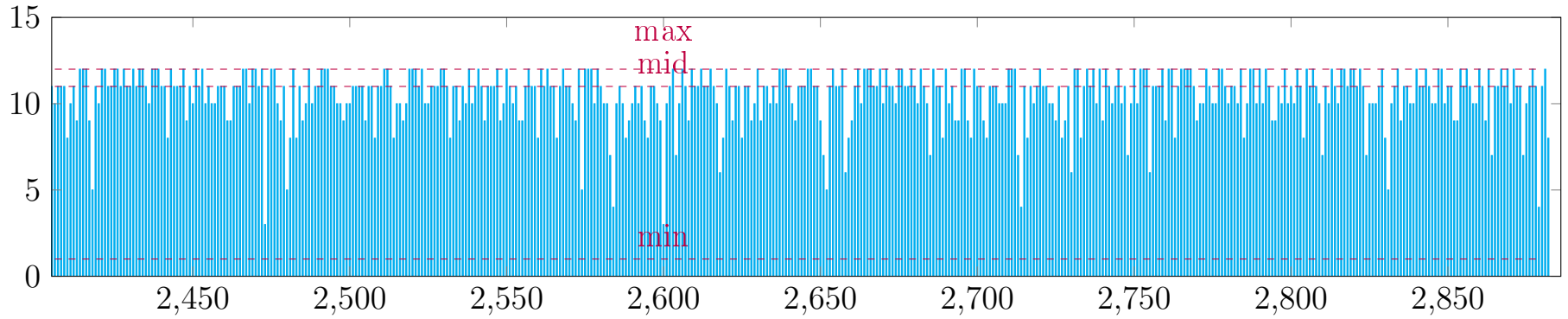

Рисунок 7 – Гистограмма количества сравнений на каждый ключ (ключи от 2946)

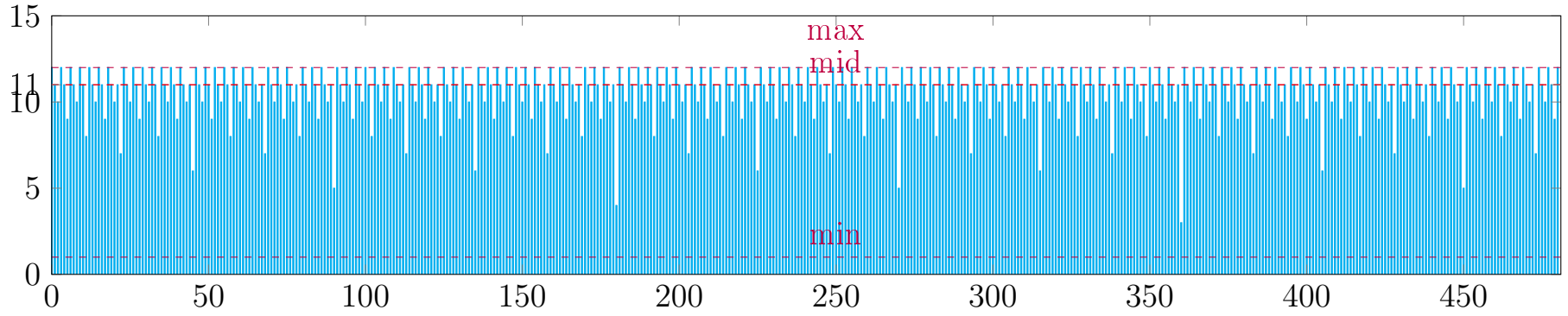

Рисунок 8 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 0 – 481)

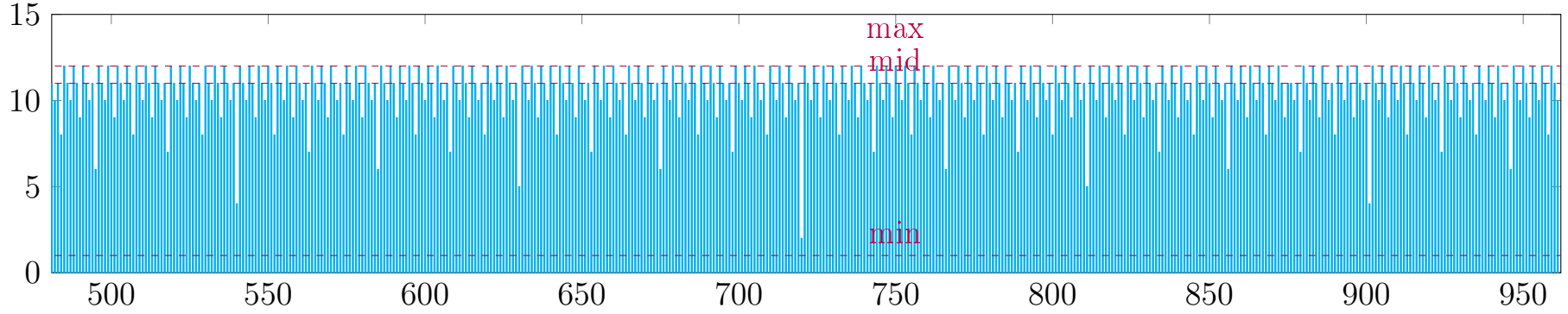

Рисунок 9 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 482 – 962)

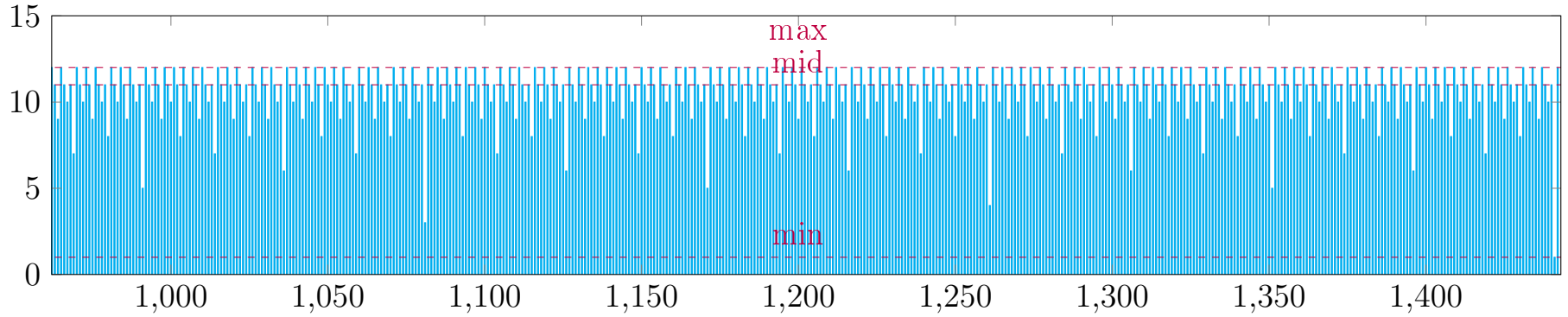

Рисунок 10 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 963 – 1443)

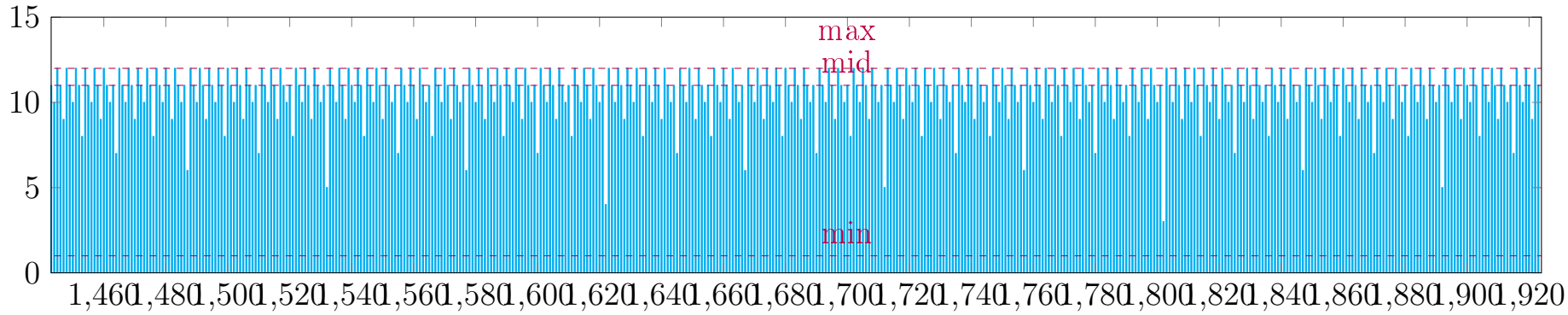

Рисунок 11 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1444 – 1924)

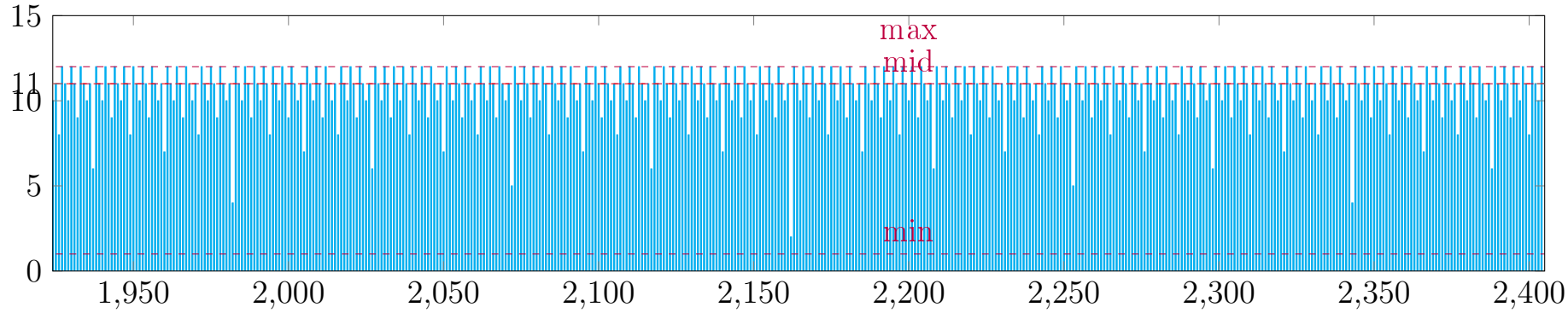

Рисунок 12 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1925 – 2405)

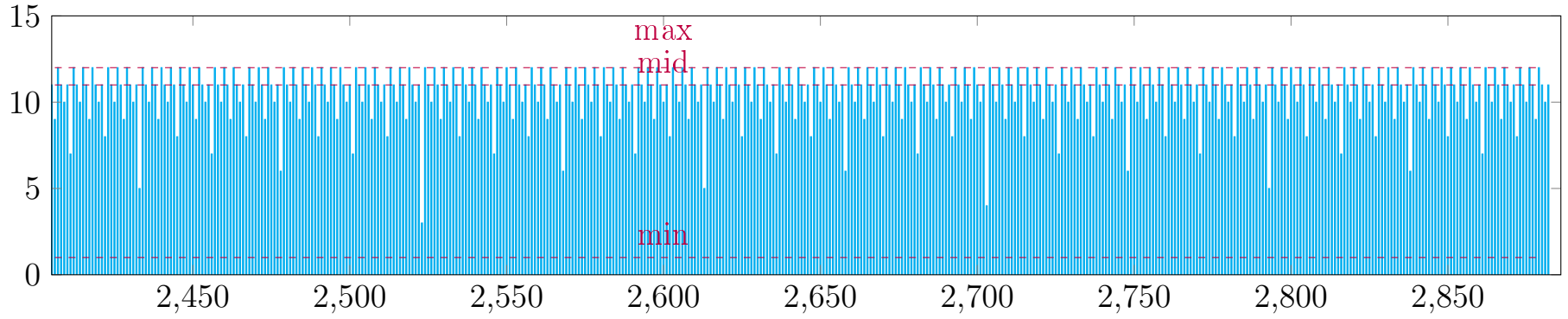

Рисунок 13 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи от 2046)

Приложение В

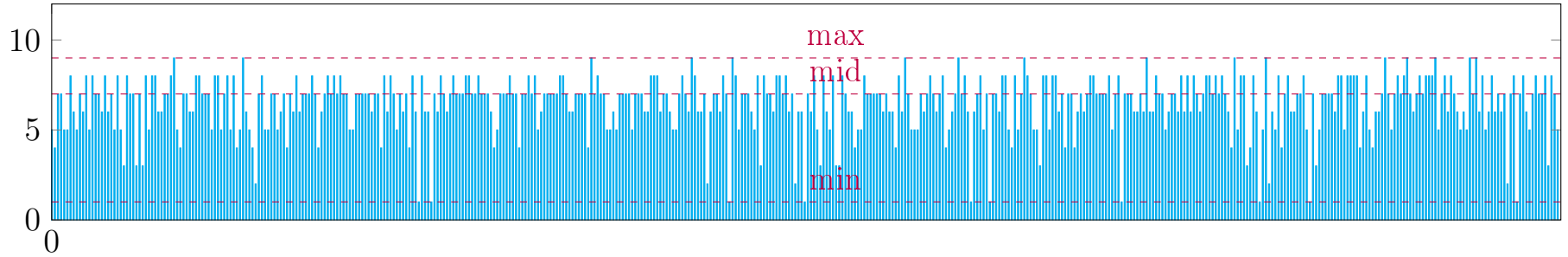

Рисунок 2 – Гистограмма количества сравнений на каждый ключ (ключи 0 – 481)

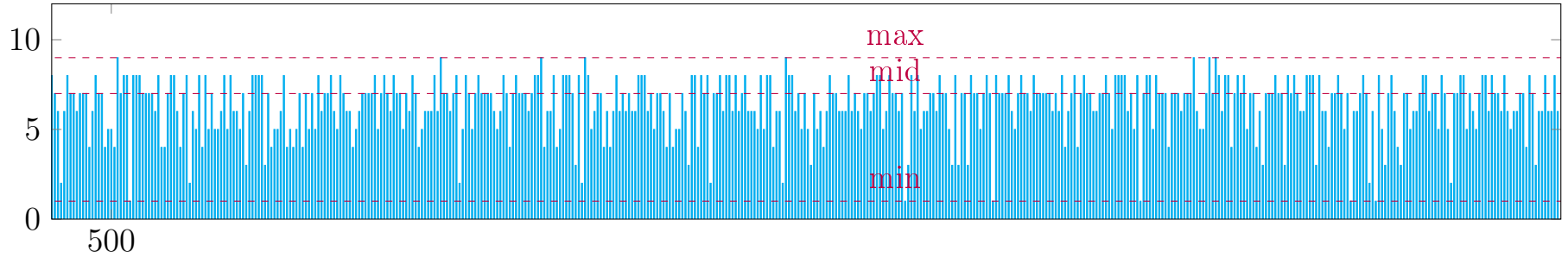

Рисунок 3 – Гистограмма количества сравнений на каждый ключ (ключи 482 – 962)

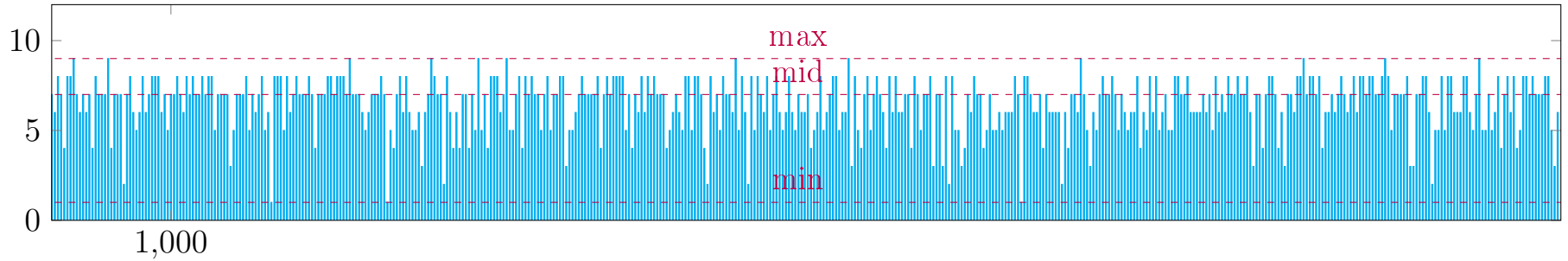

Рисунок 4 – Гистограмма количества сравнений на каждый ключ (ключи 963 – 1443)

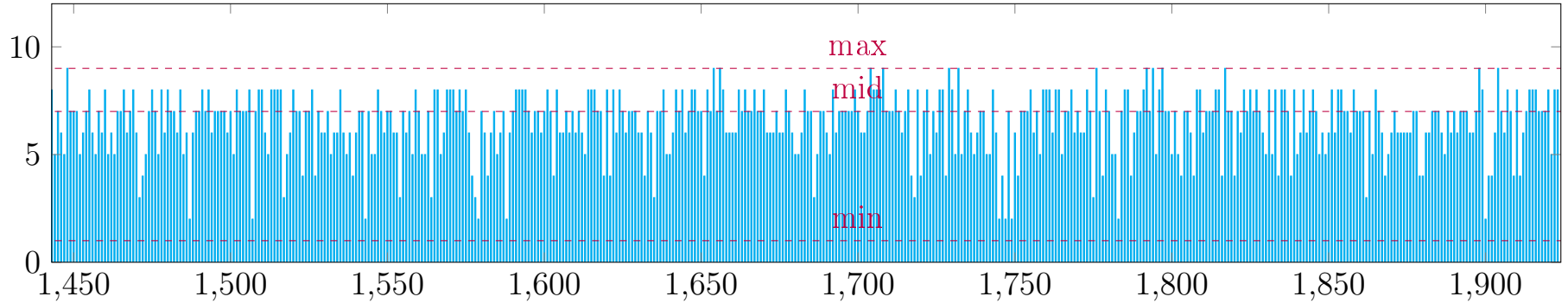

Рисунок 5 – Гистограмма количества сравнений на каждый ключ (ключи 1444 – 1924)

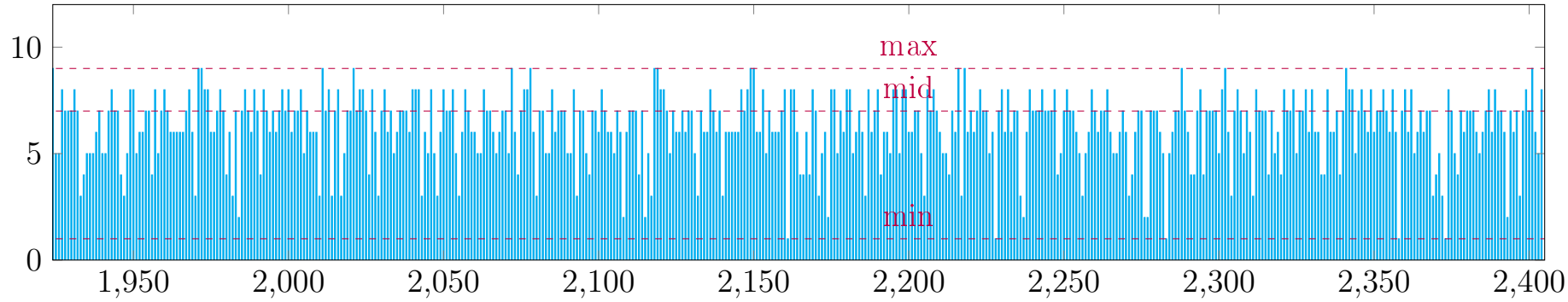

Рисунок 6 – Гистограмма количества сравнений на каждый ключ (ключи 1925 – 2405)

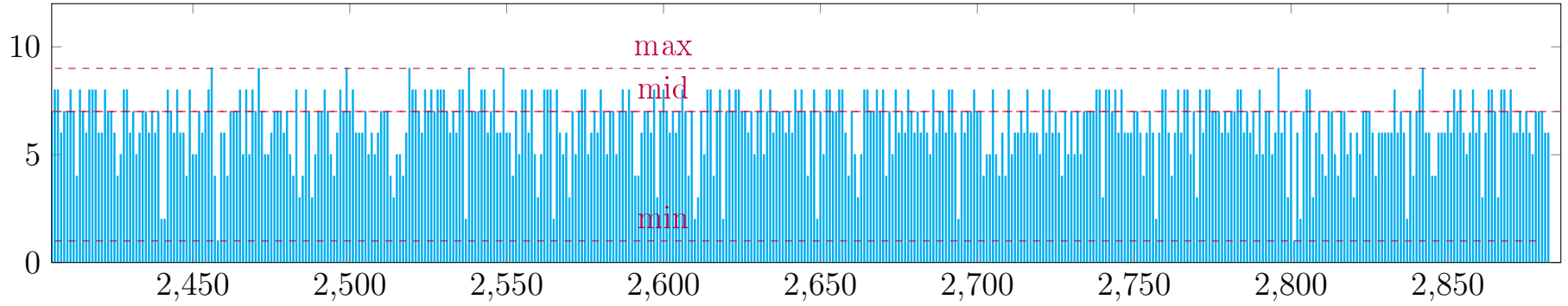

Рисунок 7 – Гистограмма количества сравнений на каждый ключ (ключи от 2046)

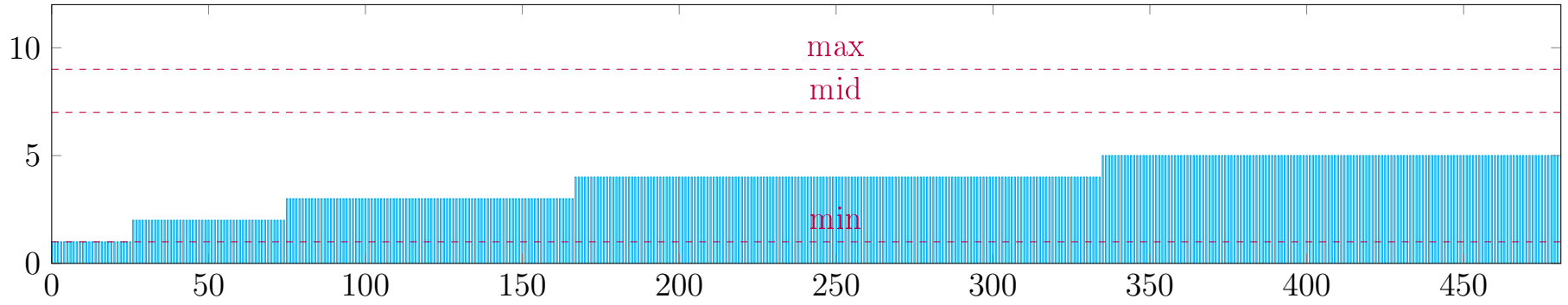

Рисунок 8 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 0 – 481)

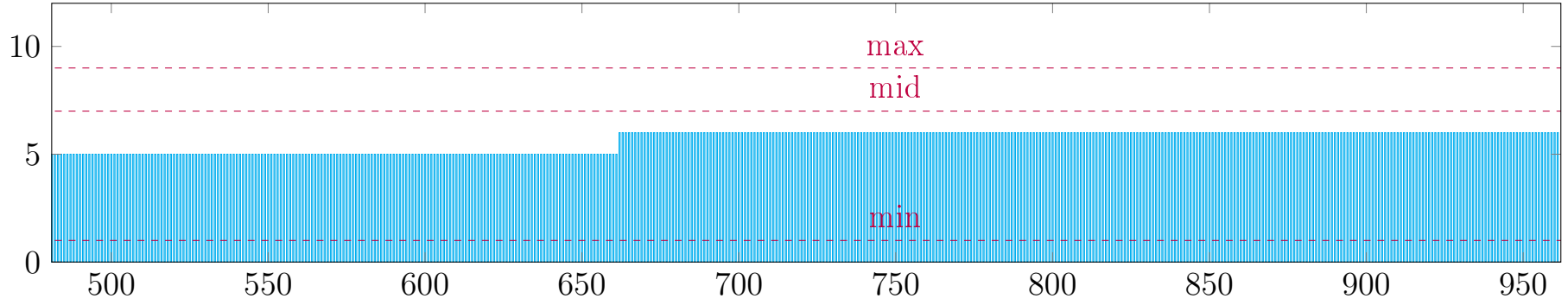

Рисунок 9 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 482 – 962)

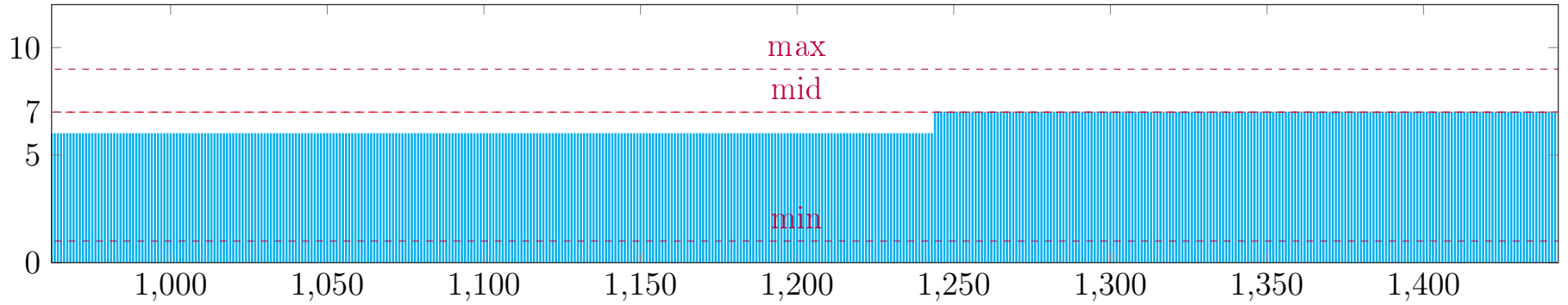

Рисунок 10 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 963 – 1443)

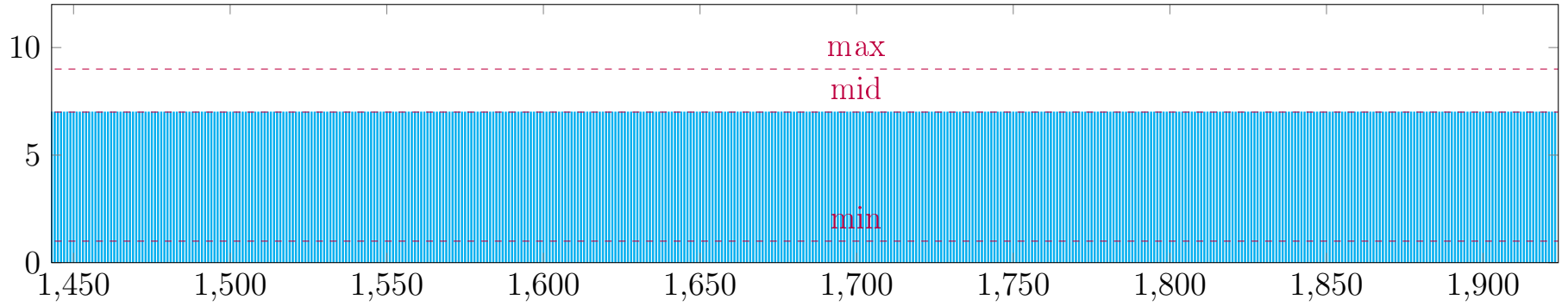

Рисунок 11 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1444 – 1924)

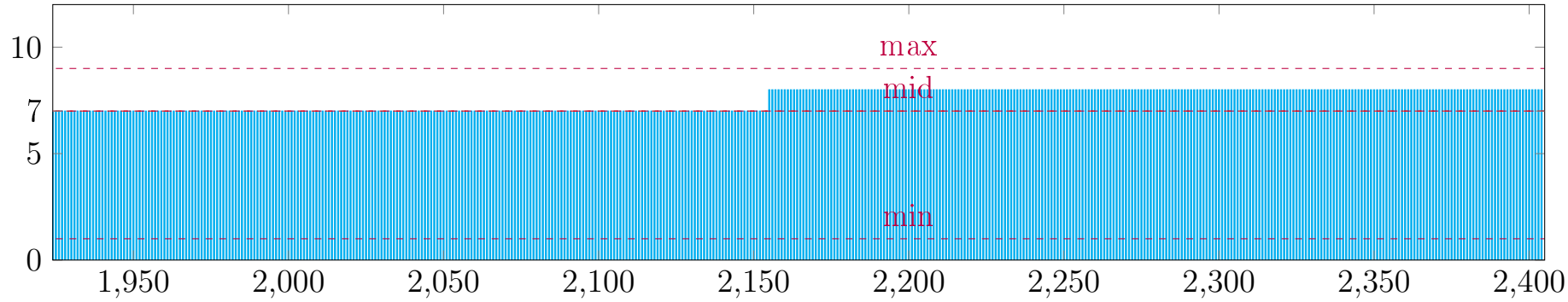

Рисунок 12 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи 1925 – 2405)

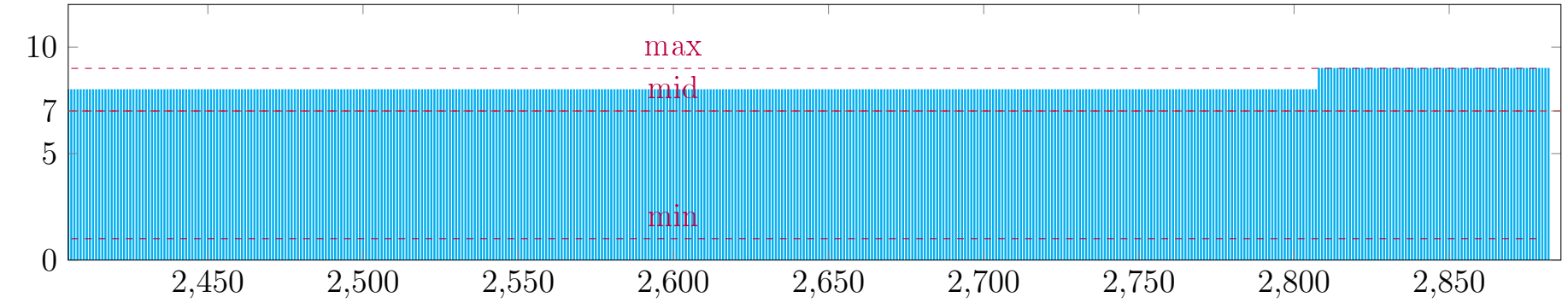

Рисунок 13 – Гистограмма количества сравнений на каждый ключ (отсортировано по алфавиту, ключи от 2046)