# Описание алгоритма

Основная идея алгоритма - асинхронная выгрузка данных о пользователях Вконтакте. На вход поступает

1. файл config.ini, в котором содержится информация для подключения к БД MySQL и список токенов
2. csv-файл с id пользователей.

На выходе алгоритм создает 2 csv-файла:

1. Информация о пользователе (пол, закрыта ли страница (1 - да, 0 - нет), год рождения пользователя (0 если не заполнено), текущий род деятельности пользователя (0 если нет), информация о карьере и об образовании).
2. Информация о группах пользователей(id, словарь с тематиками групп вида {activity: number})

Далее будет более подробно описана реализация алгоритма.

## Файл config.ini

Вид файла должен быть таким:

```
[database]
host=localhost
user=root
password=password
db=user_info
charset=utf8mb4
[tokens]
token1 = vk1.a.Sjbrtifp5ddYgtNq2DFcnse0WpGyQaE2bhmBkLfg0HPo117t-UT8eHUI
token2 = vk1.a.aPdOAhc4bjJVDm1cxRmH7iSBOvioe188wdaWWkuoda_4Iz7UpHd2SE2J
token3 = vk1.a.T6u75JPNyxtlcd0JsoKknV7hP3mRL3psJfoG_Y9ytoY_8Qa5l_jvOu12
token4 = vk1.a.c7Z_sgC4IvrRaUrn4wTXXm-EjiHKbTI8xgKVBOiWXO0DosRfITnmNIhE
token5 = vk1.a.MKrYjlkdx1RQK-SYskuog1ZHsv429Mo_B5F3AeKICn1z3VJ-NH_2623V
```

Секция database обрабатывается именно в таком порядке, его необходимо сохранить.

Далее в функции main() происходит подключение к БД, а также запускается выполнение всего алгоритма:

```python
async def main(filename, path):
    # Функция для подключения к БД и выполнения всего кода
    database, tokens = read_config(filename)
    connection = None
    try:
        connection = pymysql.connect(
            host=database[0],
            user=database[1],
            password=database[2],
            db=database[3],
            charset=database[4]
        )
        print("Connection to DB successful")
    except pymysql.Error as error:
        print(f'An error occurred: {error}')
    if connection is not None:
        ids_file = pd.read_csv(path, engine='python')
        task = asyncio.create_task(get_info(ids_file['id'].astype('str')[:500], tokens, connection))
        await task
```

## get_info()

Функция принимает на вход pandas.Series - столбец из csv файла с id пользователей; tokens - список токенов, полученный из файла конфига; connection - объект для подключения к БД.
Сначала получаем очередь с помощью get_apis(tokens) , элементами которой являются API ВК. Затем список id делится на len(tokens) частей, после этого выполняется функция get_users(). После уже происходит обработка корутин из coros_users и coros_groups.

Код выглядит следующим образом:

```python
async def get_info(ids: pd.Series, tokens: list, connection) -> None:
    # Основная функция получения информации
    apis = await get_apis(tokens)
    # Делим исходный список id пользователей на равные части
    divided_list = numpy.array_split(ids, len(tokens))
    list_of_users, list_of_groups = [], []
    coros_users = (get_users(ids, False, apis) for ids in divided_list)
    coros_groups = (get_users(ids, True, apis) for ids in divided_list)
    create_tables(connection)
    for coro in list(coros_users):
        users = await coro
        info = await user_info(users, connection)
        list_of_users.extend(info)
    user_df = pd.DataFrame(list_of_users, columns=['id', 'is_closed', 'sex', 'bdate_year', 'occupation', 'career',
                                                   'edu_information', 'edu_faculty', 'edu_graduation'])
    user_df.to_csv('data_users.csv', index=False)
    print('users parsed')
    for coro in list(coros_groups):
        groups = await coro
        info = await group_info(groups, list(ids), connection)
        list_of_groups.extend(info)
    data = numpy.array([list(ids), list_of_groups]).transpose()
    group_df = pd.DataFrame(data, columns=['id', 'activities'])
    group_df.to_csv('data_groups.csv', index=False)
    print('groups parsed')
```

## get_users()

Обработка зависит от параметра one_by_one.

Если в нем хранится значение True, то необходимо обработать запросы о группах пользователя. С помощью user_list_handler разбиваем исходный список с id пользователей на части из 25 id, после этого добавляем каждую часть в очередь. Число 25 обусловлено максимальным числом независимых запросов, обрабатываемых методом execute() (он потребуется далее). Затем каждая полученная часть кладется в очередь, каждый элемент которой будет обрабатываться с помощью response_executor()

Если же в one_by_one значение False, то необходимо обработать запросы об информации пользователей. В таком случае список с id делится на (len(user_ids) // 3) частей. Это удобно тем, что количество id в одной части зависит от длины всего списка id, а также тем, что в итоге получится всего 3(или 4) части(в случае, если len(user_ids) кратна 3, части будет 3, иначе - 4, где в четвертой части лежит остаток id). Затем в очередь кладется кортеж вида (тип запроса, {словарь с полями}).

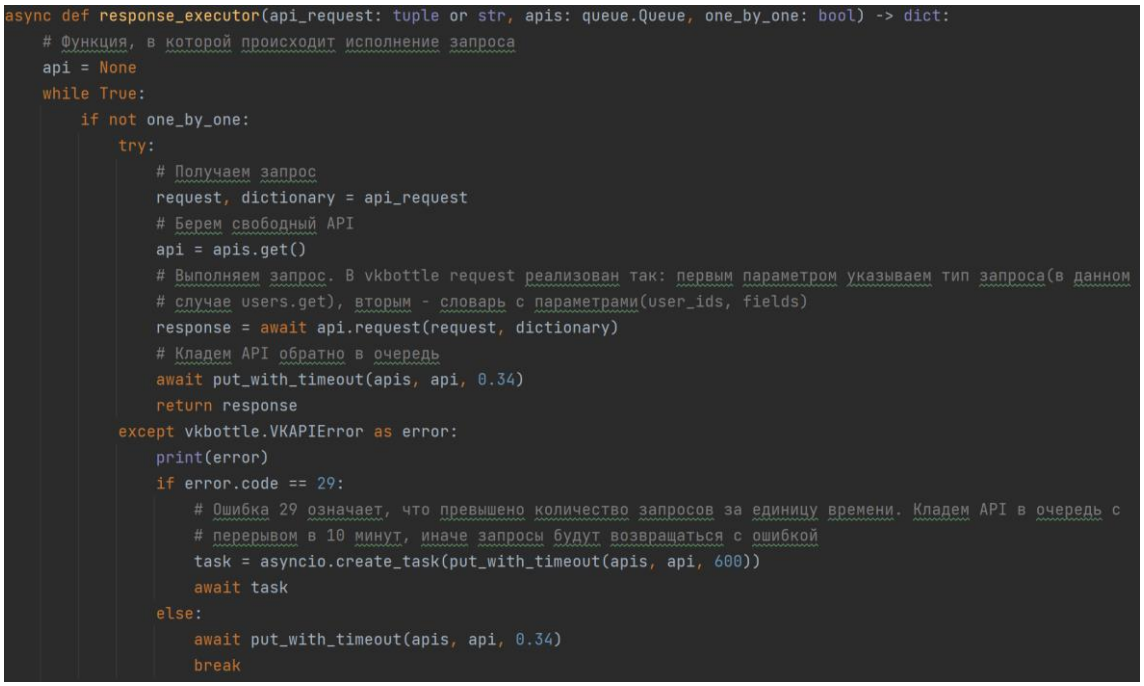
Код представлен ниже:

```python
async def get_users(user_ids: list, one_by_one: bool, apis) -> list:
    list_of_users = []
    api_requests = queue.Queue()
    # Параметр one_by_one означает, что мы хотим за 1 запрос обрабатывать 1 id. Такое может быть в случае с парсиногом
    # тематик групп
    if one_by_one:
        # Но на самом деле, вместо простого groups.get для каждого id используется метод execute, объединяющий в себе
        # несколько независимых запросов. Число 25 обусловлено тем, что метод поддерживает до 25 независимых запросов.
        user_ids_list = user_list_handler(user_ids, 25)
        for li in user_ids_list:
            api_requests.put(li)
    else:
        user_ids_list = user_list_handler(user_ids, len(user_ids) // 3)
        for user_id in user_ids_list:
            # Добавляем в очередь запрос
            api_requests.put(('users.get',
                              {'user_id': user_id, 'fields': 'sex, bdate, occupation, education, career'}))
    while not api_requests.empty():
        api_request = api_requests.get()
        # Получаем результат запроса в виде словаря с ключом response и значением - словарем с интересующими данными
        response = await response_executor(api_request, apis, one_by_one)
        list_of_users.extend(response['response'])
    return list_of_users
```

## response_executor()

Здесь также обработка зависит от параметра one_by_one

В случае, если в нем True, рассматривается запрос users.get. Сначала получаем запрос из переданной очереди, получаем свободный API, потом отправляем запрос. При успешном выполнении вернется словарь вида {response: {count: int, items: dict}}, а API кладется в очередь с паузой в 0.34 секунды. В случае ошибки обрабатывается только ошибка с кодом 29, которая говорит о том, что достигнут количественный лимит на вызов метода. В этом случае API помещается обратно в очередь с паузой в 10 минут.



```python
async def response_executor(api_request: tuple or str, apis: queue.Queue, one_by_one: bool) -> dict:
    # Функция, в которой происходит исполнение запроса
    api = None
    while True:
        if not one_by_one:
            try:
                # Получаем запрос
                request, dictionary = api_request
                # Берем свободный API
                api = apis.get()
                # Выполняем запрос. В vkbottle request реализован так: первым параметром указываем тип запроса(в данном
                # случае users.get), вторым - словарь с параметрами(user_ids, fields)
                response = await api.request(request, dictionary)
                # Кладем API обратно в очередь
                await put_with_timeout(apis, api, 0.34)
                return response
            except vkbottle.VKAPIError as error:
                print(error)
                if error.code == 29:
                    # Ошибка 29 означает, что превышено количество запросов за единицу времени. Кладем API в очередь с
                    # перерывом в 10 минут, иначе запросы будут возвращаться с ошибкой
                    task = asyncio.create_task(put_with_timeout(apis, api, 600))
                    await task
                else:
                    await put_with_timeout(apis, api, 0.34)
                    break
```

Если же one_by_one == False, рассматриваются запросы groups.get. При той же реализации, что и в прошлом блок, обработка займет много времени. Также высока вероятность, что для API будет достигнут лимит вызова. Поэтому необходим метод execute. На вход ему подается значение code - функция на языке VKScript (это JS, только с некоторыми ограничениями), с помощью которой происходит выполнение независимых запросов. Благодаря execute вместо 25 отдельных запросов посылается 1, что улучшает и скорость, и стойкость к ошибкам.

```python
    else:
        api = apis.get()
        # Метод execute на вход принимает код на языке VKScript(что-то вроде JavaScript). В данном случае если
        # запрос groups.get выполняется правильно, то помещаем его в результирующий список, иначе добавляем пустой
        # список
        response = await api.execute(
            f"var u = [{str(api_request)}];"
            "var i = 0;"
            "var res = [];"
            "while (i < u.length){"
            "var api = API.groups.get({'user_id': u[i], 'fields':'activity', 'extended': 1});"
            "if (api){"
            "res.push(api);"
            "}"
            "else{"
            "res.push([]);"
            "}"
            "i = i + 1;"
            "};"
            "return res;"
        )
        await put_with_timeout(apis, api, 0.34)
        return response
```