



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 по курсу "Анализ алгоритмов"

Тема Решение задачи поиска в словаре

Студент Леонов В.В.

Группа ИУ7-56Б

Оценка (баллы)

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение	3
1 Аналитическая часть	4
1.1 Постановка задачи	4
1.2 Алгоритм полного перебора	4
1.3 Бинарный поиск	5
1.4 Алгоритм с сегментированием словаря	5
1.5 Вывод	6
2 Конструкторская часть	7
2.1 Схемы алгоритмов	7
2.2 Описание структуры ПО	10
2.3 Вывод	10
3 Технологическая часть	11
3.1 Средства реализации	11
3.2 Листинг кода	11
3.3 Вывод	14
4 Исследовательская часть	15
4.1 Пример работы	15
4.2 Технические характеристики	15
4.3 Оценка эффективности алгоритмов	16
4.4 Вывод	18
Заключение	19
Список литературы	20

Введение

С непрерывным ростом количества доступной текстовой информации появляется потребность определенной организации ее хранения, удобного 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 [1].

Целью данной лабораторной работы являются изучение проблемы поиска записи в словаре и реализация алгоритмов, решающих данную задачу: алгоритм полного перебора, алгоритм бинарного поиска и алгоритм частотного анализа.

Для достижения указанной выше цели следует выполнить следующие задачи:

- изучить задачу поиска записи в словаре, а также идеи ее решения с помощью алгоритма полного перебора, алгоритма бинарного поиска и алгоритма частотного анализа;
- привести схемы изученных алгоритмов;
- описать используемые типы данных;
- описать структуру разрабатываемого ПО;
- реализовать изученные алгоритмы;
- протестировать разработанное ПО;
- выполнить на основе экспериментальных данных сравнительный анализ временных характеристик каждого из алгоритмов в зависимости от позиции записи в словаре;
- определить лучший и худший случай для каждого из алгоритмов;
- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе описаны основные идеи рассматриваемых алгоритмов и выполняется формальная постановка задачи, на которой будут исследоваться данные алгоритмы.

1.1 Постановка задачи

В данной лабораторной работе поиск в словаре реализовывается на примере словаря, основанного на тексте песен британской рок-группы Bring Me The Horizon. Ключом является непосредственно слово, значением - количество раз, сколько данное слово встречается в текстах песен. 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. Средняя трудоёмкость может быть рассчитана как математическое ожидание по формуле 1.1, где Ω – множество всех возможных случаев, k - кол-во единиц процессорного времени, затрачиваемого на одну операцию сравнения в словаре.

$$\begin{aligned}
\sum_{i \in \Omega} p_i \cdot f_i &= k \cdot \frac{1}{N+1} + 2 \cdot k \cdot \frac{1}{N+1} + 3 \cdot k \cdot \frac{1}{N+1} + \dots + N \cdot k \cdot \frac{1}{N+1} = \\
&= k \cdot \frac{1 + 2 + 3 + \dots + N}{N+1} = \frac{k \cdot N}{2}
\end{aligned}
\tag{1.1}$$

1.3 Бинарный поиск

Алгоритм бинарного поиска применяется к заранее отсортированному набору значений. В рамках поставленной задачи словарь должен быть отсортирован по ключам по возрастанию. Основная идея бинарного поиска для поставленной задачи заключается в следующем:

- определение значения ключа в середине словаря. Полученное значение сравнивается с искомым ключом;
- если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй;
- поиск сводится к тому, что вновь определяется значение срединного элемента в выбранной половине и сравнивается с ключом;
- 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 $O(\log_2(n))$, 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 с сегментированием словаря

Суть алгоритма заключается в разбиении исходного множества пар (ключ; значение) на некоторые т.н. сегменты по заранее заданному общему

признаку. Затем сегменты сортируются по их размеру, таким образом, сегмент, который содержит в себе больше всего вхождений, будет расположен первым, т.е. доступ к нему будет осуществляться быстрее, затем второй и так далее по убыванию.

Таким образом, на скорость доступа к сегменту влияет частота вхождений пар значений из словаря. В каждом сегменте пары (ключ; значение) сортируются по возрастанию ключей для дальнейшего применения бинарного поиска в конкретном сегменте.

Средняя трудоёмкость при множестве всех возможных случаев Ω может быть рассчитана по формуле 1.2.

$$\sum_{i \in \Omega} (f_{\text{выбор сегмента } i\text{-ого элемента}} + f_{\text{бинарный поиск } i\text{-ого элемента}}) \cdot p_i \quad (1.2)$$

1.5 Вывод

Для ПО, решающего поставленную задачу следует выделить следующие требования:

- текст песен ВМТН для наполнения словаря хранится в файле `bmth.txt`;
- ПО не проверяет корректность указанных файлов;
- ПО имеет подсказки ввода;
- ПО должно обеспечить методы логирования полученных экспериментальных данных;
- ПО выводит значение, хранимое под ключом в словаре, которое вводит пользователь с клавиатуры.

В данном разделе были рассмотрены три алгоритма решения задачи поиска в словаре: алгоритм полного перебора, алгоритм бинарного поиска и алгоритм частотного анализа. Для каждого из алгоритмов были выделены основные идеи решения поставленной задачи, а также определена асимптотическая сложность.

2 Конструкторская часть

В данном разделе представлены схемы алгоритмов, описанных в аналитическом разделе.

2.1 Схемы алгоритмов

На рисунке 2.1 приведена схема алгоритма полного перебора.

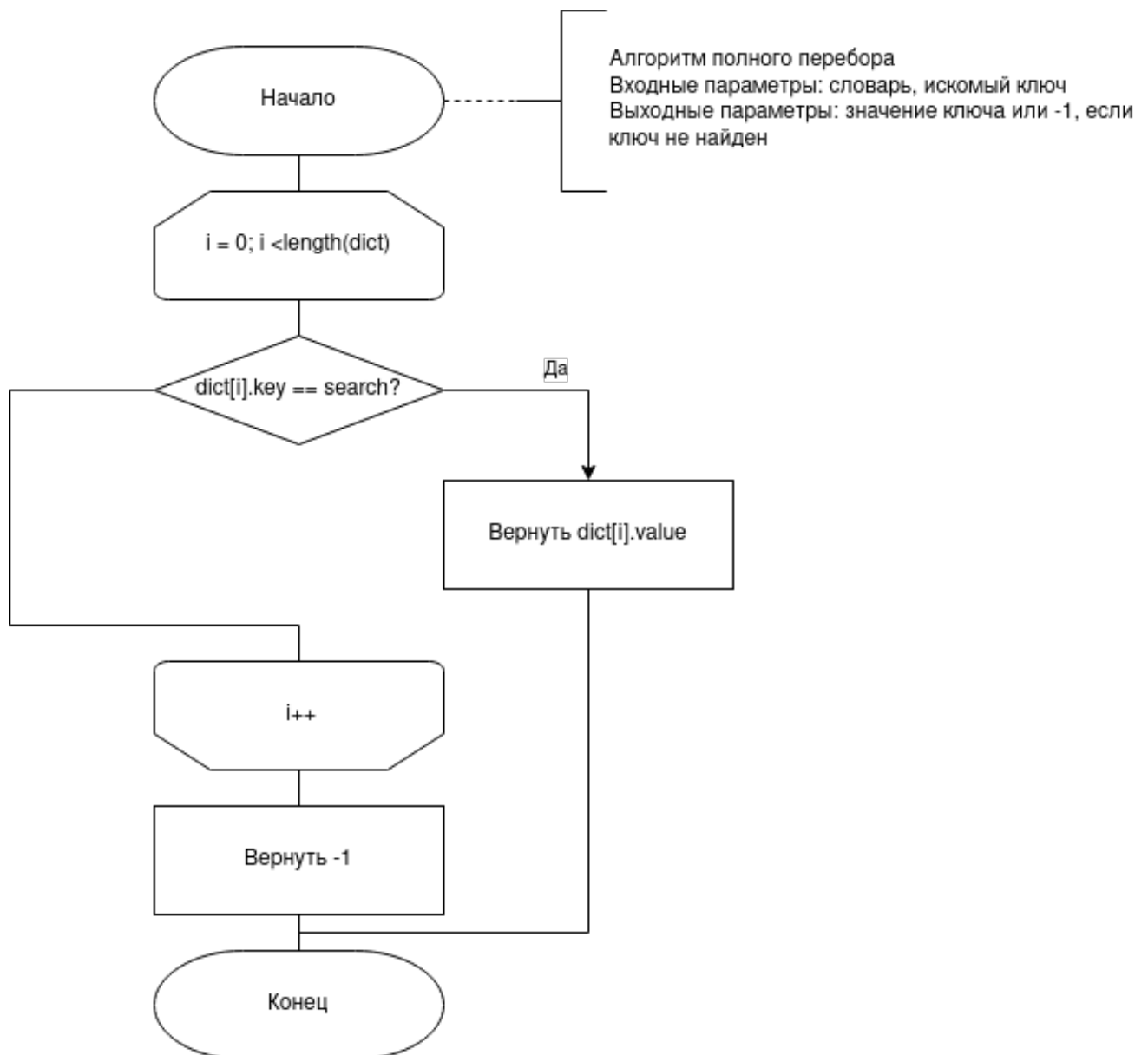


Рисунок 2.1 – Схема алгоритма полного перебора

На рисунке 2.2 приведена схема алгоритма двоичного поиска.

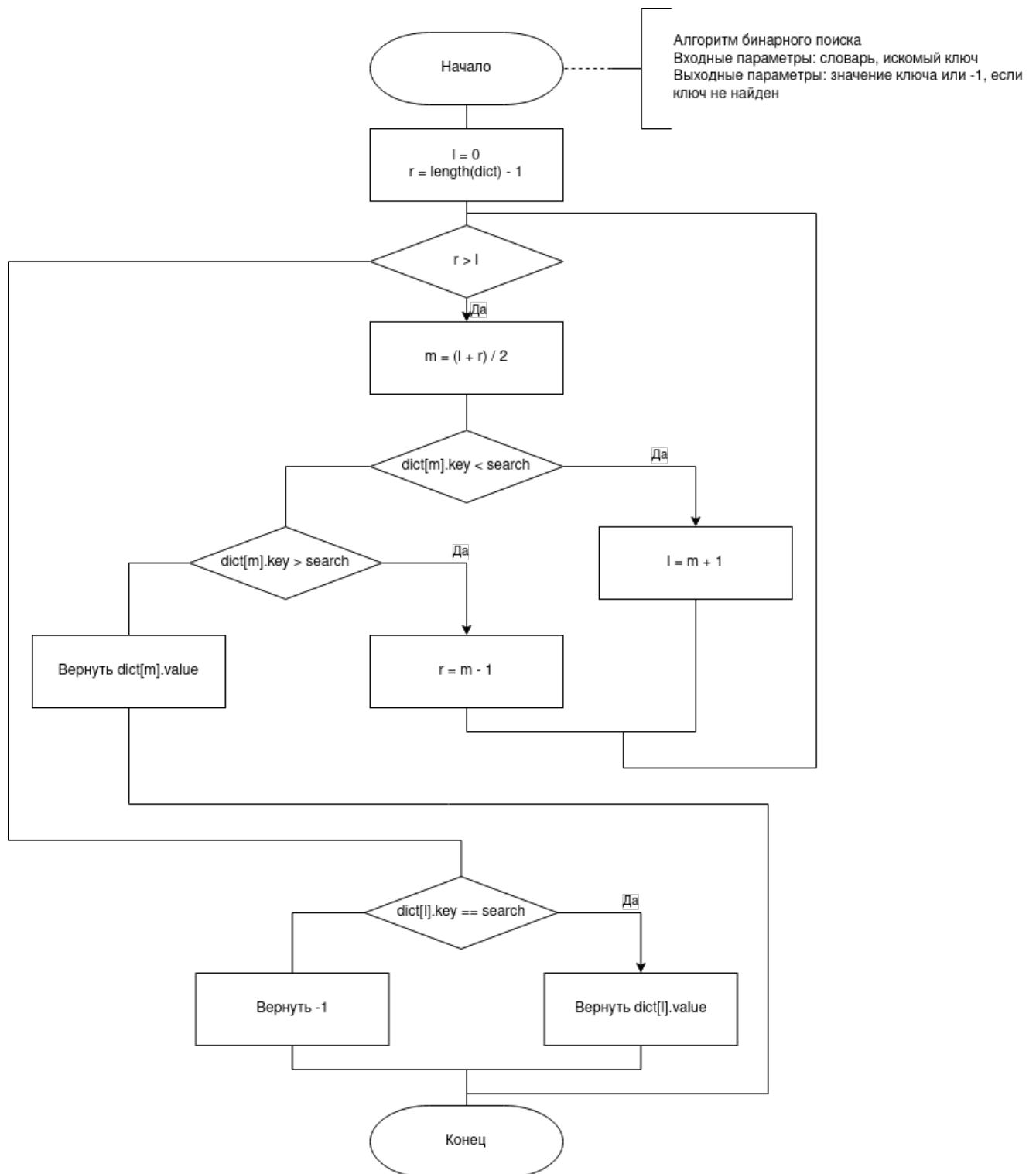


Рисунок 2.2 – Схема алгоритма двоичного поиска

На рисунке 2.3 приведена схема алгоритма частотного анализа.

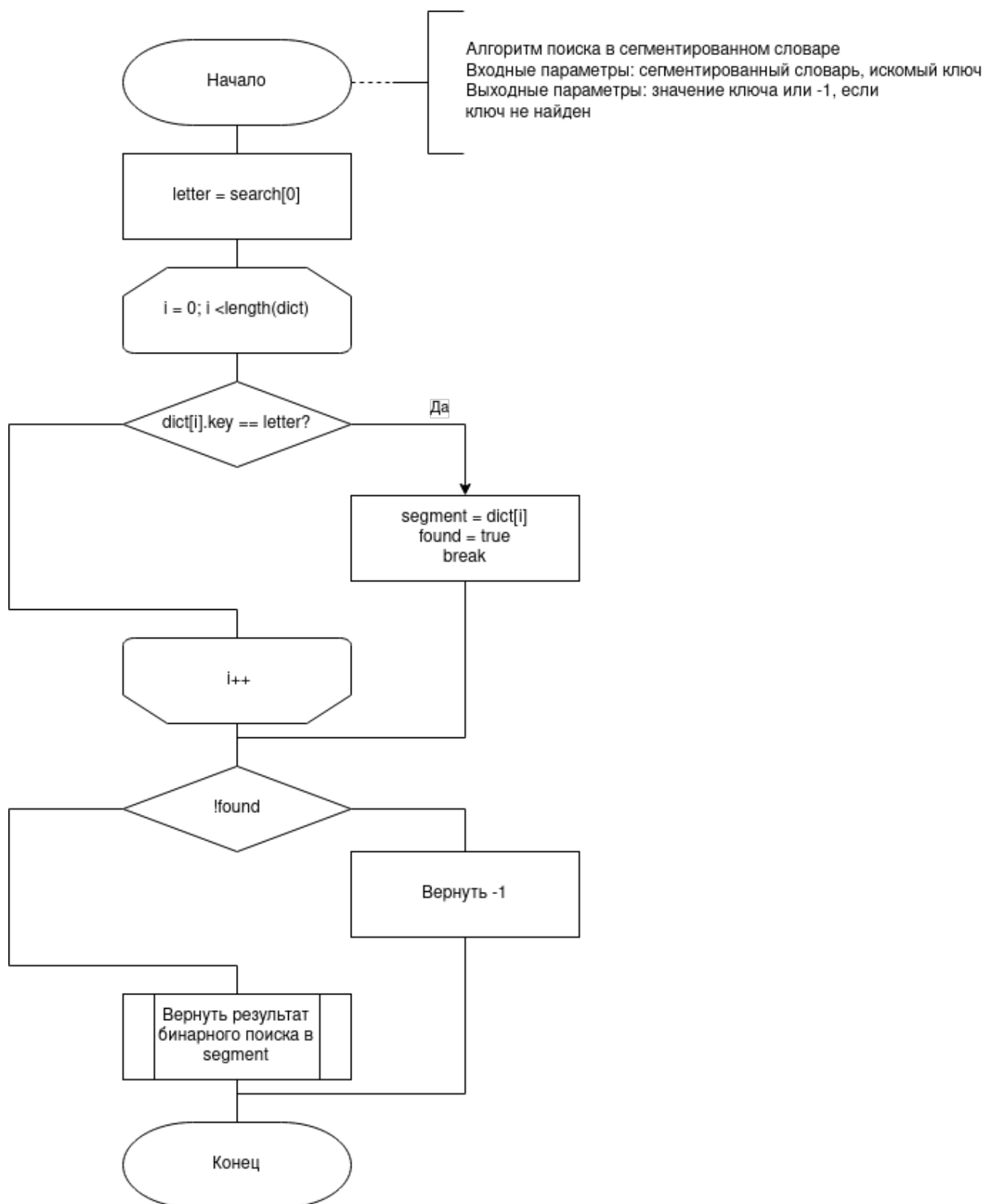


Рисунок 2.3 – Схема алгоритма частотного анализа

2.2 Описание структуры ПО

Программа будет включать в себя один смысловой модуль называемый *dict*, который будет содержать в себе несколько файлов реализующих процедуры и функции, связанные с алгоритмом полного перебора, алгоритма бинарного поиска и алгоритмом частотного анализа. Независимо от модуля будет существовать файл *main.go*, реализующий взаимодействие между пользователем и модулем, описанным ранее.

2.3 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [2]. Данный язык предоставляет средства тестирования разработанного ПО.

3.2 Листинг кода

В листингах 3.1–3.4 приведены реализации алгоритмов сортировок, а также вспомогательные функции.

Листинг 3.1 – Основной модуль программы

```
1 func main() {
2     fmt.Println("DICTIONARY_SEARCH")
3     var filename string
4     fmt.Printf("Enter the dictionary filename:")
5     fmt.Scanln(&filename)
6     var dictionary dict.Dict
7     dict.DictInit(&dictionary, filename)
8     fmt.Printf("DICTIONARY_SIZE=%v\n", len(dictionary))
9     var searchWord string
10    fmt.Printf("Enter the word to search:")
11    fmt.Scanln(&searchWord)
12    f := [3]func(dict.Dict, string) (int, int, bool){dict.BruteS, dict.BinS, dict.SegmS}
13    for _, currFunc := range f {
14        val, count, exists := currFunc(dictionary, searchWord)
15        fmt.Println(GetFunctionName(currFunc))
16        if !exists {
17            fmt.Printf("The key was not found. COMPARES=%d\n", count)
18        } else {
19            fmt.Printf("VALUE=%d COMPARES=%d\n", val, count)
20        }
21    }
22    dict.EfficiencyCheck(dictionary)
23 }
```

Листинг 3.2 – Алгоритм полного перебора

```
1 func BruteS(d Dict, searchword string) (int, int, bool) {
2     var count int
3     for _, item := range d {
4         count++
5         if item.Key == searchword {
6             return item.Value, count, true
7         }
8     }
9     return 0, count, false
10 }
```

Листинг 3.3 – Алгоритм двоичного поиска

```
1 func BinS(d Dict, searchword string) (int, int, bool) {
2     var count int
3     var l int
4     var r int = len(d) - 1
5
6     for r > l {
7         m := (l + r) / 2
8
9         if d[m].Key < searchword {
10             count++
11             l = m + 1
12         } else if d[m].Key > searchword {
13             count++
14             r = m - 1
15         } else {
16             return d[m].Value, count, true
17         }
18     }
19
20     if len(d) != 0 && d[l].Key == searchword {
21         count++
22         return d[l].Value, count, true
23     } else {
24         return 0, count, false
25     }
26 }
```

Листинг 3.4 – Алгоритм частотного анализа

```
1 func SegmS(d Dict, searchword string) (int, int, bool) {
2     sd := DictSegmentation(d)
3     var found bool
4     var segment Segment
5     var count int
6     letter := searchword[0]
7     for _, s := range sd {
8         count++
9         if s.Key == letter {
10             segment = s
11             found = true
12             break
13         }
14     }
15     if !found {
16         return 0, count, false
17     }
18     res, countBin, found := BinS(segment.Value, searchword)
19     return res, countBin, found
20 }
21
22 func DictSegmentation(d Dict) SegmentedDict {
23     var sd SegmentedDict
24     for _, item := range d {
25         letter := item.Key[0]
26         var found bool
27         for i := 0; i < len(sd); i++ {
28             if sd[i].Key == letter {
29                 sd[i].Value = append(sd[i].Value, item)
30                 found = true
31                 break
32             }
33         }
34
35         if !found {
36             var segment Segment
37             segment.Key = letter
38             segment.Value = append(segment.Value, item)
39             sd = append(sd, segment)
40         }
41     }
42     sort.Slice(sd, func(i, j int) bool { return len(sd[i].Value) > len(sd[j].Value) })
43     return sd
44 }
```

Тестирование производилось с помощью словаря, построенном на файле, содержащем данные: "the super file yes the". В таблице 3.1 приведены функциональные тесты для алгоритмов сортировки. Все тесты пройдены успешно (таблица 3.2).

Таблица 3.1 – Ожидаемый результат работы программы

№	Тестовый случай	Искомый ключ	Ожидаемый результат
1	Одно вхождение	super	value = 1, exists = 1
2	Несколько вхождений	the	value = 2, exists = 1
3	Нет вхождений	no	value = 0, exists = 0

Таблица 3.2 – Фактический результат работы программы

№	Тестовый случай	Искомый ключ	Фактический результат
1	Одно вхождение	super	value = 1, exists = 1
2	Несколько вхождений	the	value = 2, exists = 1
3	Нет вхождений	no	value = 0, exists = 0

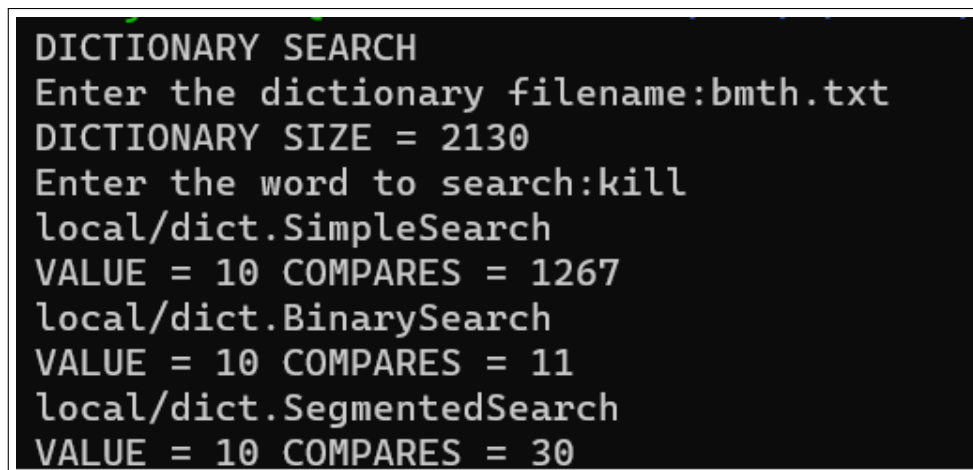
3.3 Вывод

Были разработаны и протестированы спроектированные алгоритмы: алгоритм полного перебора, алгоритм бинарного поиска и алгоритм частотного анализа.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.



```
DICTIONARY SEARCH
Enter the dictionary filename:bmth.txt
DICTIONARY SIZE = 2130
Enter the word to search:kill
local/dict.SimpleSearch
VALUE = 10 COMPARES = 1267
local/dict.BinarySearch
VALUE = 10 COMPARES = 11
local/dict.SegmentedSearch
VALUE = 10 COMPARES = 30
```

Рисунок 4.1 – Демонстрация работы алгоритмов

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10 64-bit [3].
- Память: 16 GB.
- Процессор: AMD Ryzen 5 4600H [4] @ 3.00 GHz.

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Оценка эффективности алгоритмов

Для выполнения лабораторной работы использовался словарь из 2130 элементов. Для оценки эффективности алгоритмов было подсчитано количество сравнений, необходимое для поиска каждого из ключей в словаре. Также были определены минимальное, среднее и максимальное количества сравнений для каждого алгоритма. Для наглядности результаты были представлены в виде гистограмм.

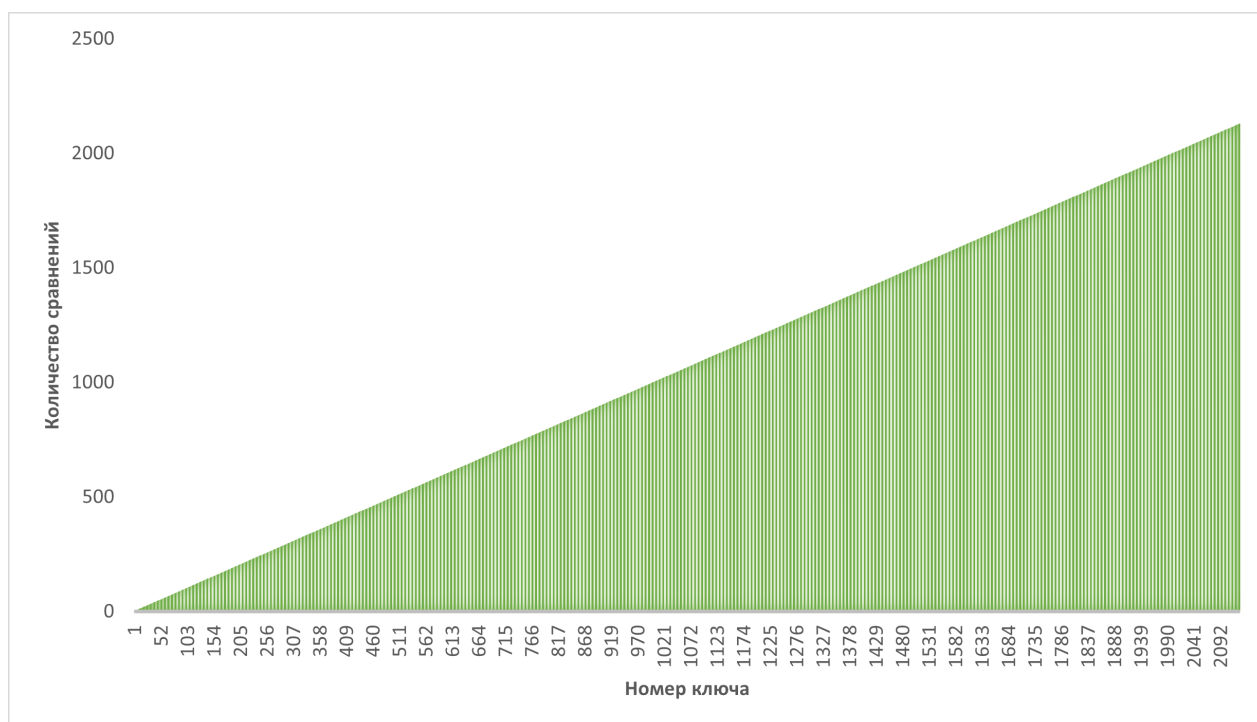


Рисунок 4.2 – Алгоритм полного перебора, сортировка ключей по алфавиту

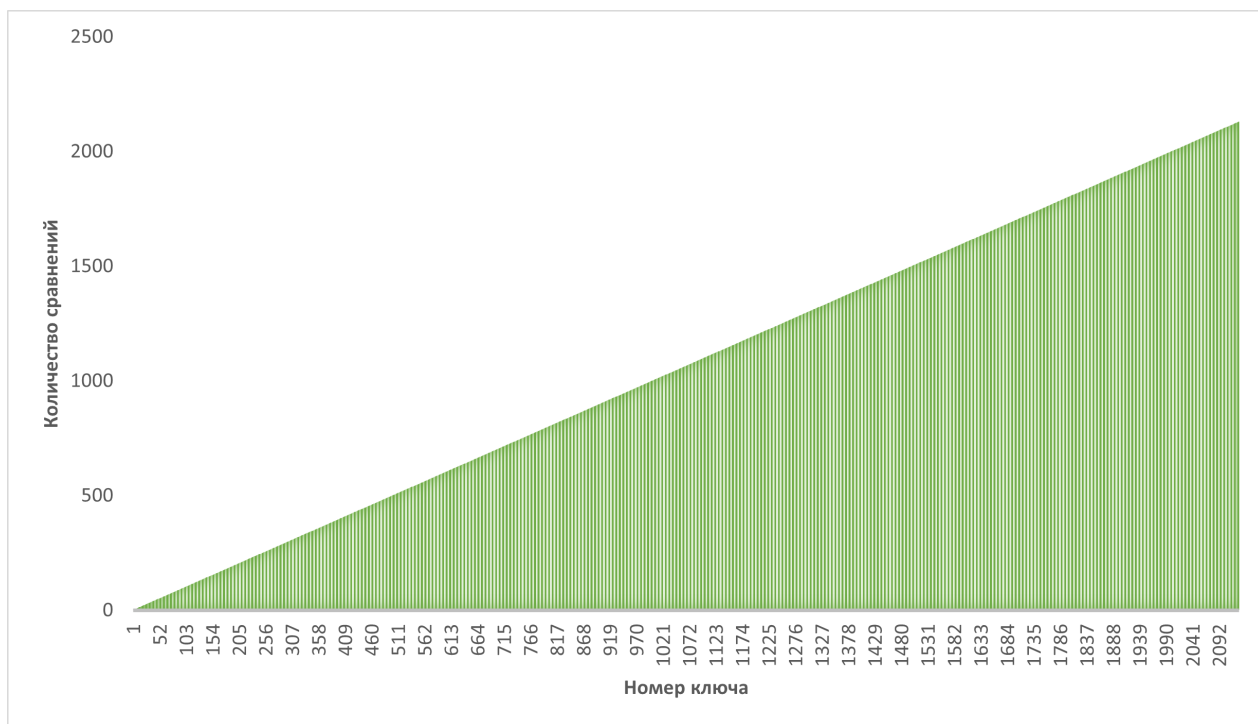


Рисунок 4.3 – Алгоритм полного перебора, сортировка по количеству сравнений

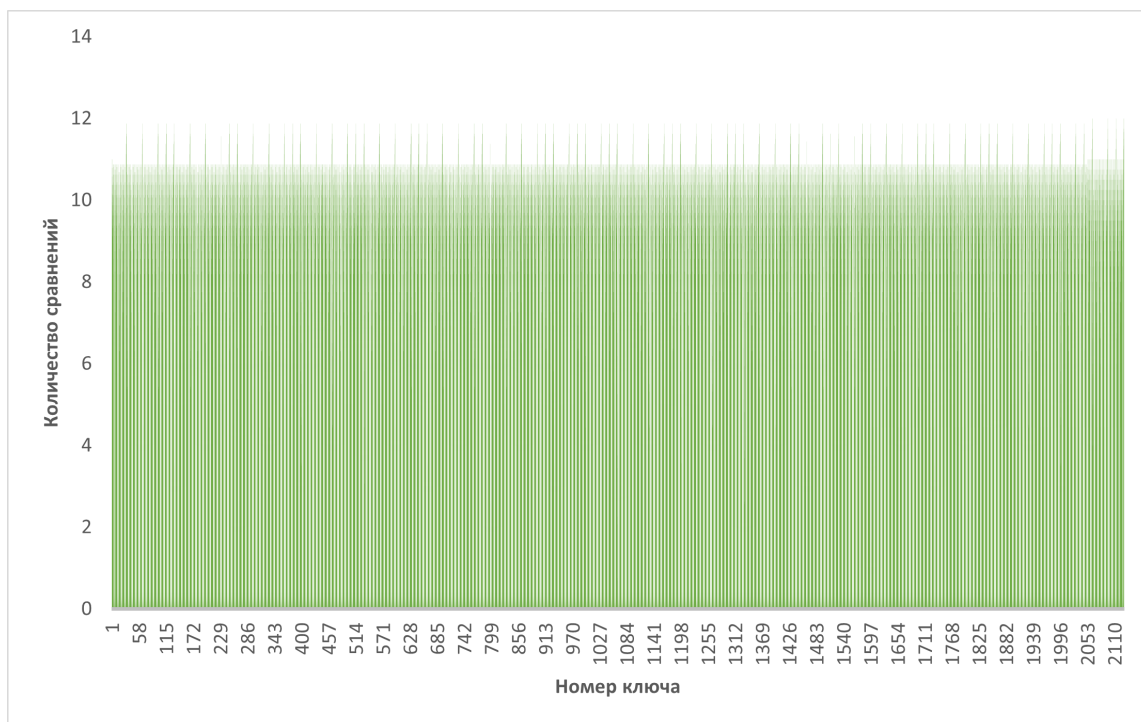


Рисунок 4.4 – Алгоритм бинарного поиска, сортировка ключей по алфавиту

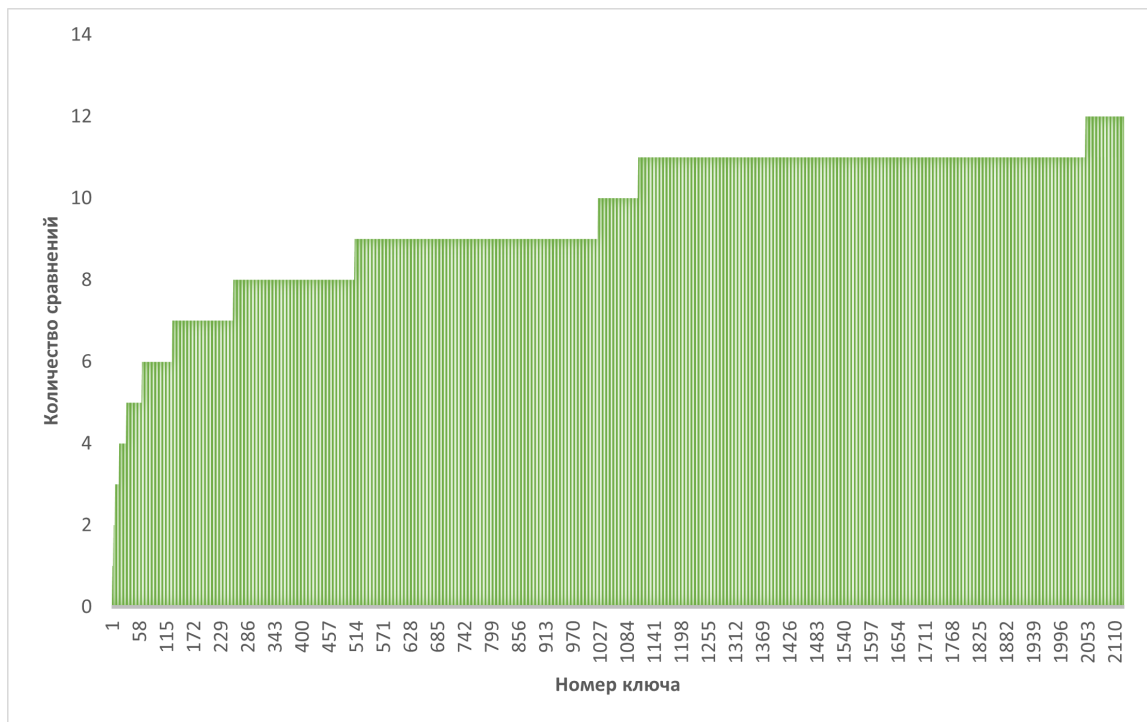


Рисунок 4.5 – Алгоритм бинарного поиска, сортировка по количеству сравнений

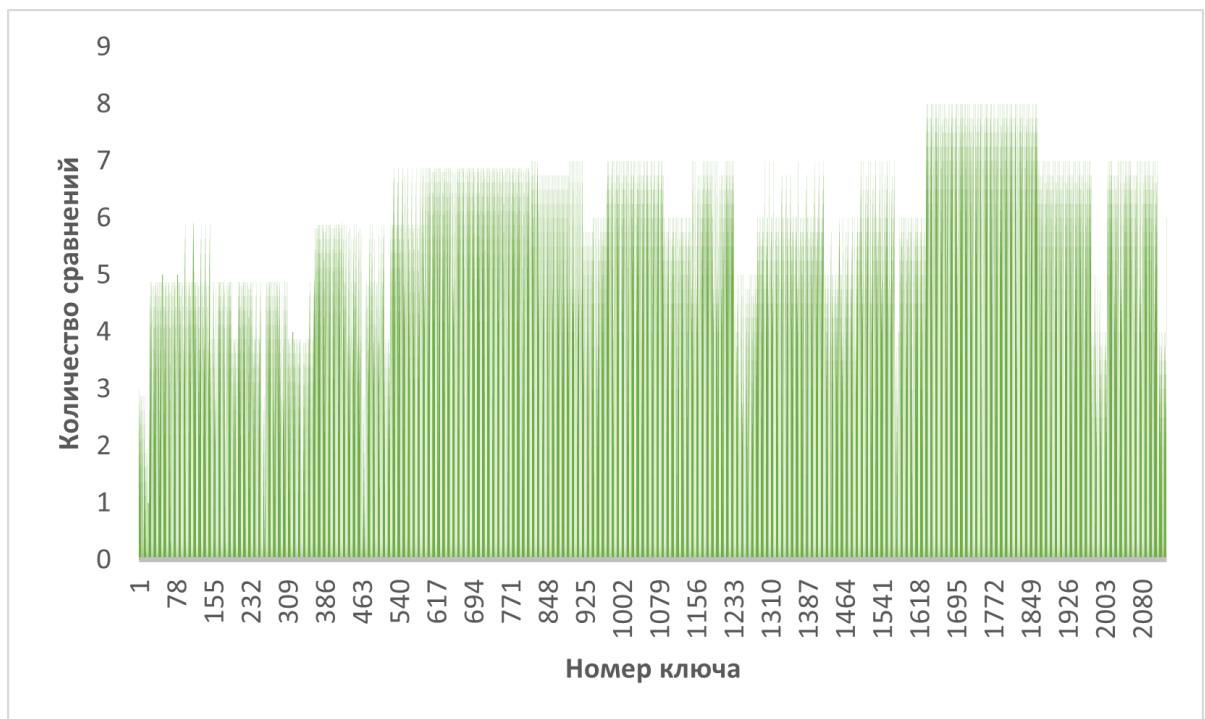


Рисунок 4.6 – Алгоритм частотного анализа, сортировка ключей по алфавиту

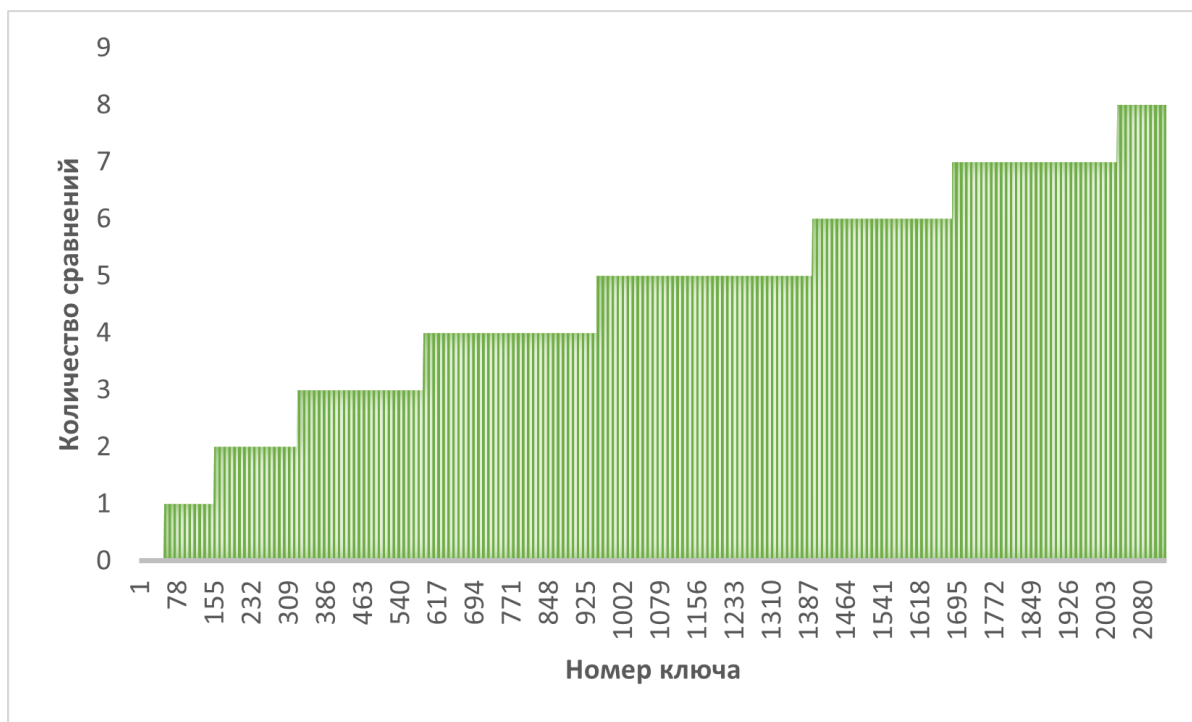


Рисунок 4.7 – Алгоритм частотного анализа, сортировка по количеству сравнений

4.4 Вывод

В данном разделе было проведено сравнение необходимых количеств сравнений при поиске ключа тремя алгоритмами: полный перебор, бинарный поиск, алгоритм частотного анализа.

Таблица 4.1 – Зависимость количества сравнений от выбора алгоритма

	Минимальное	Максимальное	Среднее
Полный перебор	1	2130	1065
Бинарный поиск	1	12	9.54
Алгоритм частотного анализа	1	8	4.54

Таким образом, сегментирование словаря в среднем сокращает количество необходимых сравнений в 1.26 раз, по сравнению с бинарным поиском, и в 222.16 раз, по сравнению с полным перебором. При этом среднее количество сравнений сократилось в 2.07 раз, по сравнению с бинарным поиском, и в 266.25 раз, по сравнению с полным перебором.

Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов поиска в словаре.

Сегментирование словаря в среднем сокращает количество необходимых сравнений в 1.26 раз, по сравнению с бинарным поиском, и в 222.16 раз, по сравнению с полным перебором. При этом среднее количество сравнений сократилось в 2.07 раз, по сравнению с бинарным поиском, и в 266.25 раз, по сравнению с полным перебором.

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] М.В. Ульянов. Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. – М.: ФИЗМАТЛИТ, 2007.
- [2] The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/> (дата обращения: 01.10.2021).
- [3] Explore Windows 10. [Электронный ресурс]. Режим доступа: <https://www.microsoft.com/en-us/windows/> (дата обращения: 02.10.2021).
- [4] AMD Processors [Электронный ресурс]. Режим доступа: <https://www.amd.com/en/products/apu/amd-ryzen-5-4600h> (дата обращения: 02.10.2021).
- [5] Testing – The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/pkg/testing/> (дата обращения: 08.10.2021).