# Курсовой проект по курсу дискретного анализа: Аудиопоиск

Выполнил студент группы М8О-308Б-20 *Ядров Артем*.

## Условие

Реализовать систему для поиска аудиозаписи по небольшому отрывку.
Задача состоит в применении быстрого преобразования Фурье для получения из списка частот списка амплитуд. Уникальность аудиозаписи дают частоты с максимальными амплитудами. Однако, в диапазон частот может варьироваться от частоты $32.70Hz$ до $4186.01Hz$. Поэтому необходимо разделить частоты на интервалы. По максимум частот в каждом интервале можно идентифицировать отрезок аудиозаписи.

## Метод решения

Чтобы преобразовать аналоговый сигнал в цифровой используется дискретизация, состоящая из двух этапов: дискретизации по времени и дискретизации по амплитуде.

Дискретизация по времени означает, что сигнал представляется последовательностью отсчетов ($Samples$), взятых через равные промежутки времени.
Например, если частота дискретизации равна 44100, то это значит, что сигнал измеряется 441000 раз в секунду.

Чем больше частота, тем точнее цифровой сигнал соответствует аналоговому. Однако с возрастанием частоты растет и память, необходимая для хранения сигнала, также усложнится процесс обработки за счет увеличения количества операций.

Однако, какую частоту выбрать для того, чтобы цифровой сигнал был достаточно точным и занимал как можно меньше памяти? Нас спасет теорема Котельникова, которая гласит, что для того, чтобы аналоговый (непрерывный по времени) сигнал можно было восстановить по его отсчетам, частота дискретизации должна быть как минимум вдвое больше максимальной звуковой частоты сигнала.

Так как человек слышит в диапазоне $[20, 20'000]Hz$, то частота дискретизации должна быть минимум $40'000Hz$.

Сегодня самыми популярными частотами являются $44,1kHz(CD), 48kHz(DAT)$.

Для получения списка отсчетов мною была использована библиотека $mpg123$. Далее полученный вектор обрабатывается дискретным преобразованием Фурье по частям. В каждой части искался максимум по диапазонам: $0-40Hz, 40-80Hz, 80-120Hz, 120-180Hz, 180-300Hz, 300+Hz$

Для общего хранения вычислялся хэш, идентифицирующий данный участок аудиозаписи. Хэш хранится в $std :: unordered\_map$ в качестве ключа, а в роли значения выступает $std :: vector < std :: pair >$, в котором хранится $ID$ аудиозаписи и номер блока.

Отмечу, что в силу того, мы последовательно обрабатываем одну аудиозапись, то в рамках одной аудиозаписи все элементы вектора будут отсортированы по времени (номеру блока). Это позволяет использовать бинарный поиск при необходимости.

Для того, чтобы идентифицировать аудиозапись, можно запустить поиск вычисленных хэш-тегов в "базе данных". Однако не всё так просто. Дело в том, что у многих различных фрагментов произведений хэш-теги совпадают. Поэтому если взять отметку $i_1, i_2$ в образце и взять отметки $j_1, j_2$, то наличие двух совпадений будет обусловлено следующим условием:

$Hash(i_1) = Hash(j_1)\ AND\ Hash(i_2) = Hash(j_2)\ AND\ abs(i_1 - j_1) = abs(i_2 - j_2)$.

На великом и могучем это будет звучать так: два фрагмента образца $i_1, i_2$ соответствуют двум фрагментам аудиозаписи $j_1, j_2$ соответственно, если совпадают хэши соответствующих элементов и совпадает временной диапазон в образце и в аудиозаписи.

## Исходный код

Немного про собственно систему:

- При конструкции объекта происходит чтение базы данных, хранящейся в файле (путь указан в заголовочном файле $Definitions.h$). Если файла не существует, то происходит процесс инициализации базы данных: добавлении всех $.mp3$ файлов в базу. Так как условное количество файлов может быть очень большим, то я распараллелил этот процесс.

- Деструктор объекта вызывает запись добавленных в базу аудиозаписей в файл. Если добавлять нечего (отсутствуют новые обработанные аудиозаписи), то, соответственно, ничего не записывается.

- Поиск выдает все потенциальные совпадения (если хотя бы один хэш совпал, то аудиозапись попадет в результат), отсортированные по проценту совпадений. Поэтому если поиск выдал неправильный результат, пользователь может найти свою искомую песню среди других песен.

---

```
#ifndef KP_DEFINITIONS_H
#define KP_DEFINITIONS_H

#include <iostream>
#include <vector>
#include <array>
#include <cassert>
#include <complex>
#include <cstring>
```

```cpp
#include <mpg123.h>

#define all(v) v.begin(), v.end()

typedef uint64_t Size;
typedef std::complex<double> Complex;

const Size BLOCK_SIZE = 4096;
const Size RANGE_NUM = 5;
const Size RANGES[RANGE_NUM] = {40, 80, 120, 180, 300};
const Size RANGE_MIN = 40;
const Size RANGE_MAX = 300;
const Size HASH_CONST = 2;
const std::string LIBRARY_PATH = std::string("./LIBRARY/");
const std::string LIB_FILE = std::string("./LIBRARY/LIB_FILE.txt");
const long RATE = 44100;
#endif //KP_DEFINITIONS_H
```

---

```cpp
#ifndef KP_SHAZAM_H
#define KP_SHAZAM_H

#include <chrono>
#include <mutex>
#include <thread>
#include <unordered_map>

#include "Definitions.h"

class Shazam {
public:
    Shazam();

    ~Shazam();

    void append(const std::string &fileName);

    void write(std::ofstream &out);

    void read(std::ifstream &input);
```

```
    std::vector<std::pair<std::string, double>> search(const std::string
↪   &name);

    static std::vector<Complex> initRoots();

    static std::vector<double> HannWindow();

private:
    void initDB();

    std::vector<float> processFile(const std::string &fileName);

    std::vector<Size> getHashes(const std::string &fileName);

    static std::vector<std::string> getSongs();

    static void FFT(std::vector<Complex> *arr);

    static Size getInd(Size freq);

    static Size hash(const std::array<Size, RANGE_NUM> &freq);

    static std::vector<Complex> roots_;
    static std::vector<double> hann_;
    std::mutex mutex_;

    std::unordered_map<Size, std::vector<std::pair<Size, Size>>> database_;
↪   // [hash, [id, block_num]]
    /* для каждого хэша хранится вектор из пар ID песни, номер блока*/
    std::unordered_map<Size, std::string> newSongs_;
    std::vector<std::string> songs_;

    static Size diff(const Size &left, const Size &right);
};


#endif //KP_SHAZAM_H
```

---

```
#include "Shazam.h"
```

```cpp
#include <filesystem>
#include <fstream>
#include <algorithm>

std::vector<Complex> Shazam::roots_ = Shazam::initRoots();
std::vector<double> Shazam::hann_ = Shazam::HannWindow();

void threadFunc(Shazam *shazam, Size start, Size count,
↪  std::vector<std::string> *songs) {
    for (Size i = 0; i < count; ++i) {
        shazam->append((*songs)[start + i]);
    }
}

Shazam::Shazam() {
    std::ifstream in(LIB_FILE);
    if (!in.is_open()) {
        std::cout << "CAN NOT OPEN FILE: " << LIB_FILE << std::endl;
        perror(std::string("CAN NOT OPEN INPUT FILE:
↪  ").append(LIB_FILE).c_str());
        initDB();
        return;
    }
    read(in);
    in.close();
}

std::vector<Complex> Shazam::initRoots() {
    std::vector<Complex> roots(BLOCK_SIZE / 2);
    double angle = 2 * M_PI / BLOCK_SIZE;
    Complex W(std::cos(angle), std::sin(angle));
    Complex root(1, 0);
    for (int i = 0; i < BLOCK_SIZE / 2; ++i, root *= W) {
        roots[i] = root;
    }
    return roots;
}

std::vector<double> Shazam::HannWindow() {
    std::vector<double> Hann(BLOCK_SIZE);
    for (int i = 0; i < BLOCK_SIZE; ++i) {
        Hann[i] = 0.5 * (1UL - std::cos(2UL * (double) M_PI * i /
↪  (BLOCK_SIZE - 1)));
```

```
    }
    return Hann;
}

std::vector<float> Shazam::processFile(const std::string &fileName) {
    auto handle = mpg123_new(nullptr, nullptr);
    assert(handle != nullptr);
    assert(mpg123_param(handle, MPG123_FLAGS, MPG123_MONO_MIX | MPG123_QUIET
↪   | MPG123_FORCE_FLOAT, 0.) == MPG123_OK);
    assert(mpg123_open(handle, fileName.c_str()) == MPG123_OK);
    long rate;
    int channels, encoding;
//    assert(mpg123_getformat(handle, &rate, &channels, &encoding) ==
↪   MPG123_OK);
    if (mpg123_getformat(handle, &rate, &channels, &encoding) != MPG123_OK)
↪   {
        std::cout << "CAN NOT GET FORMAT" << std::endl;
    }
    if (rate != RATE) {
        std::cout << "ERROR: FILE: " << fileName << " WRONG RATE: " << rate
↪   << std::endl;
        return {};
    }
    const int partSize = 1024;
    unsigned char part[partSize];
    Size bytesRead = 0;
    Size bytesProcessed = 0;
    std::vector<float> arr(partSize / sizeof(float));
    do {
        int err = mpg123_read(handle, part, partSize, &bytesRead);
        arr.resize((bytesRead + bytesProcessed) / sizeof(float) + 1);
        memcpy((unsigned char *) arr.data() + bytesProcessed, part,
↪   bytesRead);
        bytesProcessed += bytesRead;
        if (err == MPG123_DONE) {
            break;
        }
        assert(err == MPG123_OK);
    } while (bytesRead);
    arr.resize(bytesProcessed / sizeof(float));
    int err = mpg123_close(handle);
    assert(err == MPG123_OK);
```

```
    mpg123_delete(handle);
    return arr;
}

std::vector<Size> Shazam::getHashes(const std::string &fileName) {
    using namespace std::chrono;
    auto start = system_clock::now();
    std::vector<float> samples = processFile(fileName);
    std::vector<Size> res;
    Size blocks = samples.size() / BLOCK_SIZE;
    res.reserve(blocks);
    for (size_t block = 0; block < blocks; ++block) {
        std::vector<Complex> _complex(BLOCK_SIZE);
        for (int i = 0; i < BLOCK_SIZE; ++i) {
            _complex[i] = Complex(samples[block * BLOCK_SIZE + i]) *
↪   hann_[i];
        }
        FFT(&_complex);
        std::array highVal{-1., -1., -1., -1., -1.};
        std::array<Size, RANGE_NUM> maxFreq = {0, 0, 0, 0, 0};
        for (Size freq = RANGE_MIN; freq < RANGE_MAX; ++freq) {
            double intensity = std::log(std::abs(_complex[freq]) + 1.);
            Size id = getInd(freq);
            if (intensity > highVal[id]) {
                highVal[id] = intensity;
                maxFreq[id] = freq;
            }
        }
        res.emplace_back(hash(maxFreq));
    }
/*    auto end = system_clock::now();
    std::cout << "GET HASHES TIME: " << fileName << std::endl <<  end -
↪   start << std::endl;*/
    return res;
}

void Shazam::FFT(std::vector<Complex> *a) {
    int n = (int) a->size();
    if (n == 1) return;

    std::vector<Complex> a0(n / 2), a1(n / 2);
    for (int i = 0, j = 0; i < n; i += 2, ++j) {
```

```
        a0[j] = (*a)[i];
        a1[j] = (*a)[i + 1];
    }
    FFT(&a0);
    FFT(&a1);

    double ang = 2 * M_PI / n;
    Complex w(1), wn(std::cos(ang), std::sin(ang));
    for (int i = 0; i < n / 2; ++i) {
        (*a)[i] = a0[i] + w * a1[i];
        (*a)[i + n / 2] = a0[i] - w * a1[i];
        w *= wn;
    }
}

Size Shazam::getInd(Size freq) {
    if (freq < RANGE_MIN) {
        return 0;
    }
    if (freq > RANGE_MAX) {
        return RANGE_NUM - 1;
    }
    for (int i = 0; i < RANGE_NUM; ++i) {
        if (freq < RANGES[i]) {
            return i;
        }
    }
    return RANGE_NUM - 1;
}

Size Shazam::hash(const std::array<Size, RANGE_NUM> &freq) {
    Size res = freq[0] - (freq[0] % HASH_CONST);
    res += (freq[1] - (freq[1] % HASH_CONST)) * 100;
    res += (freq[2] - (freq[2] % HASH_CONST)) * 100000;
    res += (freq[3] - (freq[3] % HASH_CONST)) * 100000000;
    return res;
}

void Shazam::append(const std::string &fileName) {
    using namespace std::chrono;
    std::string name = std::filesystem::path(fileName).filename();
    auto time = system_clock::to_time_t(system_clock::now());
```

```
    std::cout << std::ctime(&time) << "APPEND FILE: " << name << " START" <<
↪   std::endl;
    auto hashes = getHashes(fileName);
    std::lock_guard<std::mutex> lock(mutex_);
    Size id = songs_.size();
    songs_.emplace_back(name);
    newSongs_[id] = name;
    for (Size i = 0; i < hashes.size(); ++i) {
        database_[hashes[i]].emplace_back(std::make_pair(id, i));
    }
    time = system_clock::to_time_t(system_clock::now());
    std::cout << std::ctime(&time) << "APPEND FILE: " << name << " END" <<
↪   std::endl;
}


void Shazam::initDB() {
    auto songs = getSongs();
    using namespace std::chrono;
    auto time = system_clock::to_time_t(system_clock::now());
    std::cout << std::ctime(&time) << "FOUND " << songs.size() << " FILES"
↪   << std::endl;
    Size threadsCount = std::thread::hardware_concurrency();
    Size songsPerThread = songs.size() / threadsCount;
    Size ost = songs.size() % threadsCount;
    std::vector<std::thread> threads;
    Size start = 0;
    for (Size i = 0; i < threadsCount; ++i) {
        threads.emplace_back(threadFunc, this, start, songsPerThread + (i <
↪   ost), &songs);
        start += songsPerThread + (i < ost);
    }
    for (Size i = 0; i < threadsCount; ++i) {
        threads[i].join();
    }
}

std::vector<std::string> Shazam::getSongs() {
    std::vector<std::string> res;
    for (const auto &file:
↪   std::filesystem::directory_iterator(LIBRARY_PATH)) {
        if (file.path().extension() == ".mp3") {
```

```
            res.emplace_back(file.path());
        }
    }
    return res;
}

void Shazam::read(std::ifstream &input) {
    std::string name;
    Size vecSize, hash, blockNum, id = 0;
    std::vector<std::string> songs;
    std::unordered_map<Size, std::vector<std::pair<Size, Size>>> database;
    while (std::getline(input, name)) {
        if (name.empty()) {
            continue;
        }
        songs.emplace_back(name);
        using namespace std::chrono;
        auto time = system_clock::to_time_t(system_clock::now());
        std::cout << std::ctime(&time) << "READ FROM DB SONG: " << name << "
↪   START" << std::endl;
        input >> vecSize;
        for (Size i = 0; i < vecSize; ++i) {
            input >> hash >> blockNum;
            database[hash].emplace_back(std::make_pair(id, blockNum));
        }
        ++id;
        time = system_clock::to_time_t(system_clock::now());
        std::cout << std::ctime(&time) << "READ FROM DB SONG: " << name << "
↪   END" << std::endl;
//        input.ignore('\n', 1);
    }
    songs_ = songs;
    database_ = database;
    input.close();
}

void Shazam::write(std::ofstream &out) {
    std::unordered_map<Size, std::vector<std::pair<Size, Size>>>
↪   map(newSongs_.size()); // [ID, {[Hash, Block number]}]
    for (const auto &[hash, vector]: database_) {
        for (const auto &[id, blockNum]: vector) {
            if (newSongs_.contains(id)) {
```

```
                map[id].emplace_back(std::make_pair(hash, blockNum));
            }
        }
    }
    for (const auto &[id, vec]: map) {
        out << songs_[id] << std::endl;
        out << vec.size() << std::endl;
        for (const auto &[hash, blockNum]: vec) {
            out << hash << " " << blockNum << std::endl;
        }
    }
    out.close();
}

Shazam::~Shazam() {
    std::ofstream out(LIB_FILE, std::ostream::app | std::ostream::out);
    if (!out.is_open()) {
        std::cout << "ERROR: CAN NOT OPEN FILE: " << LIB_FILE << std::endl;
        perror(std::string("CAN NOT OPEN OUTPUT FILE:
↪   ").append(LIB_FILE).c_str());
    } else {
        write(out);
    }
}

std::vector<std::pair<std::string, double>> Shazam::search(const std::string
↪   &name) {
    using namespace std::chrono;
    auto time = system_clock::to_time_t(system_clock::now());
    std::cout << std::ctime(&time) << "\t\tBEGIN SEARCH\nGET HASHES: START "
↪   << name << std::endl;
    auto hashes = getHashes(name);
    time = system_clock::to_time_t(system_clock::now());
    std::cout << std::ctime(&time) << "GET HASHES: END\nSearch: START" <<
↪   std::endl;
    std::unordered_map<Size, Size> statistic;
    Size cnt = 0;
    /* Собираем статистику совпадений. Берем i и j отрезок времени в
↪   заданной песне.
     * Ищем совпадающие по хэшу песни с одинаковой разницей в отрезках.
     * В результате имеем пары [ID, stat], где ID - ID песни, stat -
↪   количество совпадающих отрезков */
```

```
    for (Size firstSample = 0; firstSample < hashes.size(); ++firstSample) {
        for (Size secondSample = firstSample + 1; secondSample <
↪   hashes.size(); ++secondSample) {
            Size firstHash = hashes[firstSample], secondHash =
↪   hashes[secondSample];
            auto firstIt = database_.find(firstHash);
            auto secondIt = database_.find(secondHash);
            if (firstIt == database_.end() or secondIt == database_.end()) {
                continue;
            }
            ++cnt;
            auto firstVec = firstIt->second, secondVec = secondIt->second;
            // вектор пар [id, sample], хэш каждого сэмпла равен либо хэшу
↪   первого сэмпла образца, либо второго
            for (const auto &[firstID, firstBlock]: firstVec) {
                for (const auto &[secondID, secondBlock]: secondVec) {
                    if (firstID == secondID and diff(firstSample,
↪   secondSample) == diff(firstBlock, secondBlock)) {
                        ++statistic[firstID];
                    }
                }
            }
        }
    }
    time = system_clock::to_time_t(system_clock::now());
    std::cout << std::ctime(&time) << "SEARCH: END\nCREATE RESULT: START" <<
↪   std::endl;
    std::vector<std::pair<std::string, double>> res;
    res.reserve(statistic.size());
    for (const auto &[id, stat]: statistic) {
        double percent = static_cast<double>(stat) /
↪   static_cast<double>(cnt) * 100.;
        res.emplace_back(songs_[id], percent);
    }
    std::sort(all(res), [](const auto &left, const auto &right) { return
↪   left.second > right.second; });
    time = system_clock::to_time_t(system_clock::now());
    std::cout << std::ctime(&time) << "CREATE RESULT: END\n EXIT FROM
↪   SEARCH" << std::endl;
    return res;
}
```

```
Size Shazam::diff(const Size &left, const Size &right) {
    return (left > right) ? left - right : right - left;
}
```

---

```
#include "Shazam.h"

inline void usage() {
    std::cout
            << "USAGE:\n append <path to file>: add file to database\n
↪   search <path to file>: search mp3 at database\n exit: end program\n";
}

int main(int argv, char **argc) {
    Shazam shazam;
    std::string query, filename;
    usage();
    bool run = true;
    while (run) {
        std::cin >> query;
        if (query == "append") {
            std::cin >> filename;
            shazam.append(filename);
            usage();
        } else if (query == "search") {
            std::cin >> filename;
            auto res = shazam.search(filename);
            std::cout << "\t\tSEARCH RESULT:" << std::endl;
            std::cout.precision(4);
            for (const auto &[name, percent]: res) {
                std::cout << percent << "% " << name << std::endl;
            }
            usage();
        } else if (query == "exit") {
            run = false;
        } else {
            usage();
        }
    }
    return 0;
}
```

## Дневник отладки

При реализации чтения/записи базы данных необходимо было убедиться в том, что порядок считывание (запись) происходит в правильном порядке. К сожалению, получилось это не с первого раза.

Во время реализация быстрого преобразования Фурье я наивно предположил, что корни при рекурсивных вызовах будут совпадать. Однако, на самом деле это не так. Поэтому при первой реализации во время поиска фрагментов процент совпадений был мал, однако после исправления ошибки он и стал похож на правду.

## Результаты работы программы

Для того, чтобы программа создала базу данных, необходимо поместить в папку $LIBRARY$ аудиозаписи. Я поместил туда 3 песни известной рок-группы *Rammstein* (также включил туда кавер одной из песен):

- *Deutschland*
- *Du Hast*
- *Feuer Frei*

Я решил взять песню *Deutschland* из другого источника, записать ее фрагмент с помощью микрофона на телефон, а затем попробовать найти ее с помощью *Shazam*.

Приведу фрагмент вывода, отвечающий за результат поиска.

```
search ./Search/DeutschlandPhone.mp3
    SEARCH RESULT:
5.514% Deutschland-Cover.mp3
3.06% Rammstein-Deutschland.mp3
0.1047% Rammstein-FeuerFrei.mp3
0.06734% Rammstein-DuHast.mp3
```

Примечательно, что записывал я оригинальную версию, но поиск выдал мне кавер. Однако, в оригинальной версии тоже хороший процент совпадений относительно других, поэтому он был бы предложен пользователю в качестве альтернативы.

Теперь попробуем взять разные фрагменты песен и поискать их в базе (более всего мне было интересно, насколько сильно будут отличаться проценты у кавер-версии и оригинала, поэтому для этой песни я взял аж 3 фрагмента).

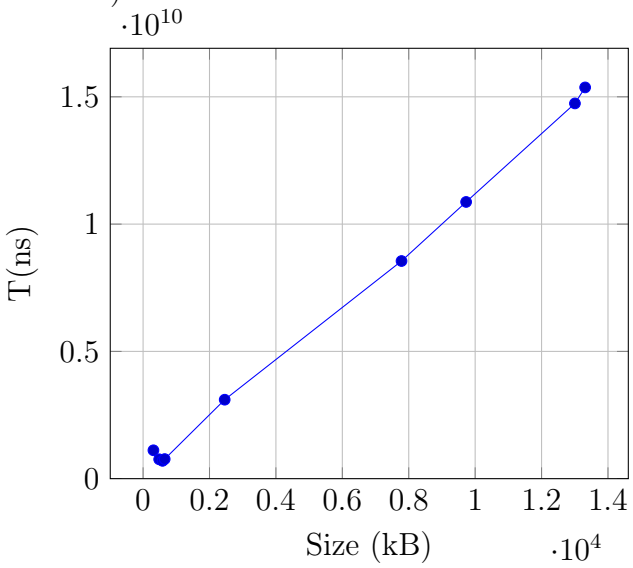
```
search ./Search/DeutschlandFirst.mp3
    SEARCH RESULT:
53.29% Rammstein-Deutschland.mp3
40.98% Deutschland-Cover.mp3
0.8573% Rammstein-FeuerFrei.mp3

search ./Search/DeutschlandSecond.mp3
    SEARCH RESULT:
```

```
    21.4%  Rammstein-Deutschland.mp3
    16.98%  Deutschland-Cover.mp3
    1.084%  Rammstein-FeuerFrei.mp3
    0.05004%  Rammstein-DuHast.mp3

    search ./Search/DeutschlandThird.mp3
        SEARCH RESULT:
    41.99%  Rammstein-Deutschland.mp3
    0.3883%  Deutschland-Cover.mp3
    0.07765%  Rammstein-DuHast.mp3
    0.03883%  Rammstein-FeuerFrei.mp3

    search ./Search/DuHast.mp3
        SEARCH RESULT:
    23.89%  Rammstein-DuHast.mp3
    0.6637%  Rammstein-FeuerFrei.mp3
    0.03161%  Rammstein-Deutschland.mp3
    0.03161%  Deutschland-Cover.mp3
```

Примечательно, что во время проигрышей оригинал и кавер-версия выдают приблизительно одинаковый процент совпадений (всё же оригинал выигрывает). Однако на третьем фрагменте есть слова, из-за которых кавер так сильно теряет проценты.

## Тест производительности

Протестируем добавление в базу упомянутых выше аудиозаписей (включая короткие фрагменты).



## Выводы

В ходе работы я подробно познакомился с дискретным преобразованием Фурье и алгоритмом быстрого преобразования Фурье (алгоритм Кули-Тьюки), позволяющий вычислять ДПФ за время $O(n \log n)$ вместо $O(n^2)$. В качестве прикладного применения преобразование Фурье широко применяется в обработке сигналов, так как оно позволяет перейти от временного представления к частотному.

В ходе работы был создан простейший сервис распознавания музыки. Конечно, он нуждается в сильной доработке: в частности, помимо $GUI$, требуется создать критерий, согласно которому определять, является ли песня подходящей или нет. В данном случае этим критерием является топ-1 в поиске, но если песни нет в базе данных или она сильно искажена (допустим, моим телефоном), то этот вариант не всегда может быть верным.