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»

Лабораторная работа №7
по дисциплине "Анализ Алгоритмов"

Тема Поиск в словаре

Студент Рядинский К. В.

Группа ИУ7-53Б

Преподаватель Волкова Л. Л.

Москва

2021 г.

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 Perl, PHP, 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 оиска по словарю. Для достижения данной цели необходимо решить следующие задачи:

1. Изучить алгоритмы поиска по словарю.
2. Протестировать алгоритмы поиска по словарю.
3. Замерить и сравнить количество сравнений алгоритмов.
4. Сделать вывод на основе проделанной работы.

1 Аналитическая часть

В данном разделе представлены теоретические сведения о рассматриваемых алгоритмах.

1.1 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), где Ω - множество всех возможных случаев.

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

Из формулы (1), сгруппировав слагаемые, получим итоговую формулу для расчета средней трудоемкости работы алгоритма:

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

1.2 Алгоритм двоичного поиска

Данный алгоритм применяется к заранее упорядоченным словарям. Процесс двоичного поиска можно описать при помощи шагов:

— сравнить значение ключа, находящегося в середине рассматриваемого интервала (изначально – весь словарь), с данным;

— в случае, если значение меньше (в контексте типа данных) данного, продолжить поиск в левой части интервала, в обратном - в правой;

— продолжать до тех пор, пока найденное значение не будет равно данному или длина интервала не станет равной нулю (означает отсутствие искомого ключа в словаре).

Использование данного алгоритма для поиска в словаре в любом из случаев будет иметь трудоемкость равную $O(log_2(N))$. 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 алгоритм сортировки понадобится всего один раз.

1.3 Алгоритм частотного анализа

Алгоритм частотного анализа строит частотный анализ полученного словаря. Чтобы провести частотный анализ, нужно взять первый элемент каждого значения в словаре по ключу и подсчитать частотную характеристику, т.е. сколько раз этот элемент встречался в качестве первого. По полученным данным словарь разбивается на сегменты так, что все записи с одинаковым первым элементом оказываются в одном сегменте.

Сегменты упорядочиваются по значению частотной характеристики таким образом, чтобы к элементу с наибольшим значением характеристики был предоставлен самый быстрый доступ.

Затем каждый из сегментов упорядочивается по значению. Это необходимо для реализации бинарного поиска, который обеспечит эффективный поиск в сегмента при сложности $O(nlog(n))$

таким образом, сначала выбирается нужный сегмент, а затем в нем проводится бинарный поиск нужного элемента. Средняя трудоемкость при длине алфавита M может быть рассчитана по формуле (3).

$$\sum_{i \in [1,M]} (f_{select_i} + f_{search_i}) \tag{3}$$

1.4 Описание словаря

В данной работе словарь будет иметь следующий вид: *название игры : разработчик*. Поиск будет произведен по ключу *название игры*.

1.5 Вывод

В данном разделе были рассмотрены основополагающие материалы, которые в дальнейшем потребуются при реалиации алгоритмов поиска по словаю.

В качестве входных данных программе будут подаваться: файл, содержащий исходный словарь; ключ, по которому будет производиться поиск. Ограничением для работы программного продукта будут являться, что файл должен существовать и содержать корректные данные. Ключ должен являться корректной строкой (иметь длину более 1 символа).

Реализуемое программное обеспечение будет работать в двух режимах: экспериментальном и пользовательском. В пользовательском режиме может будет ввести ключ, по которому будет произведен поиск по словарю. В экспериментальном режиме будет возможность сравнить реализованные алгоритмы по временным характеристикам.

Критериеми, по которому данная реализация будет сравниваться с другими реализациями, будут являться: время работы алгоритма и количество сравнений, произведенных по ходу работы алгоритма.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы работы алгоритмов, используемые типы данных и структура программного обеспечения (далее ПО).

2.1 Схемы алгоритмов

На схемах 1-3 представлены реализации алгоритмов поиска по словарю.

Рис. 1 — Алгоритм поиска методом полного перебора


Начало
left = 1
right = длине словаря
Цикл пока left < right
middle = (left + right) / 2
A

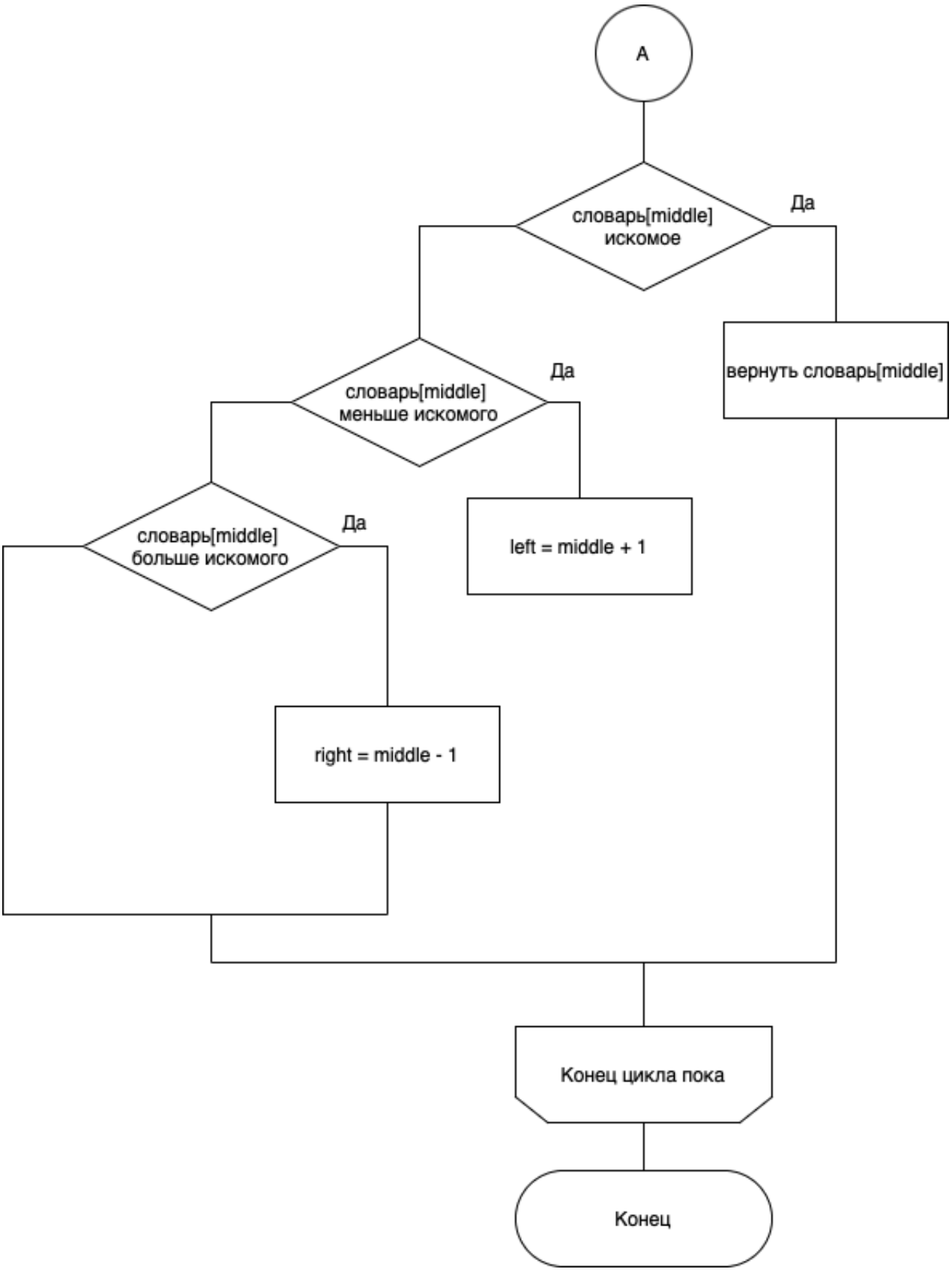

Рис. 2 — Алгоритм поиска методом дихотомии

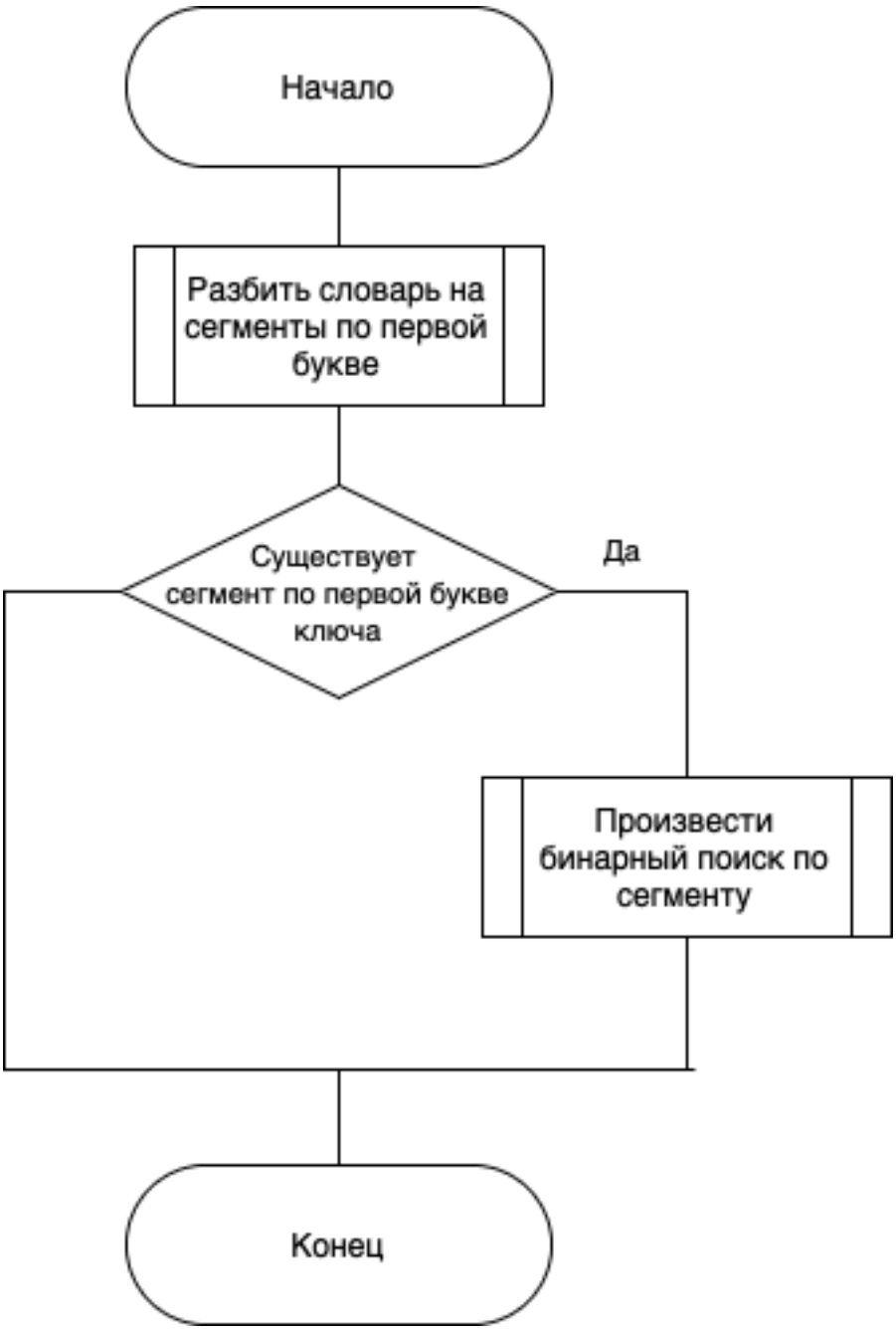

Рис. 3 — Алгоритм поиска методом частотного анализа

2.2 Описание структуры программного обеспечения

На рисунке 4 представлена *uml* диаграмма разрабатываемого программного обеспечения.

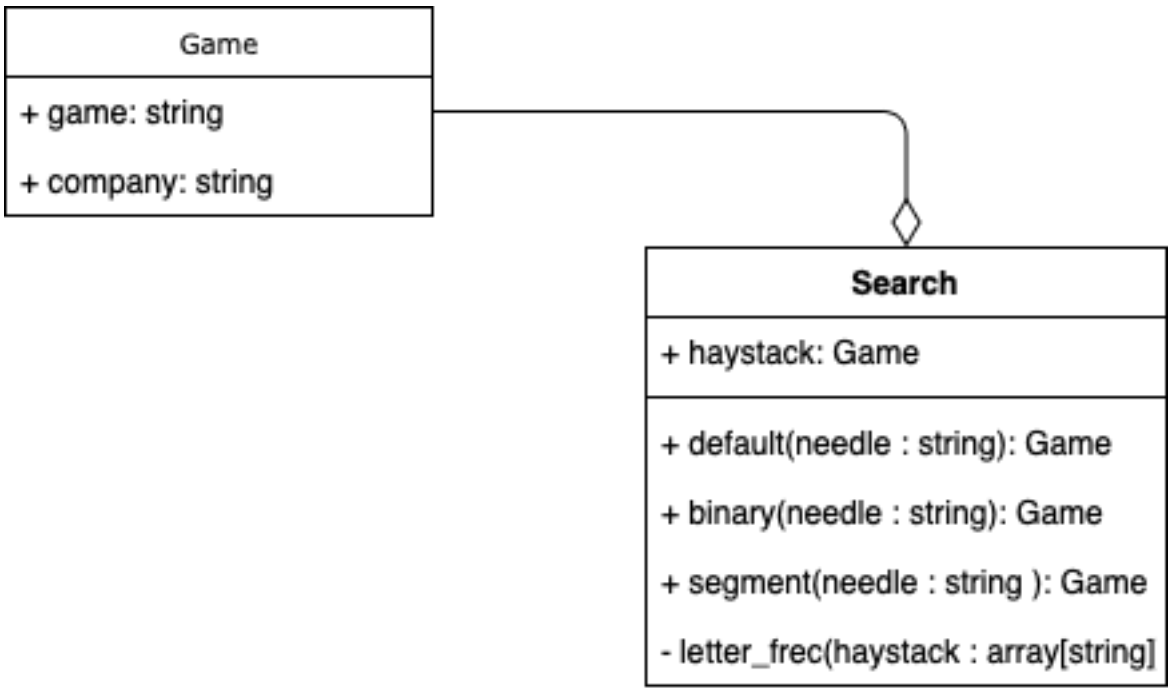

Рис. 4 — Структура программного обеспечения

2.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— ключ — строка;

— словарь — двумерный массив строк.

2.4 Структура программного обеспечения

Программное обеспечение состоит из следующих модулей:

— main.lua — модуль, содержащий код точки входа;

— search.lua — модуль, содержащий код функций поиска;

— type.lua — модуль, содержащий объявление словаря;

— benchmark.lua — модуль, содержащий функции замера временных характеристик.

2.5 Тестирование

Тестирование будет проводиться методом черный ящик. В таблице 1 представлены тесты.

Таблица 1 — Тестирование представленных реализаций алгоритмом поиска по словарю

Ввод	Ожидаемый результат	Фактический результат
qop	qop Quiet River	qop Quiet River
theHunter Classic	theHunter Classic Expansive Worlds	theHunter Classic Expansive W
Dota 3	Не найдено	Не найдено

2.6 Вывод

В данном разделе были представлены схемы алгоритмов, структура по, типы данных, тесты.

3 Технологическая часть

В данном разделе приведены средства реализации и листинги кода.

3.1 Средства реализации

К языку программирования выдвигаются следующие требования:

1. Возможность производить замер времени выполнения части программы.
2. Существуют среды разработки для этого языка.
3. Возможность чтение из файла, запись в файл, создание массивов.

По этим требованиям был выбран язык Lua.

3.2 Листинги кода

Листинг 1: Словарь

```
Game = {}

Game.__index = Game

function Game:New(n, c)
    assert(n ~= nil and c ~= nil, 'name and company must be provided')

    local self = setmetatable({}, Game)

    self.game = n
    self.company = c

    return self
end

function Game:__tostring()
    return self.game .. " : " .. self.company
end

return Game
```

Листинг 2: Точка входа

```
local s = require 'split'
local csv = require 'csv'
local game = require 'type'
local search = require 'search'
local bench = require 'benchmark'
local cmp = require 'cmp_count'

function main()
    data, err = csv.read('../data/data1.csv', ';')

    local dict = {}

    for i, v in ipairs(data) do
        local g = Game:New(v[1], v[2])

        table.insert(dict, g)
    end

    io.write("Ipnut game name: ")
    local g_name = io.stdin:read('l')

    io.write('Benchmark? (y/n): ')

    local y_n_debug = io.stdin:read('l')
    local finder = search:New(dict)

    if y_n_debug == 'y' then
        local res = total_cmp(dict)

        dump_cmps(res, '../data/cmps.csv')

        print(string.format("Default: %f", bench.cpu_run(function ()
            finder:default(g_name)
        end, 1000, 'ns')))
        print(string.format("Binary:  %f", bench.cpu_run(function ()
            finder:binary(g_name)
        end, 1000, 'ns')))
```

```
38          print(string.format("Segment: %f", bench.cpu_run(
    function ()
39              finder:segment(g_name)
40          end, 1000, 'ns')))
41      else
42          local sd = finder:default(g_name) or 'not found'
43          local sb = finder:binary(g_name)  or 'not found'
44          local ss = finder:segment(g_name) or 'not found'
45
46          print('Default:', sd[1], sd[2])
47          print('Binary: ', sb[1], sb[2])
48          print('Segment:', ss[1], ss[2])
49      end
50  end
51
52  main()
```

Листинг 3: Реализация алгоритмов поиска

```
 1  local search = {}
 2  search.__index = search
 3
 4  function search:New(h)
 5      local self = setmetatable({}, search)
 6
 7      self.haystack = h
 8      self.segments = nil
 9
10      return self
11  end
12
13  function search:default(needle)
14      local haystack = self.haystack
15
16      local comp = 0
17
18      for i, v in ipairs(haystack) do
19          comp = comp + 1
20          if v.game == needle then
21              return {v, comp}
22          end
```

```
23      end
24
25      return {nil, comp}
26  end
27
28  function search:binary(needle)
29      local haystack = self.haystack
30      local left = 1
31      local right = #haystack
32
33      local comp = 0
34
35      while left <= right do
36          local middle = math.floor((left + right) / 2)
37
38          if (haystack[middle].game == needle) then
39              comp = comp + 1
40              return {haystack[middle], comp}
41          elseif haystack[middle].game < needle then
42              comp = comp + 2
43              left = middle + 1
44          else
45              comp = comp + 2
46              right = middle - 1
47          end
48      end
49
50      return {nil, comp}
51  end
52
53  local function letter_freq(haystack)
54      local res = {}
55
56      for i, v in ipairs(haystack) do
57          if res[v.game:sub(1, 1)] == nil then
58              res[v.game:sub(1, 1)] = { v }
59          else
60              table.insert(res[v.game:sub(1, 1)], v)
61          end
62      end
```

```
63
64      return res
65  end
66
67  function search:segment(needle)
68      local haystack = self.haystack
69
70      if self.segments == nil then
71          self.segments = letter_freq(haystack)
72      end
73
74      local letter = needle:sub(1, 1)
75      local cmp = 0
76      for k, v in pairs(self.segments) do
77          cmp = cmp + 1
78          if k == letter then
79              local value = search:New(v)
80              local res = value:binary(needle)
81
82              res[2] = res[2] + cmp
83              return res
84          end
85      end
86
87      return nil
88  end
89
90  return search
```

3.3 Вывод

В данном разделе были разработаны алгоритмы поиска.

4 Исследовательский раздел

В данном разделе будет проведен замер временных характеристик выполнения алгоритмов и пример работы программы.

4.1 Пример работы программы

На рисунке 5 представлен пример запуска программы.

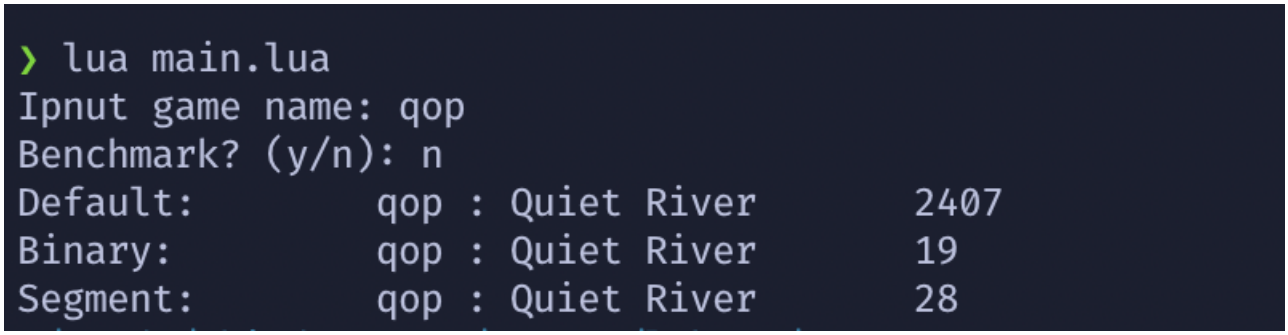

Рис. 5 — Пример работы программы

4.2 Технические характеристики

Технические характеристики электронно-вычислительной машины, на которой выполнялось тестирование:

— операционная система: macOS BigSur версия 11.4;

— оперативная память: 8 гигабайт LPDDR4;

— процессор: Apple M1.

4.3 Временные характеристики

В данном работе более интересующей нас характеристикой будет являться количество сравнений, необходимое для нахождения ключа, так как от количества сравнений прямо пропорционально зависит время работы алгоритма. Количество элементов в словаре примерно 2500.

Поиск будет проводиться с помощью каждого реализованного алгоритма, после чего будут представлены соответствующие графики. В каждом построенном графике данные будут отсортированы по возрастанию, так как количество

сравнений для алгоритмов, не использующих полный перебор, количество сравнений будет являться "случайной"величиной.

4.3.1 Алгоритм полного перебора

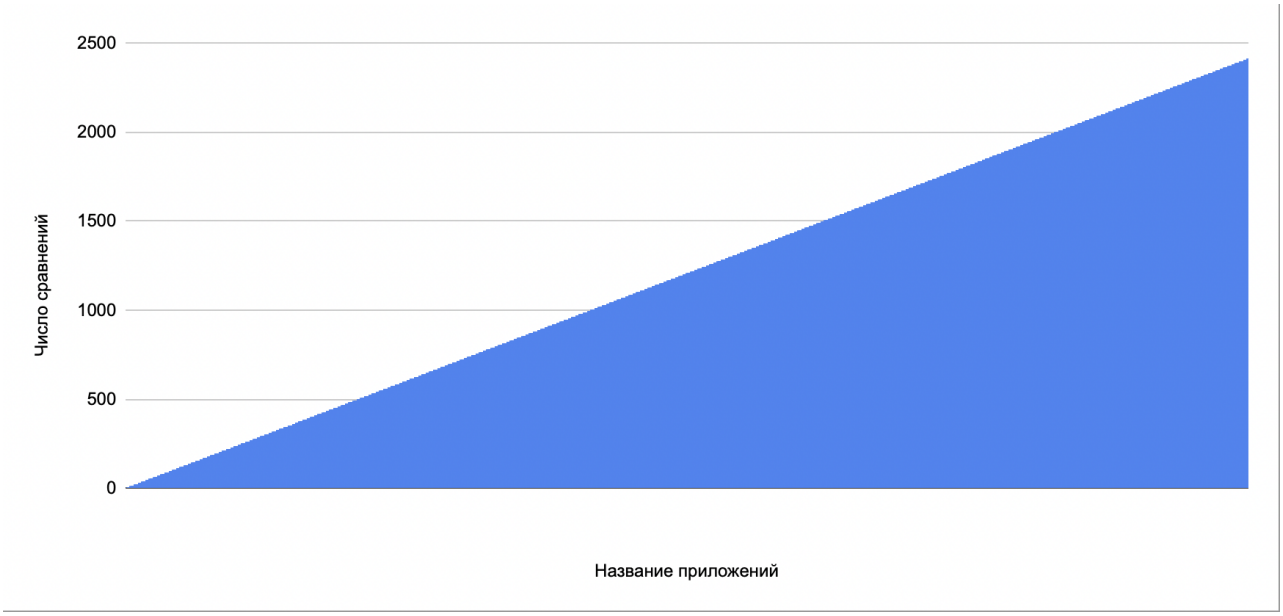

Рис. 6 — График количества сравнений для алгоритма поиска полным перебором

Как видно из графика 6 количество сравнений линейно зависит количества элементов в словаре. Лучшим случаем для алгоритма полного перебора будет являться нахождение искомого элемента в самом начале словаря, тогда как худшим будет являться элемент, находящийся в самом конце словаря.

4.3.2 Алгоритм бинарного поиска

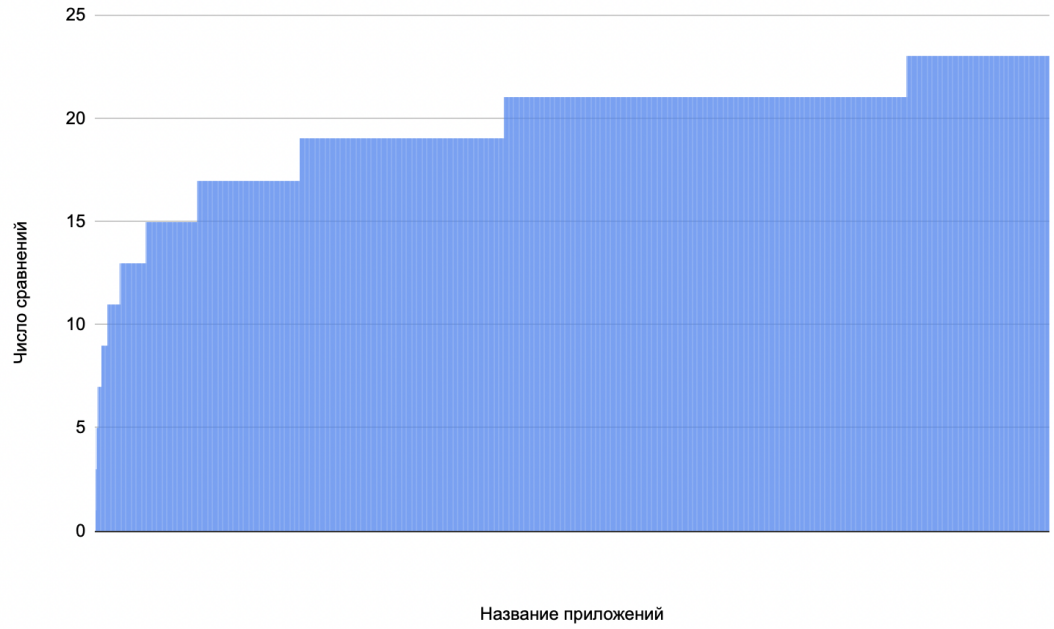

Рис. 7 — График количества сравнений для алгоритма поиска методом дихотомии (данные в графике отсортированы по возрастанию)

Как видно из графика 7 минимальное количество сравнений равно 1 (элемент находится ровно по середине), а максимальное равно 23. По сравнению с последовательным алгоритмом среднее количество сравнений отличается примерно в 61 раз.

4.3.3 Алгоритм частотного анализа

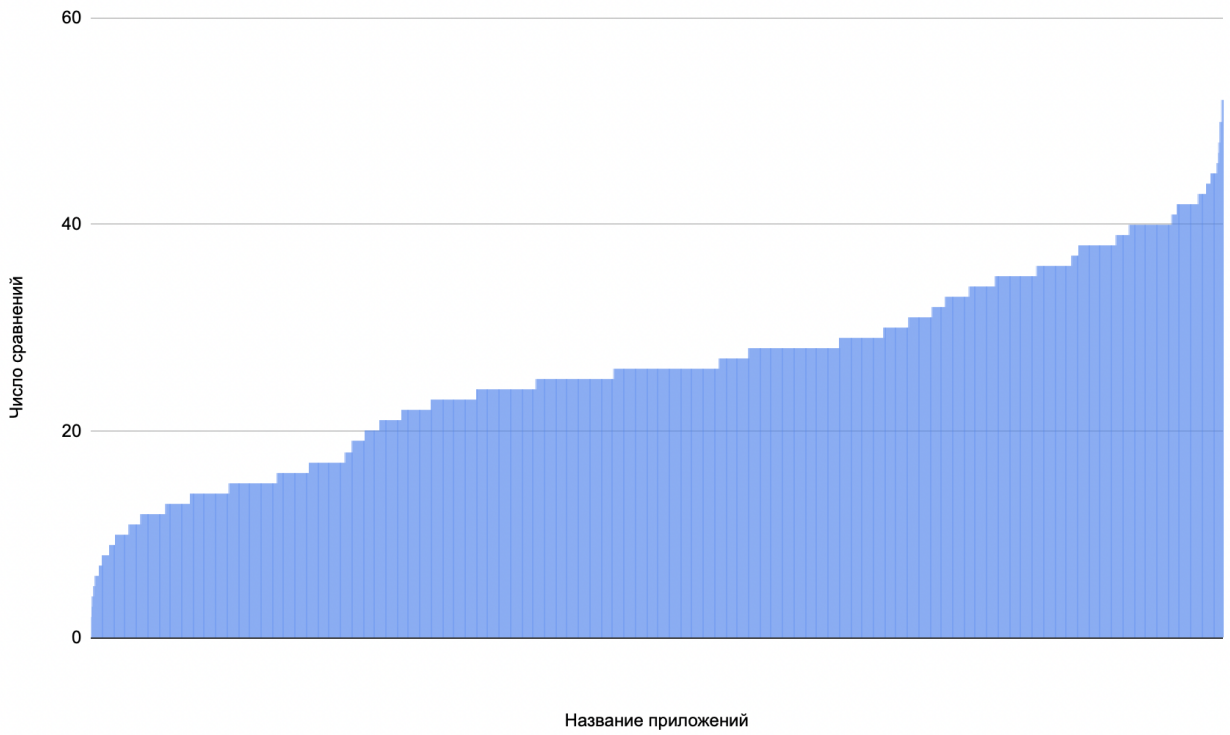

Рис. 8 — График количества сравнений для алгоритма частотного анализа (данные в графике отсортированы по возрастанию)

Так как словарь содержит не только буквы латинского словаря, то в худшем случае потребуется пройти по всему списку алфавита начальных букв словаря, что может потребовать значительное количество сравнений.

На рисунке 8 видно, что в лучшем случае потребуется 2 сравнения (элемент находится по середине первого сегмента). В худшем же 52 сравнения. В среднем количество сравнений в алгоритме полного перебора и алгоритме частотного анализа отличаются в 46 раз. Но по сравнению с алгоритмом бинарного поиска в 1.36 сравнений больше.

4.4 Вывод

Исходя из вышеперечисленных данных, можно сделать вывод, что наиболее эффективным алгоритмом поиска является бинарный алгоритм. Алгоритм частотного анализа же лишь в особых случаях будет быстрее бинарного поиска (если алфавит будет сильно ограничен и размер словаря будет очень большим),

так как требуется произвести сегментацию словаря, что требует значительное количество времени.

Отдельно стоит отметить, что бинарному поиску требуется подавать на вход отсортированный словарь, на что может потребоваться значительное количество времени и в единичных случаях поиска последовательный алгоритм может быть эффективнее. Но если требуется произвести серию поисков, то время, затраченное на сортировку словаря, окупится сниженным временем поиска.

Заключение

В данном лабораторной работе были изучены алгоритмы поиска по словарю.

Среди рассмотренных алгоритмов наиболее эффективным оказался алгоритм бинарного поиска.

В рамках данной работы была выполнена цель и решены следующие задачи:

1. Изучены алгоритмы поиска по словарю.
2. Протестированы алгоритмы поиска по словарю.
3. Замерено и сравнено количество сравнений алгоритмов.
4. Сделаны выводы на основе проделанной работы.

Список литературы

1. Visual Studio Code [Электронный ресурс], режим доступа: https://code.visualstudio.com/ (дата обращения: 14 декабря 2021 г.)
2. LPDDR4 [Электронный ресурс] https://ru.wikipedia.org/wiki/LPDDR#LPDDR4 (дата обращения: 14 декабря 2021 г.)
3. Ульянов М. В. Ресурсно-эффективные компьютерные алгоритмы. Разработка и Анализ. - Наука Физматлит, 2007. - 376.
4. Язык Lua. [Электронный ресурс] Режим доступа: https://www.lua.org (дата обращения: 14 декабря 2021 г.)
5. Вирт Н. Алгоритмы + структуры данных = программы. — М.: «Мир», 1985. — С. 28.