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	ИУ7-54Б		С. Д. Параскун
	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".

Словарь 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.

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 Постановка задачи

В данной лабораторной работе поиск в словаре реализовывается на примере словаря, основанного на тексте фанфика по фандому "Наруто". Ключом является номер слова в тексте, значением - соответственно само слово. Знаки препинания и прочие обозначения в тексте игнорируются.

1.2 Алгоритм полного перебора

Алгоритм полного перебора для любой задачи часто является самым примитивным и самым трудоемким алгоритмом, и в рамках рассматриваемой задачи этот случай не становится исключением. Для поставленной задачи алгоритм полного перебора заключается в последовательном проходе по словарю до тех пор, пока не будет найден требуемый ключ. Очевидно, что худшим случаем является ситуация, когда необходимый ключ находится в конце словаря либо когда этот ключ вовсе не представлен в словаре. Трудоемкость алгоритма зависит от положения ключа в словаре - чем дальше он от начала словаря, тем больше единиц процессорного времени потребуется на поиск.

1.3 Алгоритм бинарного поиска

Алгоритм бинарного поиска применяется к заранее остортированному набору значений. В рамках поставленной задачи словарь должен быть остортирован по ключам по возрастанию. [1]

Основная идея бинарного поиска для поставленной задачи заключается в следующем:

1. Определение значения ключа в середине словаря. Полученное значение сравнивается с искомым ключом;

2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй;

3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом;

4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска.

Как известно, сложность алгоритма бинарного поиска на заранее упорядоченном наборе данных составляет $O(log2(n))$, где n - размер словаря.

Однако, может возникнуть ситуация, что затраты на сортировку данных будут нивелировать преимущество быстрого поиска при больших размерностях массивов данных.

1.4 Алгоритм частного анализа

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 [2]. При таком разбиении повышается вероятность того, что позиция искомого слова будет ближе к началу словаря, что значительно ускорит поиск.

Таким образом, в начале алгоритм получает словарь и ключ для поиска. Сначала происходит обращение к наиболее частому сегменту, а в нем ищется совпадение ключа с помощью двоичного поиска. Если ключ не найден в первом сегменте, происходит обращение к следующему сегменту с последующим бинарным поиском до тех пор, пока не закончатся сегменты. Если ключ не найден ни в одном сегменте, то ключ не представлен в данном словаре и возникает ошибка доступа.

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.

1.6 Требования к ПО

К программе предъявляются следующие требования:

1. Программа должна предоставлять пользовательский интерфейс;
2. Принципом разбиения на сегменты является анализ первой буквы слова;
3. Пустой словарь - ошибочная ситуация;
4. Программа не проверяет корректность входных файлов;
5. Программа должна производить логирование для каждого из рассмотренных алгоритмов - создаются файлы с соответствующим названием, содержащие строки вида (номер ключа; количество сравнений во время поиска).

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 структуры ПО

Программа будет включать в себя один смысловой модуль, называемый find, который содержит в себе процедуры и функции, связанные с реализацией алгоритмов поиска в словаре по ключу. Независимо от модуля будет существовать файл main.go, реализующий пользовательский интерфейс. Также реализована программа на языке python, которая формирует данные для словаря.

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.

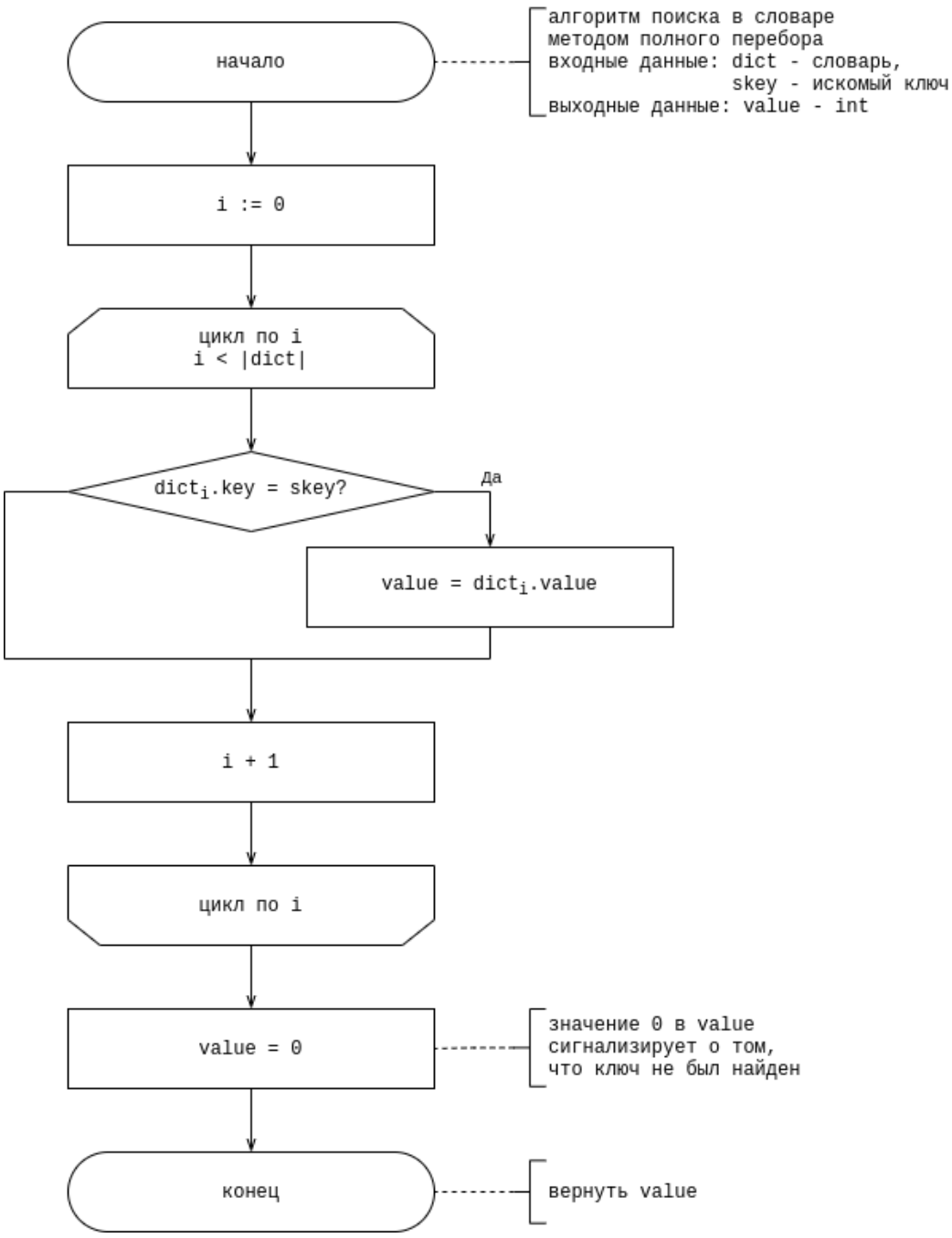

Рисунок 2.1 – Схема работы алгоритма полного перебора

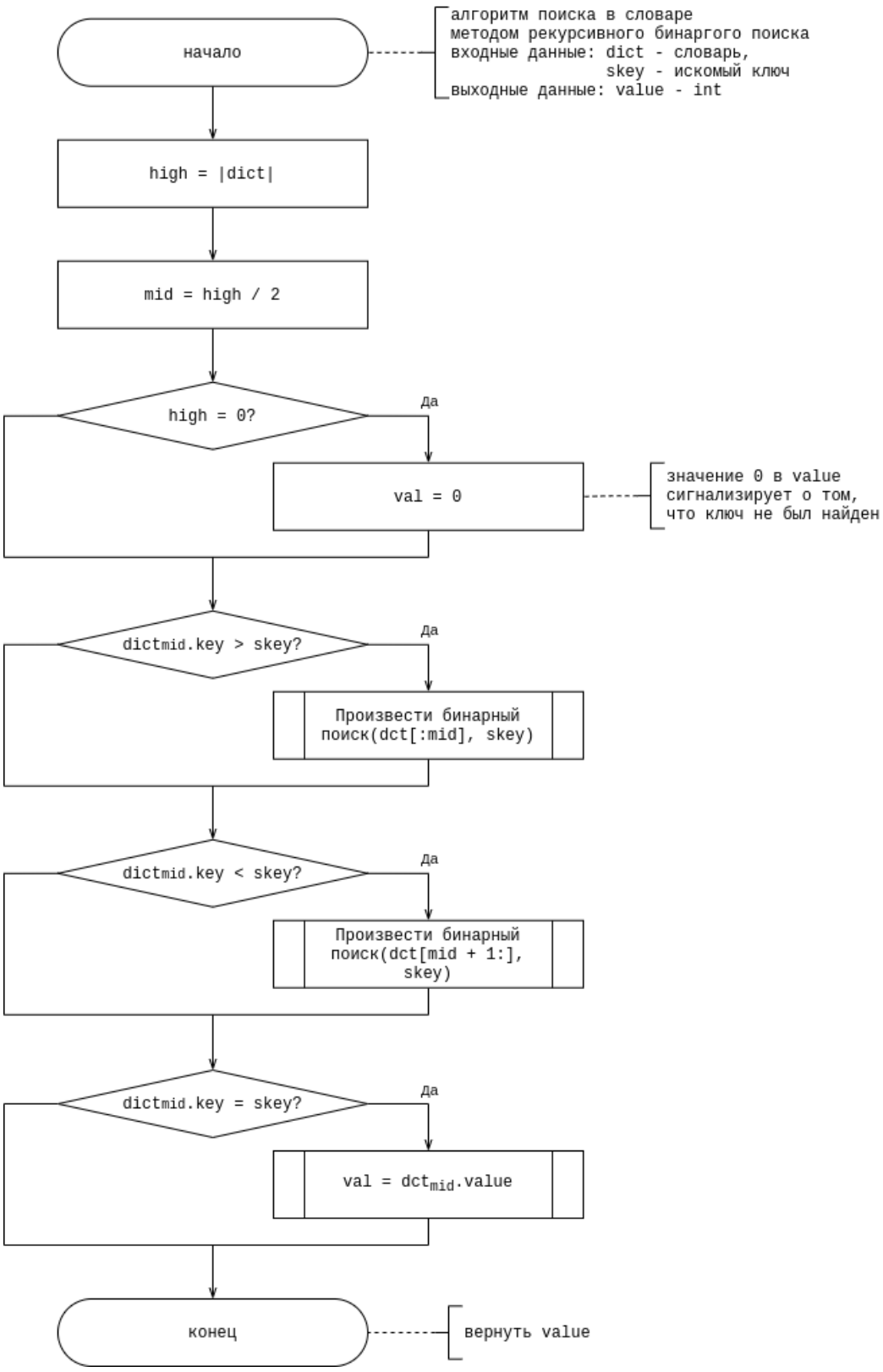

Рисунок 2.2 – Схема работы алгоритма бинарного поиска

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

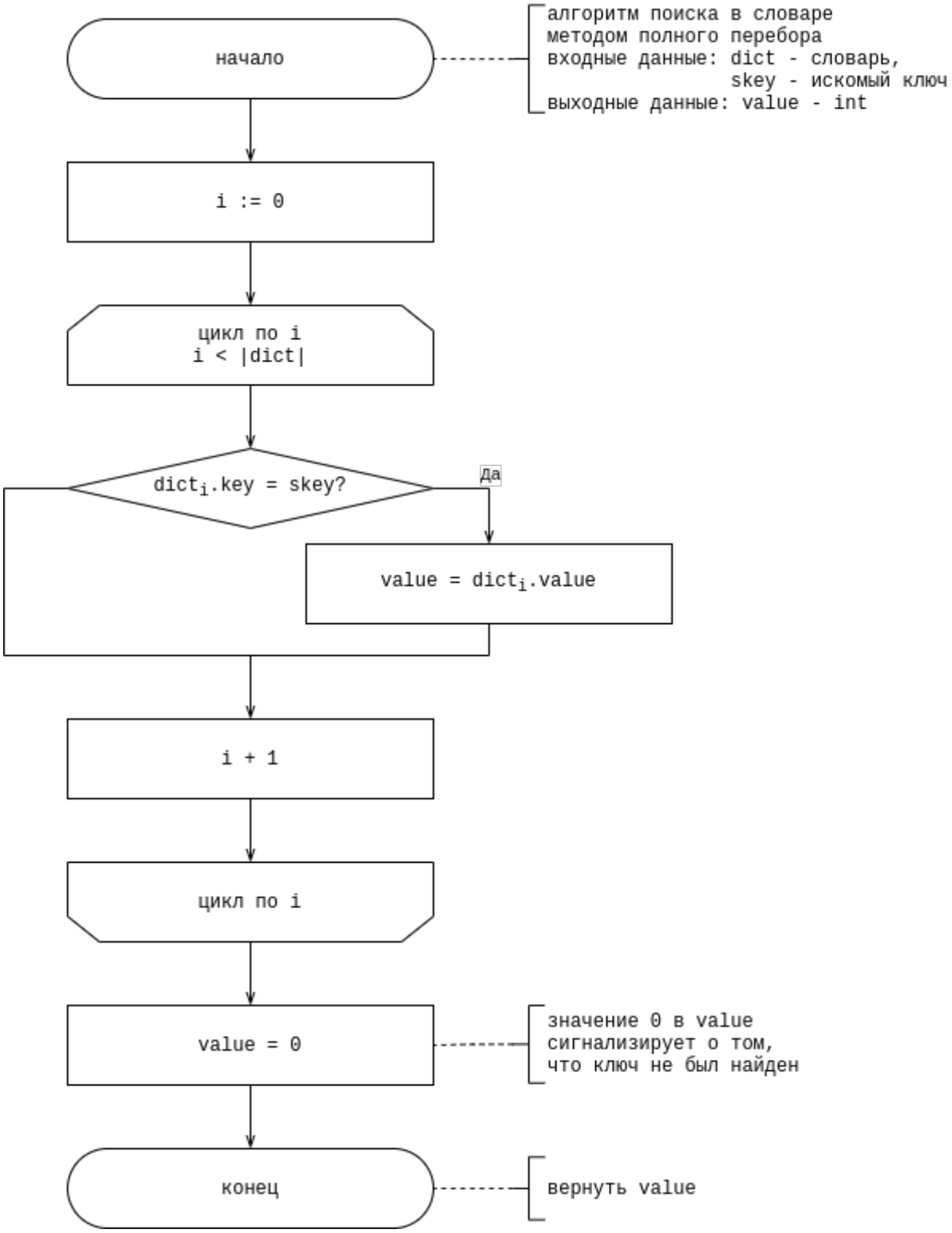

Рисунок 2.3 – Схема работы алгоритма бинарного поиска для словаря, поделенного на сегменты

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 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 Средства реализации

Для реализации ПО был выбран компилируемый многопоточный язык программирования Golang[3], поскольку язык отличается автоматическим управлением памяти. В качестве среды разработки была выбрана среда VS Code.

3.2 Листинги кода

На листинге 3.1 представлены используемые типы данных для реализации алгоритмов поиска

Листинг 3.1 – Структуры данных

```
type Dict_t map[int]string
type Segment_t Dict_t
```

Листинг 3.2 демонстрирует реализацию алгоритма полного перебора.

Листинг 3.2 – Полный перебор

```
    func BruteForce(dict Dict_t, key int) string {
    for i := 0; i < len(dict); i++ {
    if key - 1 == i {
    return dict[i]
    }
    }
    return "ERROR:NOT_FOUND"
    }
```

На листинге 3.3 представлена функция бинарного поиска:

Листинг 3.3 – Бинарный поиск

```
    func BinarySearch(dict Dict_t, key int) string {
    var (left int
    right int
    mid int)

    left = 0
    right = len(dict) - 1

    if right != 0 {
    for left <= right {
    mid = left + (right - left) / 2
    if mid == key - 1 {
    return dict[mid]
    }
    if mid < key - 1 {
    left = mid + 1
    } else {
    right = mid - 1
    }
    }
    }
    return "ERROR:NOT_FOUND"
    }
```

На листинге 3.4 представлены вспомогательные функции для реализации алгоритма частотного анализа. В то время как на листинге 3.5 представлена сама реализация алгоритма частотного анализа.

Листинг 3.4 – Вспомогательные функции для реализация алгоритма частотного анализа

```
    var alphabet = string("abcdefghijklmnopqrstuvwxyz")
    var mask = make([]int, len(alphabet))

    func sort_keys(keys []int) {
    for i := 0; i < len(keys)-1; i++ {
    for j := 0; j < len(keys)-i-1; j++ {
    if keys[j] > keys[j+1] {
    swap_in_int_arr(keys, j, j+1)
    }
    }
    }
    }

    func init_mask() {
    for i := 0; i < len(alphabet); i++ {
    mask[i] = i
    }
    }

    func create_segments(size int) []Segment_t {
    segments := make([]Segment_t, size)
    for i := 0; i < size; i++ {
    segments[i] = make(Segment_t)
    }
    return segments
    }
```

Листинг 3.5 – Реализация алгоритма частотного анализа

```
    func divide_by_first_letter(segments []Segment_t, dict Dict_t) {
    for i := 0; i < len(dict); i++ {
    index := find_pos(strings.ToLower(string(dict[i][0])))
    segments[index][i] = dict[i]
    }
    }

    func sort_by_size(segments []Segment_t) {
    for i := 0; i < len(segments)-1; i++ {
    for j := 0; j < len(segments)-i-1; j++ {
    if len(segments[j]) < len(segments[j+1]) {
    swap_segs(segments, j, j+1)
    swap_in_int_arr(mask, j, j+1)
    }
    }
    }
    }

    func binary_search(segments []Segment_t, key int) string {
    var (left  int
    right int
    mid   int)

    for i := 0; i < len(segments); i++ {
    var keys = make([]int, 0)
    for key := range segments[i] {
    keys = append(keys, key)
    }
    sort_keys(keys)

    left = 0
    right = len(segments[i]) - 1

    if right != 0 {
    for left <= right {
    mid = left + (right-left)/2
    if keys[mid] == key {
    return segments[i][keys[mid]]
    }
    if keys[mid] < key {
    left = mid + 1
    } else {
    right = mid - 1
    }
    }
    }
    }
    return "ERROR:NOT_FOUND"
    }

    func FreqFind(dict Dict_t, key int) string {
    init_mask()
    segments := create_segments(len(alphabet))
    divide_by_first_letter(segments, dict)
    sort_by_size(segments)
    return binary_search(segments, key-1)
    }
```

3.3 Тестирование ПО

Результаты тестирования ПО приведены в таблице 3.1, в которой используются следующие обозначения:

1. ПП - алгоритм полного перебора;
2. БП - алгоритм бинарного поиска;
3. ЧА - алгоритм частотного анализа;

Таблица 3.1 – Тестирование ПО

Исходный словарь	Ключ	ПП	БП	ЧА	Ожидание
1: 'I', 2:'want', 3:'sleep'	1	'I'	'I'	'I'	'I'
1: 'I', 2:'want', 3:'sleep'	2	'want'	'want'	'want'	'want'
1: 'I', 2:'want', 3:'sleep', 4:'sleep'	3	'sleep'	'sleep'	'sleep'	'sleep'
1: 'I', 2:'want', 3:'sleep', 2:'want'	3	'want'	'want'	'want'	'want'

3.4 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; [4]
- память 8 GiB;
- процессор AMD Ryzen 5 3500u.[5]

4.2 Постановка эксперимента

Эксперимент заключается в генерации гистограмм на основе ключей и соответствующих им количеств сравнений для поиска ключа. Для каждого из алгоритмов строятся гистограммы 2 видов: для гистограммы первого типа отсортировать ключи по алфавиту, а для второй - по количеству сравнений. По полученным гистограммам необходимо сделать выводы об эффективности алгоритмов.Необходимо также отметить среднее, минимальное и максимальное количество сравнений. 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

На рисунках 4.2 - 4.7 представлены результаты эксперимента. По оси X откладываются значения ключей, для оси У - количество сравнений. Зеленой горизонтальной линеей отмечено минимальное количество сравнений, желтой - среднее, красной - максимальное.

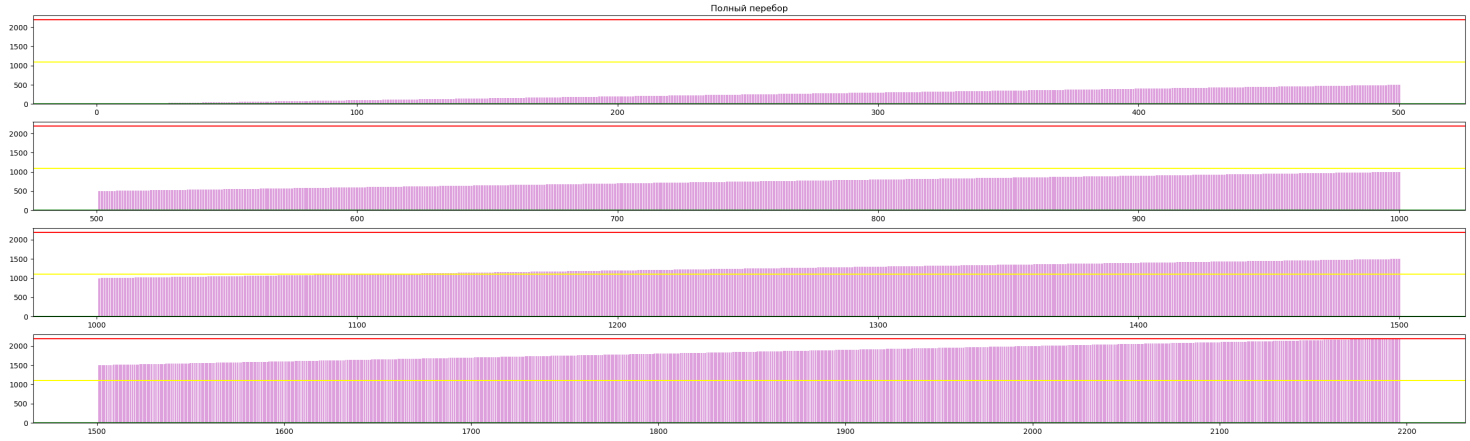

Рисунок 4.1 – Полный перебор

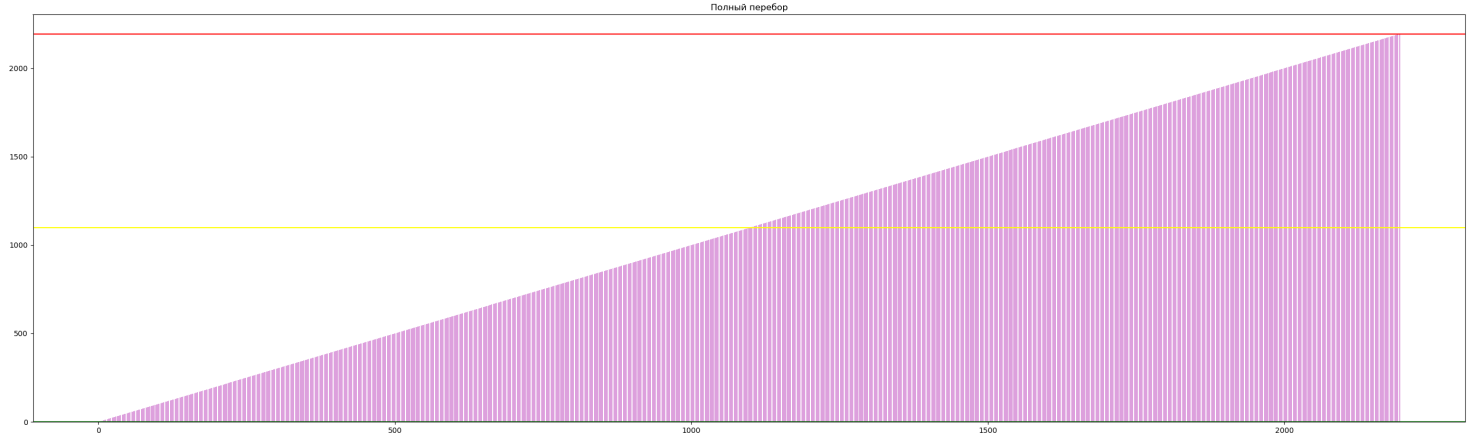

Рисунок 4.2 – Полный перебор

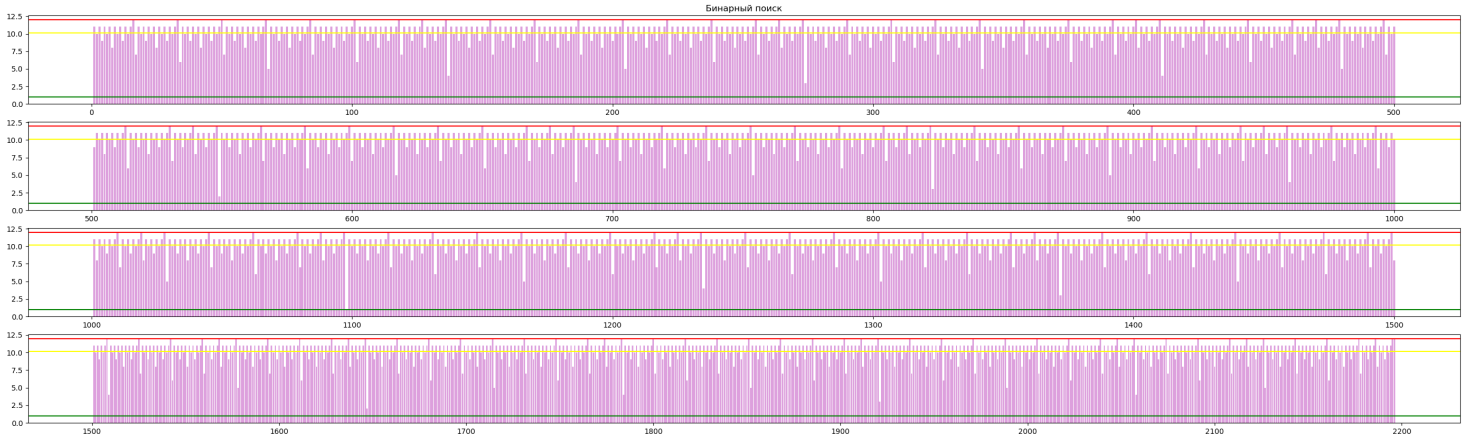

Рисунок 4.3 – Бинарный поиск

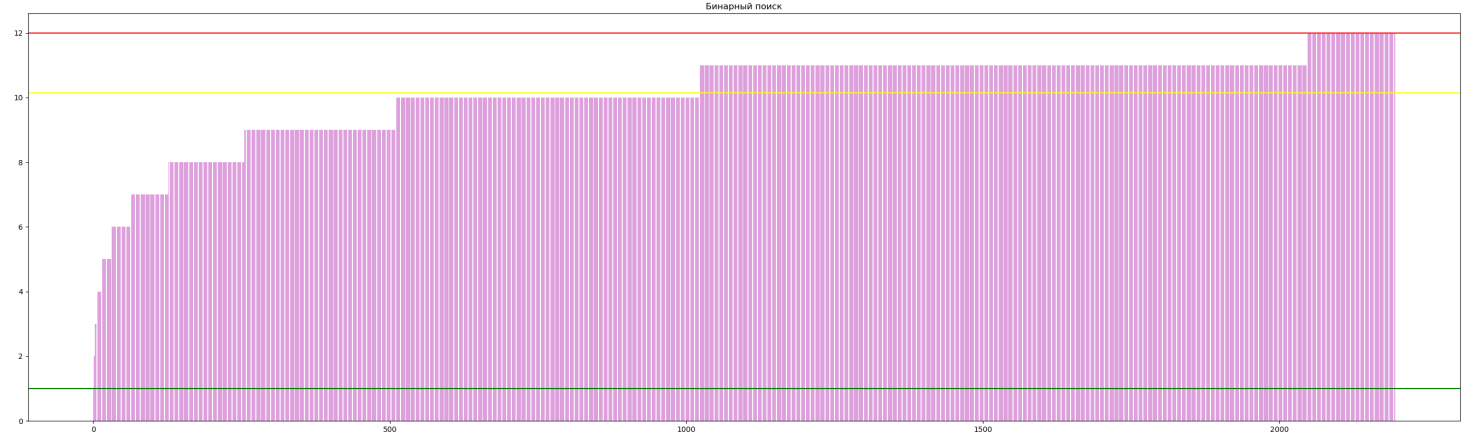

Рисунок 4.4 – Бинарный поиск

Рисунок 4.5 – Бинарный поиск

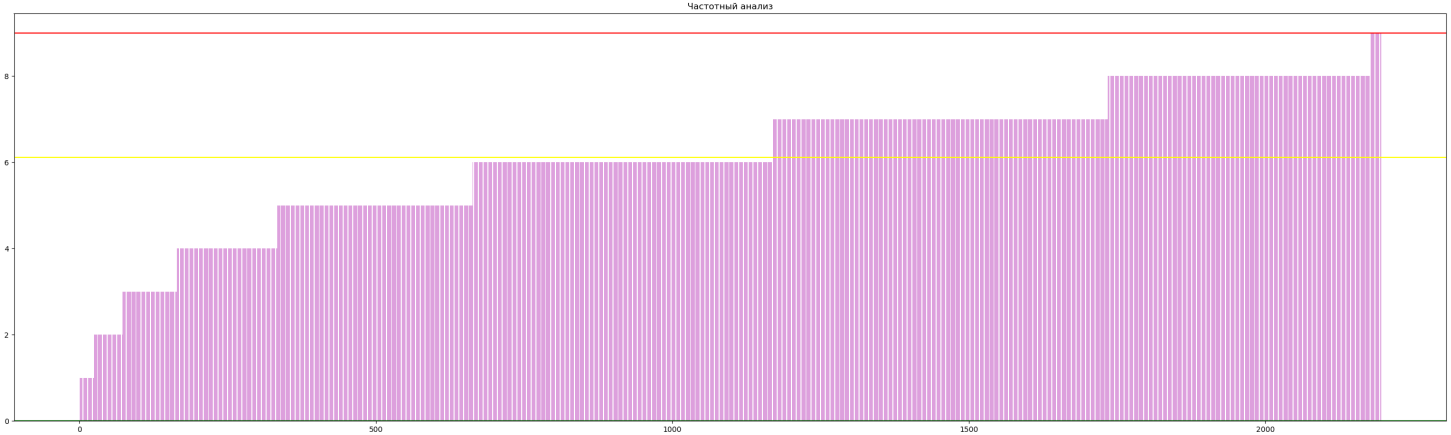

Рисунок 4.6 – Бинарный поиск

4.4 Вывод

Как видно из графиков, частотный анализ требует в среднем требует в 1.5-2 раза меньше сравнений. Как и ожидалось, алгоритм полного перебора требует наибольшего количества сравнений из всех представленных алгоритмов. Также видно, что гистограмма типа 1 для полного перебора - возрастающая последовательность, в то время как для алгоритма бинарного поиска и, в частности, алгоритма частотного анализа, последовательность не является монотонно возрастающей в виду идеи работы алгоритма бинарного поиска.

Заключение

Как видно из графиков, частотный анализ требует в среднем требует в 1.5-2 раза меньше сравнений. Как и ожидалось, алгоритм полного перебора требует наибольшего количества сравнений из всех представленных алгоритмов. Также видно, что гистограмма типа 1 для полного перебора - возрастающая последовательность, в то время как для алгоритма бинарного поиска и, в частности, алгоритма частотного анализа, последовательность не является монотонно возрастающей в виду идеи работы алгоритма бинарного поиска.

На основании проделанной работы можно сделать следующий вывод: чем больше размер словаря, тем более рационально использовать эффективные алгоритмы поиска. Однако стоит помнить, что бинарный поиск применим только к отсортированным данным, поэтому может возникнуть ситуация, когда поддержание упорядоченной структуры занимает нерационально большое время и преимущества от бинарного поиска нивелируются.

Список литературы

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

[2] English Letter Frequency Counts. URL: http://norvig.com/mayzner.html.

[3] Go. URL: https://go.dev/.

[4] Linux – Getting Started [Электронный ресурс]. Режим доступа: `https://linux.org`. Дата обращения: 14.09.2021.

[5] Процессор AMD Ryzen 5 3500U. URL: https://www.amd.com/ru/products/apu/amd-ryzen-5-3500u.