Московский авиационный институт (Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»

Кафедра «Вычислительная математика и программирование»

Курсовой проект по курсу "Дискретный анализ"

Студент: Ю. Ю. Обыденкова
Преподаватель: А. А. Журавлёв
Группа: М8О-308Б-18

Дата:
Оценка:
Подпись:

Москва
2021

Курсовой проект по теме: “Аудиопоиск”

Запуск и параметры:

```
./prog index --input <input file> --output <index file>
./prog search --index <index file> --input <input file> --output <output file>
```

Входные файлы содержат в себе имена файлов с аудио записями по одному файлу в строке.

Результатом ответа на каждый запрос является строка с названием файла, с которым произошло совпадение, либо строка “! NOT FOUND”, если найти совпадение не удалось.

Описание

Постановка задачи

Задача состоит в применении быстрого преобразования Фурье для получения списка частот из списка амплитуд. Уникальность аудиофайлу дают как раз его частоты, а если точнее — самые громкие частоты. Это гарантирует, что если искомый аудиофайл будет с посторонними шумами, в плохом качестве и намного меньшего размера чем исходный, то всё равно можно будет обнаружить совпадение с исходной записью. В теории можно ограничиться поиском единственной максимальной частоты для конкретного временного отрезка в записи, но логичнее искать несколько максимумов для разных диапазонов — ударные инструменты, голос, гитара, скрипка.

Дискретизация аналогового сигнала

Преобразование аналогового сигнала в цифровой состоит из двух этапов: дискретизации по времени и квантования по амплитуде. Дискретизация по времени означает, что сигнал представляется рядом отсчетов (сэмплов), взятых через равные промежутки времени. Например, если частота дискретизации 44100, то это означает, что сигнал измеряется 44100 раз в течение одной секунды.

Чем больше частота, тем точнее соответствует цифровой сигнал аналоговому. Однако если сделать частоту дискретизации слишком большой, то возрастёт плотность потока данных, вычислительная нагрузка на процессоры и, конечно же, объём памяти, необходимый для хранения.

Считается, что человек слышит частоты в диапазоне от 20 до 20 000 Гц. Согласно теореме Котельникова, для того, чтобы аналоговый (непрерывный по времени) сигнал можно было точно восстановить по его отсчетам, частота дискретизации должна быть как минимум вдвое больше максимальной звуковой частоты.

Сегодня самыми популярными частотами являются 44,1 кГц (CD) и 48 кГц (DAT).

Мной была использована частота 44100.

Обработка аудиофайла

Для получения списка отсчётов (семплов) была использована библиотека mpg123.

```
1  template <size_t LENGTH, uint32_t STEP>
2  std::vector<float> Shazam<LENGTH, STEP>::processFile(const std::string&
       name)
3  {
4          auto mh = mpg123_new(NULL, NULL);
5          assert(mh != NULL);
6          assert(mpg123_param(mh, MPG123_FLAGS, MPG123_MONO_MIX |
               MPG123_QUIET | MPG123_FORCE_FLOAT, 0.) == MPG123_OK);
7
8          auto errorCode = mpg123_open(mh, name.c_str());
9          assert(errorCode == MPG123_OK);
10         long rate;
11         int channels, encoding;
12         errorCode = mpg123_getformat(mh, &rate, &channels, &encoding);
13         assert(errorCode == MPG123_OK);
14         if (rate != 44100) {
15                 mpg123_delete(mh);
16                 return {};
17         }
18         const size_t part_size = 1024;
19         unsigned char part[part_size];
20         size_t bytesRead;
21         std::vector<float> samples;
22         size_t bytesProcessed = 0;
23         do {
24                 int err = mpg123_read(mh, part, part_size, &bytesRead);
25                 samples.resize((bytesProcessed + bytesRead) / 4 + 1);
26                 memcpy(reinterpret_cast<unsigned char*>(samples.data()) +
                       bytesProcessed, part, bytesRead);
27                 bytesProcessed += bytesRead;
28                 if (err == MPG123_DONE)
29                         break;
30
31                 assert(err == MPG123_OK);
32         } while (bytesRead > 0);
33         samples.resize(bytesProcessed / 4);
34         errorCode = mpg123_close(mh);
35         assert(errorCode == MPG123_OK);
36         mpg123_delete(mh);
37         return samples;
38 }
```

Полученный вектор обрабатывается дискретным преобразованием Фурье по частям и с некоторым шагом. Размер одной части по умолчанию равен 4096 элементов. Число было выбрано степенью двойки не случайно, ибо так Фурье работает наиболее эффективно.

После обработки отрезка каждый его элемнет будет соответствовать $44100/4096 \approx$

11 Герцам. Шаг означает временное смещение при обработке аудиофайла. Размер шага напрямую влияет на скорость обработки и на точность при нахождении совпанений. Чем меньше шаг — тем дольше обработка и точнее совпадение. Мной был выбран шаг 1024, что соответствует сдвигу в $1024/44100 \approx 0.02$ секунды вдоль записи.

Составление базы данных

Выбранные диапазоны частот (в Герцах): 40 — 300 — 800 — 1500 – 2700

И их относительное расположение на отрезке: “3 27 74 139 250”

Итого одному отрезку будет соответствовать 4 числа — самые громкие частоты из указанных диапазонов. Используя эти максимумы и время начала отрезка относительно начала аудиофайла можно составить таблицу для поиска совпадений.

По полученным четырём числам можно вычислить хеш, который будет соответствовать только такой последовательности чисел, например по формуле:

$hash(h_1, h_2, h_3, h_4) = h_1 + h_2 * (27 - 3)$
$+ h_3 * (74 - 27) * (139 - 74)$
$+ h_4 * (74 - 27) * (139 - 74) * (250 - 139)$
Тогда количество хешей будет ограничено числом
$(74 - 27) * (139 - 74) * (250 - 139) * (274 - 250) = 8138520$

Такого количества достаточно для точной идентификации аудиофайла, так как для 5-минутной записи будет посчитано около 13000 хешей.

Соответственно для идентификации песни по её части нужно посчитать количество хешей, совпадающих с исходной песней с некоторым смещением.

Сама же база данных будет состоять из двух таблиц — одна будет состоять из ID файлов (vector<vector<ID>>), а вторая из моментов времени вхождения хеша в песне (vector<vector<Time>>).

Индекс каждого элемента в векторе — это его хеш, а в самом элементе лежит вектор из чисел.

Сжатие базы данных

Итого соответствующие элементы таблиц будут выглядеть примерно так:

ID	1	1	1	1	2	2	2	4	5	10	10	12	12	15	15
Time	0	1	2	3	5	8	10	0	0	17	18	3	7	1	16

Можно было сделать не две таблицы, а одну - состоящую из пар “<ID, Time>”, но создание двух таблиц оправданно, так как можно к каждой из них применить алгоритм сжатия simple9.

Этот алгоритм не изменяет способ представления данных, как, например, архиваторы, он лишь позволяет оптимально использовать память для их записи.

Сам алгоритм состоит из двух частей: небольшая предварительная обработка числовой последовательности и непосредственно запись в сжатой форме.

Первый шаг будет проще показать на примере. Пусть дана исходная последовательность “1 1 1 1 2 2 2 4 5”. После обработки она будет выглядеть следующим образом: “1 0 0 0 1 0 0 2 1 5”. То есть вместо каждого числа записывается его разница с предыдущим числом (первое число записывается неизменным).

Можно заметить что для записи любого из этих чисел будет достаточно 3 бита (число 5 как самое большое будет занимать 3 бита в двоичной форме). Это и есть суть сжатия в simple9 оптимальное использование битового представления числа.

Для сжатия используются 32 байта (обычная 4-байтовая переменная типа int). Первые четыре бита отдаются под служебные нужды (запись режима), остальные же 28 используются для хранения чисел.

В примере выше мы получили последовательность “1 0 0 0 1 0 0 2 1 5” и определили что для записи лобого из её чисел будет достаточно 3 бита. Для записи чисел могут быть использованы 28 бит, итого, если мы положим размер одного числа равным трём, то сможем записать $28/3 = 9.33333$ чисел.

mode	001	000	000	000	001	000	000	010	001	101	0

Так будут выглядеть эти числа после сжатия. Но останется один неиспользованный бит, “лишний”.

Режимы:

Номер режима (mode)	1	2	3	4	5	6	7	8	9
Битовый размер числа	1	2	3	4	5	7	9	14	28
Количество чисел для записи	28	14	9	7	5	4	3	2	1
Неиспользуемые биты	0	0	1	0	3	0	1	0	0

В примере выше номер режима будет равен 3, то есть первые 4 бита будут выглядеть так: 0011.

Этот метод сжатия будет очень эффективен для первой таблицы, так как в ней будет много повторяющихся чисел, которые будут хорошо сжиматься. Для второй таблицы результат будет не столь впечатляющий — последовательность чисел не всегда будет возрастающей и нельзя будет применить оригинальную предварительную обработку, а числа будут возрастающими последовательностями, а не неубывающими, как в первой таблице, что тоже отрицательно влияет на степень сжатия.

Так же легко заметить, что при помощи simple9 невозможно записать числа большие чем 2^{28}. Хотя simple9 и прост для понимания, его реализация заняла по объёму столько же, сколько занимает весь остальной курсовой проект.

Описание реализации

Начинается всё с обработки сигнала при помощи преобразования Фурье. Но, так как известно, что будут обрабатываться только действительные числовые последо-

вательности, можно использовать один трюк — провести преобразование сразу для двух последовательностей. Если записать первую последовательность в действительную часть комплексного массива, а вторую — в мнимую, то после преобразования Фурье из полученной последовательности можно будет восстановить две исходные как будто к ним по отдельности применили Фурье. Функция DFFT запускает преобразование для двух последовательностей, записанных в одну и разделяет их для дальнейшей обработки.

Функция GetHashes прогоняет вектор семплов через Фурье и получает наборы частот, на которых ищет максимумы, и потом по ним считает хеш. Хеш заносится в базу данных.

Поиск осуществляется при помощи функций _search и search. Первая составляет следующую таблицу: map<SongID, map<TimeOffset, Count>>. Это означает что для каждой песни составляется таблица смещений по времени и для каждого смещения заводится счётчик — количество совпавших хешей для конкретной песни SongID, со смещением относительно начала TimeOffset и счетчиком Count. Функция search лишь немного преобразовывает структуру, полученную из первой функции: map<MatchPercent, pair<SongID , TimeOffset>>.

Исходный код

main.cpp

```
1  #include <fstream>
2  #include <iostream>
3
4  #include "Shazam.hpp"
5
6  #define usage
       \
7      {
           \
8          std::cout << argv[0] << " index --input <input file> --output <
               index file>\n"                                              \
9                    << argv[0] << " search --index <index file> --input <
                         input file> --output <output file>" << std::endl; \
10         return 1;
               \
11     }
12
13 // returns the argument going after the key
14 std::string GetName(const int argc, char** argv, const std::string key)
15 {
16     for (int i = 0; i < argc; ++i)
```

```
17          if (key == argv[i])
18              return ++i != argc ? argv[i] : "";
19      return "";
20  }
21
22  int main(int argc, char** argv)
23  {
24      Shazam base;
25      if (!strcmp(argv[1], "index")) {
26          std::string input, output;
27          input = GetName(argc, argv, "--input");
28          output = GetName(argc, argv, "--output");
29          if (output == "" || input == "")
30              usage;
31          std::ifstream in(input);
32          std::ofstream out(output);
33
34          if (!in.is_open()) {
35              std::cout << "could not open file \"" << input << "\" !\n";
36              return 1;
37          }
38          if (!in.is_open()) {
39              std::cout << "could not open file \"" << output << "\" !\n";
40              return 1;
41          }
42
43          std::string line;
44          std::vector<std::string> names;
45          while (std::getline(in, line))
46              names.push_back(line);
47          //!!! threading
48          base.CreateBase(names, 8);
49          base.write(out);
50      } else if (!strcmp(argv[1], "search")) {
51          std::string index, input, output;
52          index = GetName(argc, argv, "--index");
53          input = GetName(argc, argv, "--input");
54          output = GetName(argc, argv, "--output");
55          if (index == "" || input == "" || output == "")
56              usage;
57          std::ifstream ind(index);
58          std::ifstream inp(input);
59          std::ofstream out(output);
60          if (!ind.is_open()) {
61              std::cout << "could not open file \"" << index << "\" !\n";
62              return 1;
63          }
64          if (!inp.is_open()) {
65              std::cout << "could not open file \"" << input << "\" !\n";
66              return 1;
67          }
```

```
68          if (!out.is_open()) {
69              std::cout << "could not open file \"" << output << "\" !\n";
70              return 1;
71          }
72          base.read(ind);
73
74          std::string line;
75          while (std::getline(inp, line)) {
76              std::string res = base.search(line);
77              out << res << "\n";
78          }
79      } else
80          usage;
81      return 0;
82  }
```

TComplex.hpp

```
1   #ifndef TCOMPLEX_HPP
2   #define TCOMPLEX_HPP
3   #include <cmath>
4   #include <iostream>
5
6   struct TComplex {
7       double re, im;
8
9       constexpr TComplex(double real = 0, double imag = 0)
10          : re(real)
11          , im(imag)
12      {
13      }
14      constexpr TComplex& operator+=(const TComplex& rhs)
15      {
16          re += rhs.re;
17          im += rhs.im;
18          return *this;
19      }
20      constexpr TComplex& operator-=(const TComplex& rhs)
21      {
22          re -= rhs.re;
23          im -= rhs.im;
24          return *this;
25      }
26
27      friend constexpr TComplex operator+(const TComplex& lhs, const
            TComplex& rhs) { return TComplex(lhs) += rhs; }
28      friend constexpr TComplex operator-(const TComplex& lhs, const
            TComplex& rhs) { return TComplex(lhs) -= rhs; }
29      constexpr TComplex operator*(const TComplex& rhs) const { return { re
             * rhs.re - im * rhs.im, re * rhs.im + im * rhs.re }; }
30      constexpr TComplex operator/(const int rhs) const { return { re / rhs
```

```
        , im / rhs }; }
31      constexpr double Abs() const { return re * re + im * im; }
32  };
33
34  #endif
```

SimpleVector.hpp

```
1   #ifndef SIMPLEVECOR
2   #define SIMPLEVECOR
3
4   #include <cassert>
5   #include <iostream>
6   #include <list>
7   #include <queue>
8   #include <vector>
9
10  class SimpleVector {
11      std::vector<uint32_t> jumps;
12      std::queue<uint32_t, std::list<uint32_t>> need_compress;
13
14      uint32_t avail_mode, local_size, last;
15      uint32_t getMinMode(uint32_t num);
16      void NewElem(uint32_t elem_size);
17      void read(std::ifstream&);
18      void _write(std::ofstream& os);
19
20      uint32_t _size;
21      uint32_t _count;
22      bool modifable;
23
24  public:
25      class Iter {
26          const uint32_t* src;
27          std::vector<uint32_t> jumps;
28          uint32_t mode, pos, local_pos, local_size, last, jump_index,
                jump_count, _size;
29
30      public:
31          Iter(const SimpleVector* vect, bool end = false);
32          uint32_t operator*();
33          Iter& operator++();
34          bool operator==(const Iter& i);
35          bool operator!=(Iter const& i) { return !(*this == i); }
36      };
37
38      uint32_t* compressed;
39
40      SimpleVector();
41      SimpleVector(std::ifstream& in);
42      SimpleVector(const std::vector<uint32_t>& vect);
```

```
43      SimpleVector(const SimpleVector& vect);
44      SimpleVector(SimpleVector&& vect);
45      ~SimpleVector()
46      {
47          if (compressed)
48              delete[] compressed;
49      }
50
51      std::vector<uint32_t> to_vector() const;
52
53      void push_back(uint32_t new_elem);
54      void check();
55      bool empty();
56
57      uint32_t size() { return _count; }
58      uint32_t size_local() { return local_size; }
59      uint32_t back() { return last; }
60      Iter begin();
61      Iter cbegin() const;
62      Iter end();
63      Iter cend() const;
64
65      uint32_t operator[](uint32_t pos) const;
66
67      SimpleVector& operator=(const SimpleVector& rhs);
68      SimpleVector& operator=(SimpleVector&& rhs);
69      void write(std::ofstream& os);
70  };
71
72  #include "SimpleVector.h"
73
74  #endif //!SIMPLEVECOR
```

Shazam.hpp

```
1   #ifndef SHAZAM
2   #define SHAZAM
3
4   #include <array>
5   #include <cassert>
6   #include <fstream>
7   #include <iostream>
8   #include <map>
9
10  #include "SimpleVector.hpp"
11  #include "TComplex.hpp"
12
13  // specifically for the search function (43% faster than usual map)
14  #include <unordered_map>
15  // #define unordered_map map // (test)
16
```

```cpp
#include <mpg123.h>
#include <vector>

template <size_t LENGTH = 4096, uint32_t STEP = 1024>
class Shazam {
    using Table = std::vector<std::vector<uint32_t>>;
    // calculation of complex roots of degree n from 1
    static constexpr std::array<TComplex, LENGTH / 2> InitRoots()
    {
        std::array<TComplex, LENGTH / 2> res;
        constexpr double angle = -2 * M_PI / LENGTH;

        constexpr TComplex W(std::cos(angle), std::sin(angle));
        TComplex root(1, 0);

        for (size_t idx = 0; idx < LENGTH / 2; ++idx, root = root * W)
            res[idx] = root;

        return res;
    }
    // calculation of the Hann function values in points
    static constexpr std::array<double, LENGTH> InitHann()
    {
        std::array<double, LENGTH> res{};
        for (size_t i = 0; i < LENGTH - 1UL; ++i)
            res[i] = 0.5 * (1UL - cos(M_PI / double(LENGTH - 1UL) * 2UL *
                double(i)));
        return res;
    }

    static constexpr std::array<double, LENGTH> hann = InitHann();
    static constexpr std::array<TComplex, LENGTH / 2> roots = InitRoots()
        ;

    // boundaries of the segments of frequencies
    static constexpr uint32_t parts[5] = {
        (40 * LENGTH) / 44100,
        (300 * LENGTH) / 44100,
        (800 * LENGTH) / 44100,
        (1500 * LENGTH) / 44100,
        (2700 * LENGTH) / 44100
    };

    // lengths of segments
    static constexpr uint32_t len[4] = {
        parts[1] - parts[0],
        parts[2] - parts[1],
        parts[3] - parts[2],
        parts[4] - parts[3]
    };

```

```
66      // number of hashes
67      static constexpr size_t max_value = len[3] * len[2] * len[1] * len
            [0];
68      mpg123_handle* mh;
69
70      // table of id
71      Table SongID;
72
73      // table of time
74      Table SongTime;
75
76      uint32_t SongCount;
77
78      // song name
79      std::vector<std::pair<std::string, size_t>> Articles;
80
81      size_t LastHashCount;
82
83      // returns the index of the segment in which the point lies
84      size_t GetPos(size_t);
85      uint32_t hash(const uint32_t arr[4]);
86      void FFT(TComplex* X);
87      void DFFT(TComplex* X,
88          TComplex* X1,
89          TComplex* X2);
90
91      // returns the vector of the amplitude values for the song
92      std::vector<float> processFile(const std::string&);
93
94      // fills the tables of ID and Time
95      std::vector<uint32_t> GetHashes(const std::string& name);
96
97      std::unordered_map<uint32_t, std::unordered_map<int, uint32_t>>
            _search(const std::string& name);
98
99  public:
100     Shazam()
101         : SongID(max_value)
102         , SongTime(max_value)
103         , SongCount(0)
104     {
105         assert(LENGTH >= parts[4]
106             && len[0] && len[1] && len[2] && len[3]);
107         std::cout << max_value << std::endl;
108         assert(mpg123_init() == MPG123_OK);
109     }
110
111     ~Shazam()
112     {
113         mpg123_exit();
114     }
```

```
115
116         void append(const std::string& name);
117         void CreateBase(const std::vector<std::string>& names, size_t threads
                = 1);
118
119         // write base to file
120         void write(std::ofstream& out);
121
122         // read base from file
123         void read(std::ifstream& in);
124
125         std::string search(const std::string&);
126 };
127
128 #include "Shazam.h"
129
130 #endif //!SHAZAM
```

SimpleVector.h

```
1  #ifndef SIMPLEVECTOR_H
2  #define SIMPLEVECTOR_H
3
4  #include "SimpleVector.hpp"
5  #include <cstring>
6  #include <fstream>
7
8  static constexpr uint32_t digits[9] = {
9      28,
10     14,
11     9,
12     7,
13     5,
14     4,
15     3,
16     2,
17     1
18 };
19
20 SimpleVector::SimpleVector()
21     : jumps()
22     , need_compress()
23     , avail_mode(1)
24     , local_size(0)
25     , last(0)
26     , _size(0)
27     , _count(0)
28     , modifable(true)
29     , compressed(nullptr)
30 {
31 }
```

```
32
33 SimpleVector::SimpleVector(std::ifstream& in)
34     : SimpleVector()
35 {
36     read(in);
37     auto _jumps = SimpleVector();
38     _jumps.read(in);
39     if (_jumps._count)
40         for (auto i : _jumps)
41             jumps.push_back(i);
42 }
43
44 SimpleVector& SimpleVector::operator=(const SimpleVector& rhs)
45 {
46     _count = rhs._count;
47     _size = rhs._size;
48     local_size = rhs.local_size;
49     need_compress = rhs.need_compress;
50     avail_mode = rhs.avail_mode;
51     last = rhs.last;
52     if (rhs.compressed) {
53         compressed = new uint32_t[rhs._size];
54         std::memcpy(compressed, rhs.compressed, _count * 4);
55     } else
56         compressed = nullptr;
57     return *this;
58 }
59
60 SimpleVector& SimpleVector::operator=(SimpleVector&& rhs)
61 {
62     _count = rhs._count;
63     _size = rhs._size;
64     local_size = rhs.local_size;
65     need_compress = rhs.need_compress;
66     avail_mode = rhs.avail_mode;
67     last = rhs.last;
68     std::swap(compressed, rhs.compressed);
69     return *this;
70 }
71
72 SimpleVector::SimpleVector(const SimpleVector& rhs)
73     : jumps(rhs.jumps)
74     , need_compress(rhs.need_compress)
75     , avail_mode(rhs.avail_mode)
76     , local_size(rhs.local_size)
77     , last(rhs.last)
78     , _size(rhs._size)
79     , _count(rhs._count)
80     , modifable(true)
81     , compressed(nullptr)
82
```

```
83  {
84      if (this == &rhs)
85          return;
86      if (rhs.compressed) {
87          compressed = new uint32_t[rhs._size];
88          std::memcpy(compressed, rhs.compressed, _count * 4);
89      }
90  }
91
92  SimpleVector::SimpleVector(SimpleVector&& rhs)
93      : jumps(rhs.jumps)
94      , need_compress(rhs.need_compress)
95      , avail_mode(rhs.avail_mode)
96      , local_size(rhs.local_size)
97      , last(rhs.last)
98      , _size(rhs._size)
99      , _count(rhs._count)
100     , modifable(true)
101     , compressed(nullptr)
102 {
103     std::swap(compressed, rhs.compressed);
104 }
105
106 void SimpleVector::read(std::ifstream& in)
107 {
108     in >> _count >> local_size;
109     _size = _count;
110     if (_count) {
111         compressed = new uint32_t[_size];
112         in.get();
113         in.read(reinterpret_cast<char*>(compressed), _size * 4);
114     }
115 }
116
117 SimpleVector::SimpleVector(const std::vector<uint32_t>& vect)
118     : SimpleVector()
119 {
120     for (auto new_elem : vect)
121         push_back(new_elem);
122 }
123
124 std::vector<uint32_t> SimpleVector::to_vector() const
125 {
126     std::vector<uint32_t> res;
127     res.reserve(local_size);
128     for (auto i = this->cbegin(); i != this->cend(); ++i)
129         res.push_back(*i);
130     return res;
131 }
132
133 uint32_t SimpleVector::getMinMode(uint32_t num)
```

```
{
    assert(num < (1U << digits[0]));
    uint32_t index = 1;
    for (; index < 9 && (num < (1U << digits[index])); ++index)
        ;
    return digits[index - 1];
}

void SimpleVector::NewElem(uint32_t elem_size)
{
    uint32_t mode = 28 / elem_size;
    uint32_t hold_mode = mode;
    uint32_t res = 0;
    for (; digits[res] != elem_size; ++res)
        ;
    res = 9 - res;

    while (mode--) {
        res <<= elem_size;
        res += need_compress.empty() ? 0 : need_compress.front();
        if (!need_compress.empty())
            need_compress.pop();
    }
    res <<= 28 % hold_mode;
    if (_size < _count + 1) {
        if (!_size)
            _size = 1;
        else
            _size *= 2;
        uint32_t* temp = new uint32_t[_size];
        if (compressed) {
            std::memcpy(temp, compressed, _count * 4);
            delete[] compressed;
        }
        compressed = temp;
    }
    compressed[_count++] = res;
}

void SimpleVector::push_back(uint32_t new_elem)
{
    assert(modifable);

    uint32_t t = new_elem - last;
    bool ind = (new_elem >= last);
    if (last != new_elem)
        last = new_elem;

    if (ind)
        new_elem = t;
    else
```

```
185             jumps.push_back(local_size);
186
187         need_compress.push(new_elem);
188         local_size++;
189         uint32_t temp = getMinMode(new_elem);
190         if (temp > avail_mode) {
191             //need to compress maximum count of numbers, excluding new_elem
192             while (need_compress.size() >= 28 / temp) {
193                 if (14 < need_compress.size())
194                     avail_mode = 2;
195                 else if (9 < need_compress.size())
196                     avail_mode = 3;
197                 else if (7 < need_compress.size())
198                     avail_mode = 4;
199                 else if (5 < need_compress.size())
200                     avail_mode = 5;
201                 else if (4 < need_compress.size())
202                     avail_mode = 7;
203                 else if (3 < need_compress.size())
204                     avail_mode = 9;
205                 else if (2 < need_compress.size())
206                     avail_mode = 14;
207                 else
208                     avail_mode = 28;
209                 NewElem(avail_mode);
210             }
211             avail_mode = need_compress.empty() ? 1 : temp;
212         }
213         if (need_compress.size() >= 28 / avail_mode) {
214             NewElem(avail_mode);
215             uint32_t max = 0;
216             std::queue<uint32_t, std::list<uint32_t>> trash;
217             while (!need_compress.empty()) {
218                 uint32_t val = need_compress.front();
219                 need_compress.pop();
220                 if (val > max)
221                     max = val;
222                 trash.push(val);
223             }
224             need_compress = trash;
225             avail_mode = getMinMode(max);
226         }
227 }
228
229 void SimpleVector::check()
230 {
231         if (!need_compress.empty())
232             NewElem(avail_mode);
233         modifable = false;
234 }
235
```

```
236 bool SimpleVector::empty()
237 {
238     check();
239     return local_size == 0;
240 }
241
242 SimpleVector::Iter SimpleVector::begin()
243 {
244     check();
245     return Iter(this, local_size == 0);
246 }
247
248 SimpleVector::Iter SimpleVector::cbegin() const
249 {
250     return Iter(this, local_size == 0);
251 }
252
253 SimpleVector::Iter SimpleVector::end()
254 {
255     check();
256     return Iter(this, true);
257 }
258
259
260 SimpleVector::Iter SimpleVector::cend() const
261 {
262     return Iter(this, true);
263 }
264
265 uint32_t SimpleVector::operator[](uint32_t pos) const
266 {
267     assert(pos < local_size && need_compress.empty());
268     auto it = Iter(this, local_size == 0);
269     while (pos--)
270         ++it;
271     return *it;
272 }
273
274 void SimpleVector::_write(std::ofstream& of)
275 {
276     check();
277     of << _count << ' ' << local_size << '\n';
278     if (compressed) {
279         of.write(reinterpret_cast<char*>(compressed), _count * 4);
280         of << '\n';
281     }
282 }
283
284 void SimpleVector::write(std::ofstream& of)
285 {
286     _write(of);
```

```
287     SimpleVector(jumps)._write(of);
288 }
289
290 SimpleVector::Iter::Iter(const SimpleVector* vect, bool end)
291     : src(vect->compressed)
292     , jumps(vect->jumps)
293     , mode(0)
294     , pos(0)
295     , local_pos(0)
296     , local_size(end ? 0 : vect->local_size)
297     , last(0)
298     , jump_index(0)
299     , jump_count(0)
300     , _size(vect->_count)
301 {
302     if (!end && src)
303         mode = digits[9 - (src[0] >> 28)];
304     else
305         pos = _size;
306 }
307
308 uint32_t SimpleVector::Iter::operator*()
309 {
310     if (pos == _size)
311         return 0;
312     return last + ((src[pos] >> (28 - (local_pos + 1) * mode)) & (-1U >>
        (32 - mode)));
313 }
314
315 SimpleVector::Iter& SimpleVector::Iter::operator++()
316 {
317     if (jumps.size() > jump_count && jumps[jump_count] == ++jump_index) {
318         last = 0;
319         ++jump_count;
320     } else
321         last = **this;
322     if (!local_size--)
323         return *this;
324
325     if (++local_pos >= 28 / mode) {
326         local_pos = 0;
327         if (++pos == _size)
328             return *this;
329         mode = digits[9 - (src[pos] >> 28)];
330     }
331
332     return *this;
333 }
334
335 bool SimpleVector::Iter::operator==(const Iter& i)
336 {
```

```
337     return (local_size == 0 && i.local_size == 0)
338         || ((i.src == src)
339                 && pos == i.pos
340                 && local_pos == i.local_pos);
341 }
342
343 #endif /* ifndef SIMPLEVECTOR_H */
```

Shazam.h

```
1  #ifndef SHAZAM_H
2  #define SHAZAM_H
3
4  #include "Shazam.hpp"
5  #include <future>
6  #include <mutex>
7
8  template <size_t LENGTH, uint32_t STEP>
9  size_t Shazam<LENGTH, STEP>::GetPos(size_t num)
10 {
11     assert(num < parts[4]);
12     if (num < parts[1] && num >= parts[0])
13         return 0;
14     if (num < parts[2] && num >= parts[1])
15         return 1;
16     if (num < parts[3] && num >= parts[2])
17         return 2;
18     if (num < parts[4] && num >= parts[3])
19         return 3;
20     return -1UL;
21 }
22
23 template <size_t LENGTH, uint32_t STEP>
24 uint32_t Shazam<LENGTH, STEP>::hash(const uint32_t arr[4])
25 {
26     return (arr[0] - parts[0])
27         + (arr[1] - parts[1]) * len[0]
28         + (arr[2] - parts[2]) * len[1] * len[0]
29         + (arr[3] - parts[3]) * len[2] * len[1] * len[0];
30 }
31
32 template <size_t LENGTH, uint32_t STEP>
33 void Shazam<LENGTH, STEP>::DFFT(TComplex* X,
34     TComplex* X1,
35     TComplex* X2)
36 {
37     FFT(X);
38     constexpr size_t N_2 = LENGTH / 2;
39     X1[0].re = X[0].re;
40     X2[0].re = X[0].im;
41     X1[N_2].re = X[N_2].re;
```

```
42      X2[N_2].re = X[N_2].im;
43      X2[0].im = X1[0].im = X1[N_2].im = X2[N_2].im = 0;
44      for (size_t k = 1; k < N_2; ++k) {
45          X1[k] = TComplex((X[k].re + X[LENGTH - k].re), (X[k].im - X[
                LENGTH - k].im)) / 2;
46          X2[k] = TComplex((X[k].im + X[LENGTH - k].im), (X[LENGTH - k].re
                - X[k].re)) / 2;
47          X1[LENGTH - k] = { X1[k].re, -X1[k].im };
48          X2[LENGTH - k] = { X2[k].re, -X2[k].im };
49      }
50  }
51
52  template <size_t LENGTH, uint32_t STEP>
53  void Shazam<LENGTH, STEP>::FFT(TComplex* X)
54  {
55      for (size_t i = 1, j = 0; i < LENGTH; ++i) {
56          size_t bit = LENGTH >> 1;
57          for (; j >= bit; bit >>= 1)
58              j -= bit;
59          j += bit;
60          if (i < j)
61              std::swap(X[i], X[j]);
62      }
63
64      for (size_t ln = 2; ln <= LENGTH; ln <<= 1) {
65          for (size_t i = 0; i < LENGTH; i += ln) {
66              for (size_t j = 0; j < ln / 2; ++j) {
67                  TComplex u = X[i + j], v = X[i + j + ln / 2] * roots[
                        LENGTH / ln * j];
68                  X[i + j] = u + v;
69                  X[i + j + ln / 2] = u - v;
70              }
71          }
72      }
73  }
74
75  template <size_t LENGTH, uint32_t STEP>
76  std::vector<float> Shazam<LENGTH, STEP>::processFile(const std::string&
        name)
77  {
78      auto mh = mpg123_new(NULL, NULL);
79      assert(mh != NULL);
80      assert(mpg123_param(mh, MPG123_FLAGS, MPG123_MONO_MIX | MPG123_QUIET
            | MPG123_FORCE_FLOAT, 0.) == MPG123_OK);
81
82      auto errorCode = mpg123_open(mh, name.c_str());
83      assert(errorCode == MPG123_OK);
84      long rate;
85      int channels, encoding;
86      errorCode = mpg123_getformat(mh, &rate, &channels, &encoding);
87      assert(errorCode == MPG123_OK);
```

```
88      if (rate != 44100) {
89          mpg123_delete(mh);
90          return {};
91      }
92      const size_t part_size = 1024;
93      unsigned char part[part_size];
94      size_t bytesRead;
95      std::vector<float> samples;
96      size_t bytesProcessed = 0;
97      do {
98          int err = mpg123_read(mh, part, part_size, &bytesRead);
99          samples.resize((bytesProcessed + bytesRead) / 4 + 1);
100         memcpy(reinterpret_cast<unsigned char*>(samples.data()) +
                bytesProcessed, part, bytesRead);
101         bytesProcessed += bytesRead;
102         if (err == MPG123_DONE)
103             break;
104
105         assert(err == MPG123_OK);
106     } while (bytesRead > 0);
107     samples.resize(bytesProcessed / 4);
108     errorCode = mpg123_close(mh);
109     assert(errorCode == MPG123_OK);
110     mpg123_delete(mh);
111     return samples;
112 }
113
114 std::mutex mutex;
115
116 template <size_t LENGTH, uint32_t STEP>
117 std::vector<uint32_t> Shazam<LENGTH, STEP>::GetHashes(const std::string&
        name)
118 {
119     const std::vector<float> samples = processFile(name);
120
121     std::vector<uint32_t> res;
122     res.reserve(samples.size() / STEP);
123     uint32_t start = 0;
124     for (; start < samples.size(); start += 2 * STEP) {
125         TComplex first[LENGTH],
126             second[LENGTH],
127             tmp[LENGTH];
128         size_t end = start + LENGTH;
129         for (size_t i = start, idx = 0; i < end; ++i, ++idx)
130             tmp[idx] = (i >= samples.size())
131                 ? TComplex{ 0, 0 }
132                 : TComplex{ samples[i] * hann[idx], (i + STEP < samples.
                        size()) ? (samples[i + STEP] * hann[idx]) : 0 };
133
134         DFFT(tmp, first, second);
135         double max1 = -1,
```

```
136                 max2 = -1,
137                 temp;
138         uint32_t pos1[] = { parts[0], parts[1], parts[2], parts[3] },
139                  pos2[] = { parts[0], parts[1], parts[2], parts[3] };
140         size_t pos;
141         for (uint32_t i = parts[0]; i < parts[4]; ++i) {
142             if (pos = GetPos(i), pos != GetPos(i - 1)) {
143                 max1 = -1;
144                 max2 = -1;
145             }
146             if (temp = first[i].Abs(), temp > max1) {
147                 max1 = temp;
148                 pos1[pos] = i;
149             }
150             if (temp = second[i].Abs(), temp > max2) {
151                 max2 = temp;
152                 pos2[pos] = i;
153             }
154         }
155         res.push_back(hash(pos1));
156         res.push_back(hash(pos2));
157     }
158
159     return res;
160 }
161
162 template <size_t LENGTH, uint32_t STEP>
163 void Shazam<LENGTH, STEP>::append(const std::string& name)
164 {
165     const std::vector<uint32_t> hashes = GetHashes(name);
166
167     mutex.lock();
168
169     std::cout << "process file \"" << name << "\" \n";
170     uint32_t count = -1U;
171     for (auto i : hashes) {
172         SongID[i].push_back(SongCount);
173         SongTime[i].push_back(++count);
174     }
175     std::cout << "Song count: " << ++SongCount << "\thash count: " <<
        hashes.size() << '\n';
176     Articles.push_back({ name, hashes.size() });
177
178     mutex.unlock();
179 }
180
181 template <size_t LENGTH, uint32_t STEP>
182 void Shazam<LENGTH, STEP>::CreateBase(const std::vector<std::string>&
    names, size_t threads)
183 {
184     std::vector<std::future<void>> run(threads);
```

```
185     for (size_t i = 0; i < names.size(); i += threads)
186         for (size_t it = 0; it < threads && it + i < names.size(); ++it)
187             run[it] = std::async(&Shazam<LENGTH, STEP>::append, this,
                    names[i + it]);
188 }
189
190 template <size_t LENGTH, uint32_t STEP>
191 void Shazam<LENGTH, STEP>::write(std::ofstream& out)
192 {
193     std::cout << "prepare to write...\n";
194     out << LENGTH << ' ' << STEP << ' ' << max_value << '\n';
195     size_t count = -1UL;
196     for (auto i : SongID) {
197         ++count;
198         if (i.empty())
199             continue;
200         out << count << ' ';
201         SimpleVector(i).write(out);
202     }
203     count = -1UL;
204     for (auto i : SongTime) {
205         ++count;
206         if (i.empty())
207             continue;
208         out << count << ' ';
209         SimpleVector(i).write(out);
210     }
211     out << "0\n"
212         << Articles.size() << '\n';
213     for (auto i : Articles)
214         out << i.second << '\n'
215             << i.first << '\n';
216     std::cout << "WRITE!!!\n";
217 }
218
219 template <size_t LENGTH, uint32_t STEP>
220 void Shazam<LENGTH, STEP>::read(std::ifstream& in)
221 {
222     size_t len, step, count;
223     std::cout << "prepare to read...\n";
224     in >> len >> step >> count;
225     assert(len == LENGTH
226         && step == STEP
227         && count == max_value);
228     size_t length = 0, compressed = 0;
229     size_t h, h_prev = 0;
230     while (in >> h, h > h_prev || !h_prev) {
231         auto s = SimpleVector(in);
232         SongID[h] = s.to_vector();
233         length += SongID[h].size();
234         compressed += s.size();
```

```
235             h_prev = h;
236         }
237
238         h_prev = 0;
239         do {
240             auto s = SimpleVector(in);
241             SongTime[h] = s.to_vector();
242             length += SongTime[h].size();
243             compressed += s.size();
244             h_prev = h;
245         } while (in >> h, h > h_prev || !h_prev);
246
247         in >> SongCount;
248         Articles.reserve(SongCount);
249         std::string line;
250         step = SongCount;
251         while (step--) {
252             in >> len;
253             in.get();
254             std::getline(in, line);
255             Articles.push_back({ line, len });
256         }
257         std::cout << length << "\nREAD!!!\n"
258                   << compressed << '\n';
259 }
260
261 template <size_t LENGTH, uint32_t STEP>
262 std::unordered_map<uint32_t, std::unordered_map<int, uint32_t>> Shazam<
        LENGTH, STEP>::_search(const std::string& name)
263 {
264         // map<SongID, map<TimeOffset, Count>>
265         std::unordered_map<uint32_t, std::unordered_map<int, uint32_t>> res;
266         std::cout << "process file \"" << name << "\" \n";
267         std::vector<uint32_t> hashes = GetHashes(name);
268         LastHashCount = hashes.size();
269         std::cout << "hash count: " << LastHashCount << '\n';
270
271         for (uint32_t i = 0; i != hashes.size(); ++i) {
272             for (auto iter = SongTime[hashes[i]].begin(), idx = SongID[hashes
                    [i]].begin();
273                  iter != SongTime[hashes[i]].end();
274                  ++iter, ++idx)
275                 res[*idx][int(*iter) - int(i)]++;
276         }
277         return res;
278 }
279
280 template <size_t LENGTH, uint32_t STEP>
281 std::string Shazam<LENGTH, STEP>::search(const std::string& name)
282 {
283         auto base = _search(name);
```

```
284     // map<MatchPercent, pair<SongID, TimeOffset>>
285     std::map<double, std::pair<uint32_t, int>> results;
286     for (auto id : base) {
287         int offset = 0;
288         double max = 0;
289         for (auto i : id.second)
290             if (double(i.second) > max) {
291                 max = double(i.second);
292                 offset = i.first;
293             }
294         max *= 100.0 / double(LastHashCount);
295         if (max < 101 && max > 2)
296             results[max] = { id.first, offset };
297     }
298
299 // #ifdef DEBUG
300     std::cout.precision(2);
301
302     for (auto i : results)
303         std::cout << "match: " << std::fixed << i.first << "%\t"
304                   << "name: \"" << Articles[i.second.first].first << "\"
                      "
305                   << "time offset: " << double(i.second.second) / 44100 *
                       STEP << '\n';
306     std::cout << std::endl;
307 // #endif
308
309     auto temp = results.rbegin();
310     return (!results.empty() && temp->first > 10) ? Articles[temp->second
        .first].first : "! NOT FOUND";
311 }
312
313 #endif /* ifndef SHAZAM_H */
```

Тестирование

Известно, что формат записи mp3 не хранит данные несжатой форме, следовательно на извлечение данных требуется некоторое время.

При помощи утилиты gprof было выделено соотношение — получение хешей из вектора семплов примерно в 400 раз медленнее чем получение самих семплов.

Тестовый запуск:

search.txt

```
1 pripev1.mp3
2 pripev1_exo.mp3
3 pripev2.mp3
4 life_line.mp3
5 heart.mp3
```

```
./kp index --input search.txt --output search.base
8138520
process file "pripev1.mp3"
Song count: 1   hash count: 664
process file "pripev1_exo.mp3"
Song count: 2   hash count: 666
process file "pripev2.mp3"
Song count: 3   hash count: 858
process file "life_line.mp3"
Song count: 4   hash count: 7850
process file "heart.mp3"
Song count: 5   hash count: 10768
prepare to write...
WRITE!!!
```

```
./kp search --index search.base  --input search.txt --output outt
8138520
prepare to read...
READ!!!
process file "pripev1.mp3"
hash count: 664
process file "pripev1_exo.mp3"
hash count: 666
process file "pripev2.mp3"
hash count: 858
process file "life_line.mp3"
hash count: 7850
process file "heart.mp3"
hash count: 10768
```

outt

```
1 pripev1.mp3
2 pripev1_exo.mp3
3 pripev2.mp3
4 life_line.mp3
5 heart.mp3
```

Как можно увидеть, все аудиофайлы были найдены.

Изначально составление базы песен просиходило в 1 поток и при поиске песни создавалась ещё одна база данных для записи её хешей. Также число самих хешей было достатоно небольшим (порядка 2000), что очень негативно отразилось на скорости поиска.

Запуск старой версии программы на файле с 13 файлами средней длительностью 5 минут:

```
./kp index --input ss  --output ss.base  8,83s user 0,50s system 99% cpu 9,337 total
./kp search  --index ss.base --input ss  --output outt  13,43s user 0,36s system 99% cpu 13,807 total
```

Запуск конечной версии программы:

```
./kp index --input ss  --output ss.base  14,19s user 0,82s system 531% cpu 2,825 total
./kp search  --index ss.base --input ss  --output outt  9,36s user 0,58s system 99% cpu 9,959 total
```

На больших файлах разница в скорости ещё больше в пользу конечного варианта. База из около 1000 песен обрабатывалась первым варантом 15 минут, конечная же программа справилась за 5.

Поиск же ускорился ещё сильнее, в основном за счёт равномерного распределения хешей по таблице (10000 хешей на песню и около 8 000 000 хешей на всю базу)

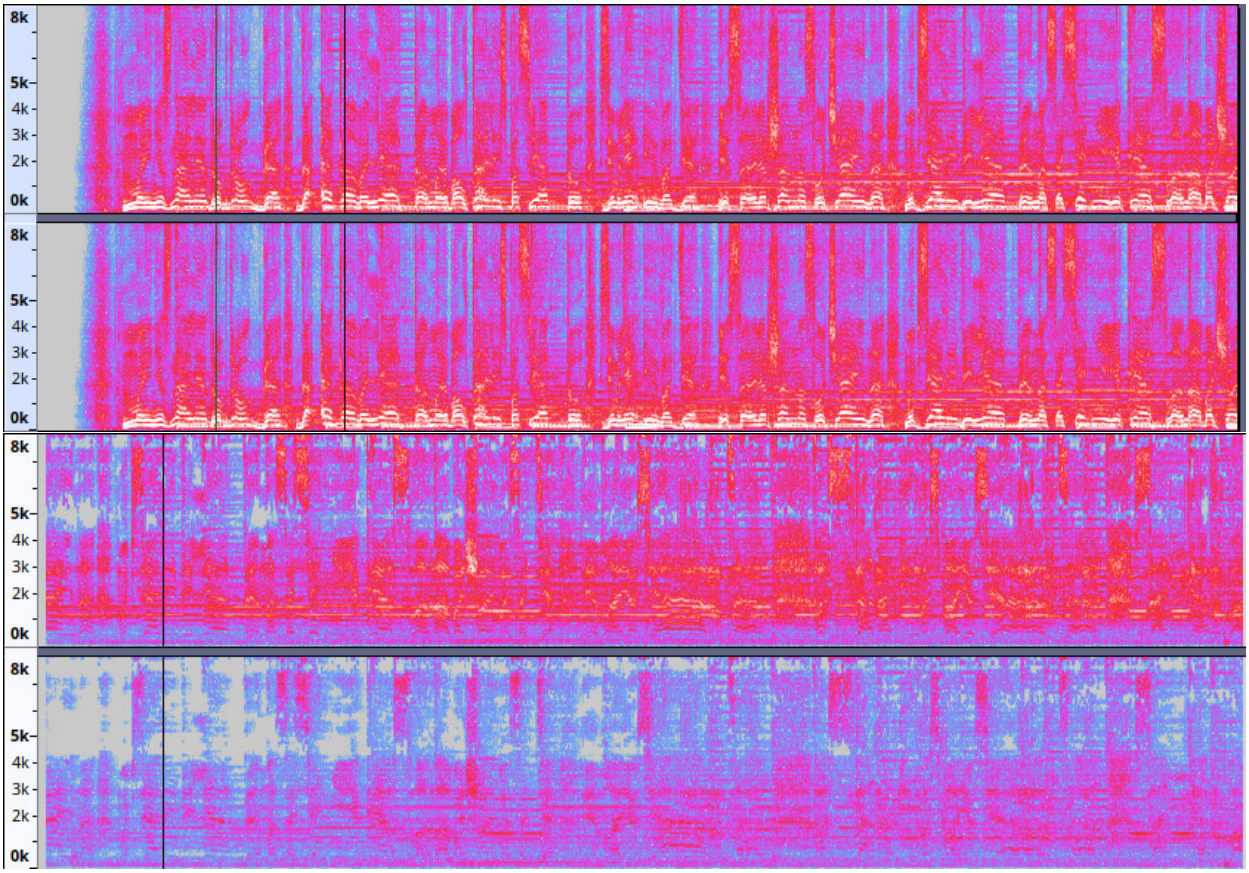
Немного об эффективности simple9: для базы из 1000 песен получилось 16782820 элементов на каждую из таблиц. Учитывая, что это переменные типа int, можно посчитать сколько это будет занимать в памяти: $16782820 * 4/1024/1024 \approx 64$ МБ

После обработки с помощью simple9 размер изменился всего до 9206818 элементов, или 35 мегабайт.

Файлы с сильными шумами (запись части песни на микрофон телефона) тоже находятся, но процент абсолютного совпадения по хешам очень мал (около 2-3%), хотя и отличается от случайных совпадений (менее 0.5%)

```
process file /dev/shm/1.mp3
hash count: 878
match: 0.11%    name: /home/lol/Music/Knяzz/2005_-_Любовь_негодяя/13._Летучий_скелетик.mp3
    time offset: 125.29
match: 0.23%    name: /home/lol/Music/KiSh/Альбомы/1999_-
    _Акустический_Альбом_(четвёртое_издание_OPT-Рекордс,_1999)/05._Девушка_и_Граф.mp3 time
    offset: 176.01
match: 0.34%    name: /home/lol/Music/Сплин/+ Альбомы/1997 - Фонарь под глазом/02. Я не хочу
    домой.mp3 time offset: 11.94
match: 0.46%    name: /home/lol/Music/Ария/Albums/2011_-_Феникс_[Soyuz,_SZCD_7354-
    11,_Russia]/08._Аттила.mp3 time offset: 54.24
match: 2.73%    name: /home/lol/Music/Сплин/+ Альбомы/2009 - Сигнал из Космоса/12. Человек не
    спал.mp3 time offset: 25.10
```

В основном это происходит из за моего выбора частотных диапазонов и из за несовершенства записывающей аппаратуры (мой микрофон практически не записывает низкие частоты). На представленной ниже спектрограмме можно это увидеть:



Выше находится оригинальная спектрограмма, снизу её зашумлённый отрезок. Частот ниже 1 000 Герц практически нет, а учитывая что 3 из 4-х отрезков, для которых я ищу максимумы находятся как раз в этом промежутке поиск совпадений представляется сомнительным и совпадение в 2.73% уже настоящее чудо.

Выводы

Я взяла такую тему для курсового проекта ещё не зная про преобразование фурье и тем более про simple9. Мне сказали что будет сложно, но интересно. Конечно же меня обманули.

Ничего сложного для понимания или реализации я не встретила, преобразование фурье довольно часто используется и было несложно найти материалы, в отличие от simple9. Он не спроста содержит простоту в своём названии — найти в нём сложные места довольно сложно, однако реализация оказалась довольно муторной и громоздкой. Конечно было бы эффективнее использовать любой другой алгоритм сжатия, но и simple9 отлично справляется со своей задачей.

Фурье тоже может оказаться интересным, особенно если считать 2 фурье за одно. Конечно нерекурсивная реализация может взорвать мозг своими перестановками, но это того стоит.

Конечно же не удалось достичь производительности оригинального шазама (со скрипом проходят файлы, которые были намеренно зашумлены), получилась лишь сильно упрощённая версия, отражающая основную суть.