

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Факультет безопасности информационных технологий

Дисциплина:

«Алгоритмы и структуры данных»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

«PSRS-сортировка на кольцевой очереди. Оценка сложности»

Выполнили:

Суханкулиев М.,
студент группы N3246

(подпись)

Проверил:

Ерофеев С. А.

(отметка о выполнении)

(подпись)

Санкт-Петербург

2025 г.

СОДЕРЖАНИЕ

Введение	3
1 Алгоритм PSRS	4
1.1 Описание алгоритма	4
1.2 Блок-схема алгоритма	5
2 Спецификация переменных	8
3 Реализация программы	9
3.1 Описание программы	9
3.2 Код программы	9
3.2.1 psrs.h	9
3.2.2 psrs.c	9
3.2.3 main.c	12
4 Тестирование программы	14
4.1 Скриншоты тестирования	14
5 Оценка сложности	15
Заключение	17
Список использованных источников	18

ВВЕДЕНИЕ

Цель работы – Разработать программу PSRS-сортировки чисел из файла, которые записываются в кольцевую очередь на базе массива. Оценить сложность.

Для достижения поставленной цели необходимо решить следующие **задачи**:

- Разработать блок-схему алгоритма;
- Составить спецификацию всех переменных;
- Реализовать программу на языке C;
- Провести тестирование программы;
- Оценить сложность алгоритма.

1 АЛГОРИТМ PSRS

1.1 Описание алгоритма

Начало.

Шаг 1. Исходный массив `numbers` из n элементов разделим поровну между p потоками. Каждый поток получит подмассив размера $\text{base_size} = n / p$ (с учетом возможного остатка $\text{remainder} = n \% p$).

Шаг 2. На каждом потоке запускаем быструю сортировку (`qsort`) для соответствующего подмассива (`thread_data[i].array`).

Шаг 3. Формируем вспомогательный массив `sample` из p отсортированных подмассивов `sub_arrays[i]`, выбирая из каждого p элементов под индексами: $j \cdot \left(\frac{n}{p^2}\right)$, где $j = 0, 1, 2, \dots, p - 1$.

Шаг 4. Общий размер массива `samples` будет $p * p = p^2$. Сортируем вспомогательный массив `samples` с помощью быстрой сортировки (`qsort`).

Шаг 5. Формируем массив разделителей (`splitters`) из элементов `samples`, взятых под индексами $k \cdot p + \left\lfloor \frac{p}{2} \right\rfloor - 1$, где $k = 1, 2, \dots, p - 1$. Таким образом, `splitters` содержит $p-1$ разделитель: a_1, a_2, \dots, a_{p-1} .

Шаг 6. На каждом потоке разбиваем локально отсортированные подмассивы `sub_arrays[i]` на p групп по значениям разделителей `splitters`, то есть получаем группы: $(-\infty, a_1], (a_1, a_2], \dots, (a_{p-1}, +\infty)$. Размеры групп фиксируются в массиве `counts[i][j]`, где i – номер потока, j – номер группы. Каждому потоку передаются соответствующие группы от всех остальных потоков (т. е. поток i получает i -е группы от всех p потоков).

Шаг 7. Каждый поток получает по p отсортированных фрагментов, которые объединяются в один подмассив с помощью многопутевого слияния.

Шаг 8. Полученные p отсортированных подмассивов объединяются в итоговый массив `result`, с помощью последовательной записи.

Шаг 9. Записываем отсортированные данные из `result` в выходной файл (`write_result_to_file`).

Конец.

1.2 Блок-схема алгоритма

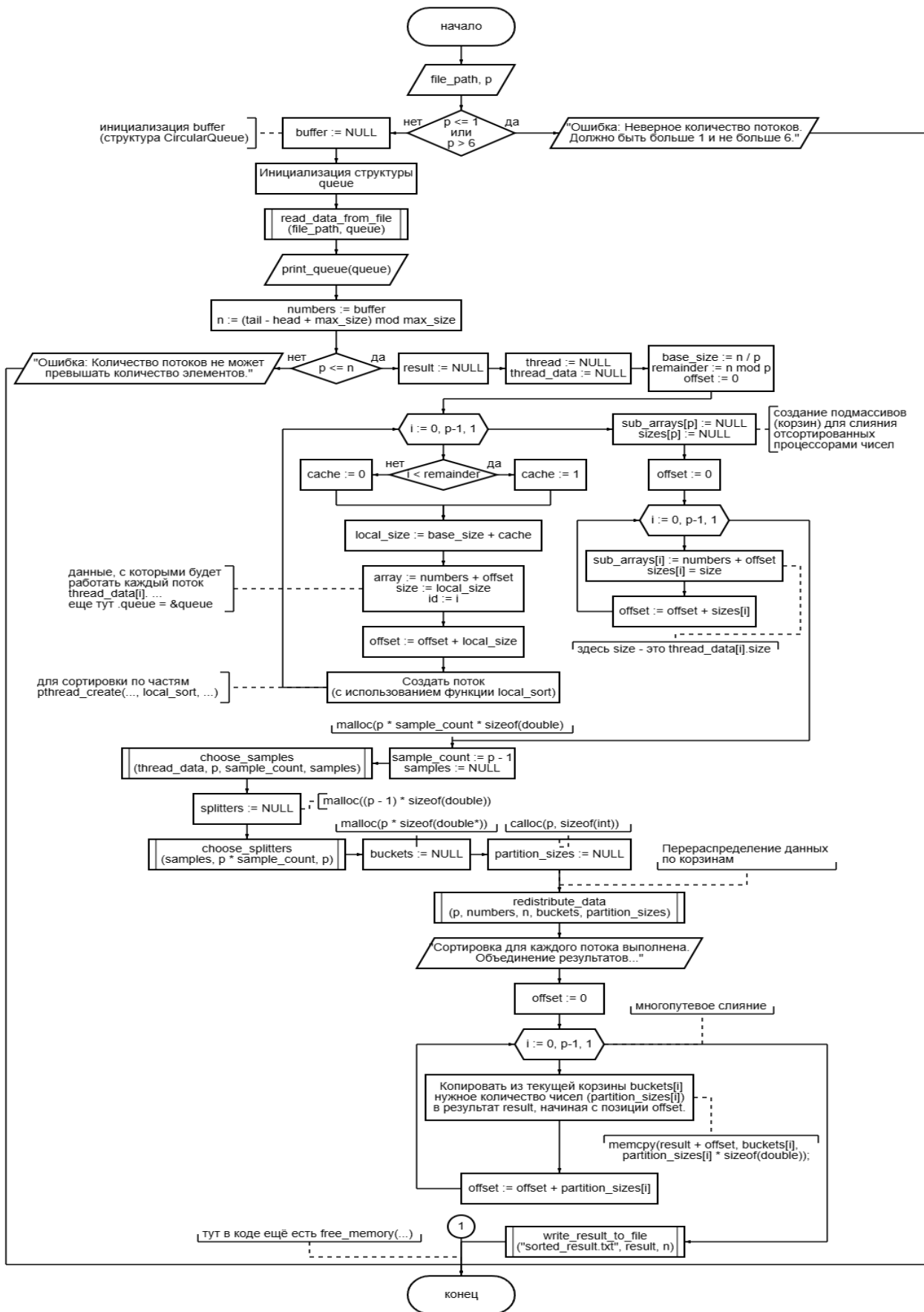


Рисунок 1 – Основная схема алгоритма

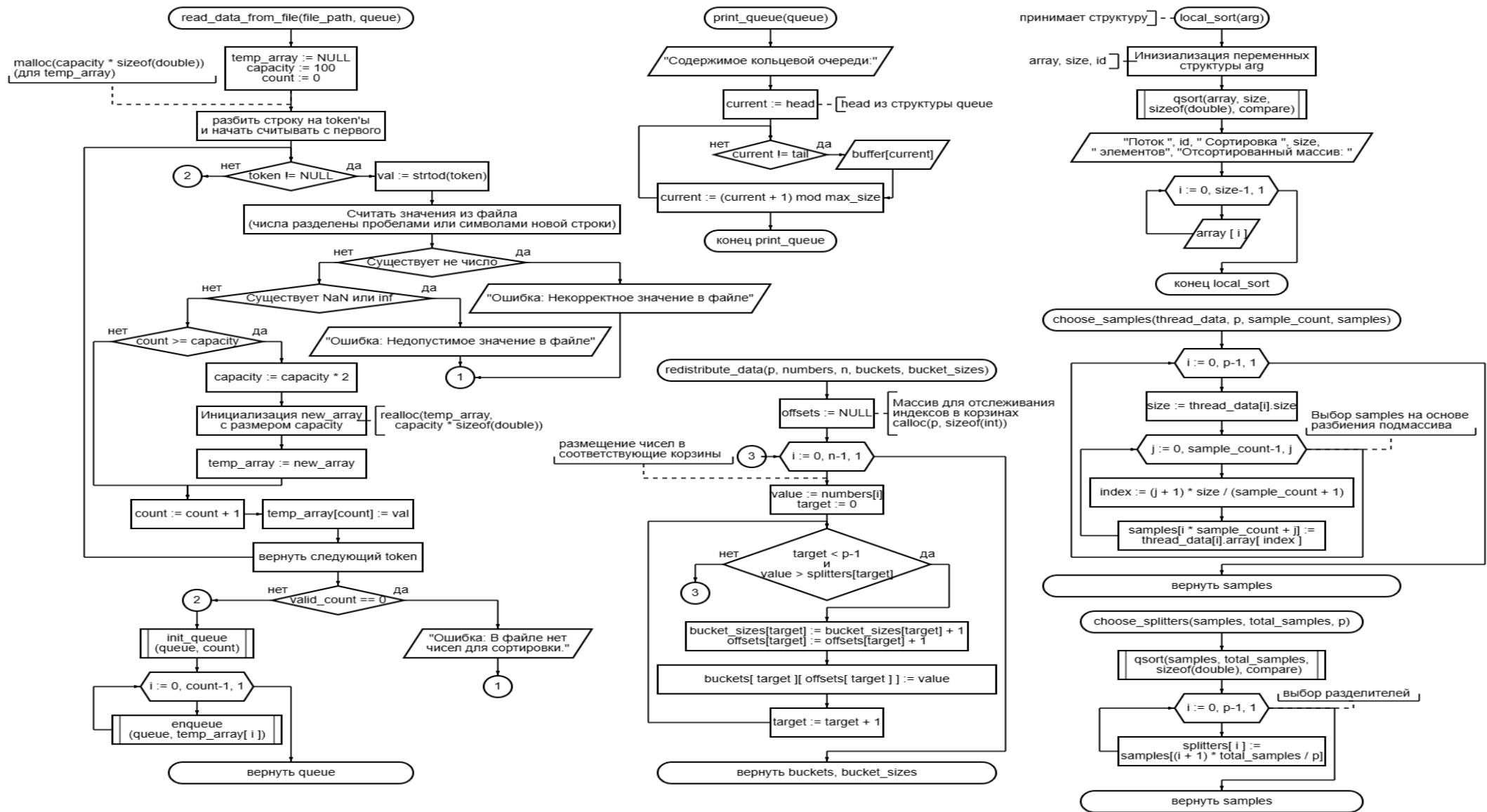


Рисунок 2 – Подпрограммы (I часть)

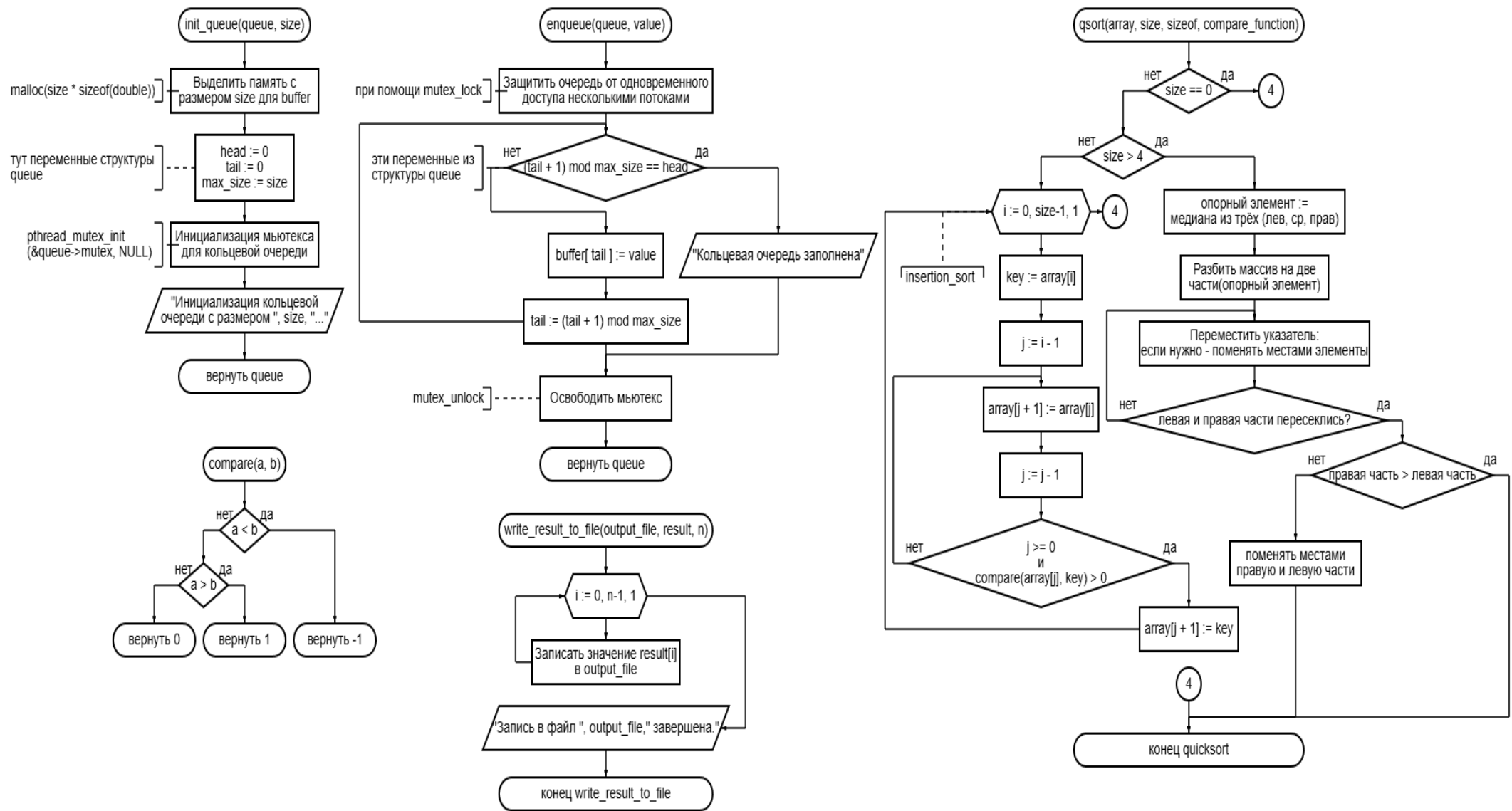


Рисунок 3 – Подпрограммы (II часть)

2 СПЕЦИФИКАЦИЯ ПЕРЕМЕННЫХ

Переменная	Описание	Тип использования	Тип	Размер (байт)	Диапазон значений
file_path	Путь к файлу, содержащему данные для сортировки	Входная	const char	8	Путь к файлу в файловой системе (строка)
p	Количество потоков	Входная	char	8	[2, 6]
queue	Кольцевая очередь для хранения чисел	Промежуточная	CircularQueue	8	Указатель на структуру кольцевой очереди
numbers	Массив чисел для сортировки	Входная/Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
n	Количество чисел в массиве	Промежуточная	int	4	[0, 2 147 483 647]
threads	Массив потоков	Промежуточная	pthread_t	8	Системный указатель на поток (зависит от платформы)
thread_data	Массив структур с данными для каждого потока	Промежуточная	ThreadData	8	Указатель на структуру с информацией о потоках
samples	Массив выборок для вычисления разделителей	Промежуточная	double	8	[2, 30]
splitters	Массив разделителей для многопутевого слияния	Промежуточная	double	8	[1, 5]
buckets	Массив указателей на корзины для перераспределенных данных	Промежуточная	double	8	Указатели на массивы с перераспределенными данными
partition_sizes	Массив размеров корзин	Промежуточная	int	4	[0, 2 147 483 647]
offset	Смещение для подсчета и копирования элементов	Промежуточная	int	4	[0, 2 147 483 647]
base_size	Базовый размер подмассива для одного потока	Промежуточная	int	4	[0, 1 073 741 824]
line	Буфер для чтения строки из файла	Промежуточная	char[256]	256	Строка с ASCII-символами, длина до 255 символов + \0
token	Буфер для токенов строки при разборе данных из файла	Промежуточная	char	8	Указатель на текущий токен
file	Указатель на файл	Промежуточная	FILE	8	Указатель на открытый файл
remainder	Остаток элементов при делении на количество потоков	Промежуточная	int	4	[0, 5]
temp_array	Временный массив для считывания данных до заполнения очереди	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
capacity	Текущая вместимость temp_array	Промежуточная	int	4	[0, 2 147 483 647]
count	Счётчик прочитанных чисел из файла	Промежуточная	int	4	[0, 2 147 483 647]
result	Массив для хранения отсортированных данных	Выходная	double	8	[-1.7977e + 308, 1.7977e + 308]

Примечание: размер переменных в памяти указан для стандартных платформ x86-64.

3 РЕАЛИЗАЦИЯ ПРОГРАММЫ

3.1 Описание программы

Программа написана на языке C и состоит из трех основных файлов:

- 1) Заголовочный файл (`psrs.h`) – содержит объявления функций и структур данных, которые используются в реализации программы.
- 2) Основной файл (`main.c`) – включает функцию `main()`, которая управляет процессом выполнения программы.
- 3) Файл с реализациями функций (`psrs.c`) – содержит определения всех функций, объявленных в заголовочном файле `psrs.h`.

3.2 Код программы

3.2.1 `psrs.h`

```
#ifndef PSRS_H
#define PSRS_H
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
typedef struct {
    double *buffer;           // Массив для хранения элементов кольцевой очереди
    int head;                 // Индекс головы очереди (где следующий элемент будет извлечен)
    int tail;                 // Индекс хвоста очереди (куда следующий элемент будет добавлен)
    int max_size;             // Максимальный размер очереди
    pthread_mutex_t mutex;    // Мьютекс для синхронизации доступа к очереди
} CircularQueue;
typedef struct {
    double *array;            // Указатель на массив, который будет сортироваться
    int size;                 // Количество элементов в этом массиве
    int id;                   // Уникальный идентификатор потока
    CircularQueue *queue;     // Указатель на кольцевую очередь
} ThreadData;
// Глобальные переменные (определены в psrs.c)
extern double *splitters;
extern int *partition_sizes;
// Функции работы с очередью
void init_queue(CircularQueue *queue, int size);
void enqueue(CircularQueue *queue, double value);
void print_queue(CircularQueue *queue);
// Функции сортировки и PSRS
int compare(const void *a, const void *b);
void *local_sort(void *arg);
void *final_sort(void *arg);
void choose_samples(ThreadData *thread_data, int p, int sample_count, double *samples);
void choose_splitters(double *samples, int total_samples, int p);
void redistribute_data(int p, double *numbers, int n, double **buckets, int *bucket_sizes);
// Работа с потоками
int initialize_threads(int p, pthread_t **threads, ThreadData **thread_data);
// Работа с файлами
int read_data_from_file(const char *file_path, CircularQueue *queue);
void write_result_to_file(const char *output_file, double *result, int n);
// Очистка памяти
void free_memory(double *numbers, double *result, pthread_t *threads, ThreadData *thread_data);
#endif // PSRS_H
```

3.2.2 `psrs.c`

```
#include "psrs.h"
double *splitters;
int *partition_sizes;
// Инициализация кольцевой очереди
void init_queue(CircularQueue *queue, int size) {
    queue->buffer = (double*)malloc(size * sizeof(double));
    if (!queue->buffer) {
        printf("Ошибка: Не удалось выделить память для кольцевой очереди\n");
        exit(1);
    }
    queue->head = 0;
    queue->tail = 0;
    queue->max_size = size;
```

```

    pthread_mutex_init(&queue->mutex, NULL);
    printf("Инициализация кольцевой очереди с размером %d...\n", size);
}
// Добавление элемента в кольцевую очередь
void enqueue(CircularQueue *queue, double value) {
    pthread_mutex_lock(&queue->mutex);
    if ((queue->tail + 1) % queue->max_size != queue->head) { // Проверка на заполненность
        queue->buffer[queue->tail] = value;
        queue->tail = (queue->tail + 1) % queue->max_size;
    } else {
        printf("Кольцевая очередь заполнена.\n");
    }
    pthread_mutex_unlock(&queue->mutex); // Освобождение мьютекса после работы с очередью
}
// Функция сортировки
int compare(const void *a, const void *b) {
    if (*(double*)a < *(double*)b) return -1;
    if (*(double*)a > *(double*)b) return 1;
    return 0;
}
// Локальная сортировка для каждого потока
void *local_sort(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    qsort(data->array, data->size, sizeof(double), compare); // Сортировка каждого подмассива
    printf("Поток %d: Сортировка %d элементов...\nОтсортированный подмассив: \n", data->id, data->size);
    for (int i = 0; i < data->size; i++) {
        printf("%.15g ", data->array[i]);
    }
    printf("\n");
    return NULL;
}
// Чтение данных из файла и заполнение кольцевой очереди
int read_data_from_file(const char *file_path, CircularQueue *queue) {
    FILE *file = fopen(file_path, "r");
    if (!file) {
        printf("Ошибка: Не удалось открыть файл %s\n", file_path);
        return 0;
    }
    double *temp_array = NULL;
    int capacity = 100;
    int count = 0;
    temp_array = malloc(capacity * sizeof(double));
    if (!temp_array) {
        printf("Ошибка: не удалось выделить память\n");
        fclose(file);
        return 0;
    }
    char line[256];
    while (fgets(line, sizeof(line), file)) {
        // Пропускаем строки, содержащие только пробелы или символы новой строки
        int only_whitespace = 1;
        for (char *p = line; *p != '\0'; ++p) {
            if (!isspace((unsigned char)*p)) {
                only_whitespace = 0;
                break;
            }
        }
        if (only_whitespace) {
            continue;
        }
        char *token = strtok(line, " \n\r\t");
        while (token) {
            char *endptr;
            double val = strtod(token, &endptr);
            // Проверка на корректность числа
            if (*endptr != '\0') {
                printf("Ошибка: Некорректное значение \"%s\" в файле\n", token);
                free(temp_array);
                fclose(file);
                return 0;
            }

```

```

        if (isnan(val) || isinf(val)) {
            printf("Ошибка: Значение \"%s\" в файле недопустимо\n", token);
            free(temp_array);
            fclose(file);
            return 0;
        }
        // Расширение массива при необходимости
        if (count >= capacity) {
            capacity *= 2;
            double *new_array = realloc(temp_array, capacity * sizeof(double));
            if (!new_array) {
                printf("Ошибка: не удалось перераспределить память\n");
                free(temp_array);
                fclose(file);
                return 0;
            }
            temp_array = new_array;
        }
        temp_array[count++] = val;
        token = strtok(NULL, " \\n\\r\\t");
    }
}
if (count == 0) {
    printf("Ошибка: В файле нет чисел для сортировки.\n");
    free(temp_array);
    fclose(file);
    return 0;
}
fclose(file);
init_queue(queue, count);
for (int i = 0; i < count; i++) {
    enqueue(queue, temp_array[i]);
}
free(temp_array);
return 1;
}
// Функция для выбора выборок для разделителей
void choose_samples(ThreadData *thread_data, int p, int sample_count, double *samples) {
    for (int i = 0; i < p; ++i) {
        int size = thread_data[i].size;
        // Процесс выбора samples[i * sample_count + j] на основе разбиения подмассива
        for (int j = 0; j < sample_count; ++j) {
            int index = (j + 1) * size / (sample_count + 1); // Выбор индекса для выборки
            samples[i * sample_count + j] = thread_data[i].array[index];
        }
    }
}
// Функция выбора разделителей (splitters)
void chooseSplitters(double *samples, int total_samples, int p) {
    qsort(samples, total_samples, sizeof(double), compare); // Сортировка выборок
    for (int i = 0; i < p - 1; ++i) {
        splitters[i] = samples[(i + 1) * total_samples / p]; // Выбор разделителей
    }
}
// Функция перераспределения данных по корзинам на основе разделителей
void redistribute_data(int p, double *numbers, int n, double **buckets, int *bucket_sizes) {
    int *offsets = (int*)calloc(p, sizeof(int)); // Массив для отслеживания индексов в корзинах
    memset(bucket_sizes, 0, p * sizeof(int)); // Инициализация всех размеров корзин в 0
    // Преобразование чисел в корзины на основе разделителей
    for (int i = 0; i < n; ++i) {
        double value = numbers[i];
        int target = 0;
        // Определение корзины, в которую попадет элемент
        while (target < p - 1 && value > splitters[target]) target++;
        bucket_sizes[target]++; // Увеличение размера корзины
    }
    // Выделение памяти для каждой корзины
    for (int i = 0; i < p; ++i)
        buckets[i] = (double*)malloc(bucket_sizes[i] * sizeof(double));
    // Размещение чисел в соответствующих корзинах
    for (int i = 0; i < n; ++i) {
        double value = numbers[i];

```

```

        int target = 0;
        // Поиск корзины для текущего элемента
        while (target < p - 1 && value > splitters[target]) target++;
        buckets[target][offsets[target]++] = value; // Размещение в корзине
    }
    free(offsets); // Освобождение памяти для индексов
}
// Финальная сортировка для каждого потока
void *final_sort(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    qsort(data->array, data->size, sizeof(double), compare);
    return NULL;
}
// Инициализация потоков и данных
int initialize_threads(int p, pthread_t **threads, ThreadData **thread_data) {
    *threads = malloc(p * sizeof(pthread_t)); // Выделение памяти для потоков
    *thread_data = malloc(p * sizeof(ThreadData)); // Выделение памяти для данных потоков
    if (!(*threads) || !(*thread_data)) {
        printf("Ошибка: Не удалось выделить память для потоков\n");
        free(*threads);
        free(*thread_data);
        return 0;
    }
    return 1;
}
// Запись отсортированных данных в файл
void write_result_to_file(const char *output_file, double *result, int n) {
    FILE *file = fopen(output_file, "w");
    if (!file) {
        printf("Ошибка: Не удалось открыть выходной файл %s\n", output_file);
        return;
    }
    for (int i = 0; i < n; i++)
        fprintf(file, "%.15g ", result[i]); // Запись каждого числа в файл
    fclose(file);
    printf("Запись в файл \"%s\" завершена.\n", output_file);
}
// Функция вывода содержимого кольцевой очереди
void print_queue(CircularQueue *queue) {
    printf("Содержимое кольцевой очереди: \n");
    int current = queue->head;
    // Перебор элементов в кольцевой очереди и вывод их на экран
    while (current != queue->tail) {
        printf("%.15g ", queue->buffer[current]);
        current = (current + 1) % queue->max_size;
    }
    printf("\n");
}
// Освобождение выделенной памяти
void free_memory(double *numbers, double *result, pthread_t *threads, ThreadData *thread_data) {
    free(numbers);
    free(result);
    free(threads);
    free(thread_data);
}

```

3.2.3 main.c

```

#include "psrs.h"
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Использование: %s <путь к файлу> <количество потоков>\n", argv[0]);
        return 1;
    }
    const char *file_path = argv[1];
    int p = atoi(argv[2]);
    if (p <= 1 || p > 6) {
        printf("Ошибка: Неверное количество потоков. Должно быть больше 1 и не больше 6.\n");
        return 1;
    }
    CircularQueue queue;
    if (!read_data_from_file(file_path, &queue)) return 1;
    // Выводим содержимое кольцевой очереди после её заполнения
    print_queue(&queue);
}

```

```

double *numbers = queue.buffer;
int n = (queue.tail - queue.head + queue.max_size) % queue.max_size;
if (p > n) {
    printf("Ошибка: Количество потоков не может превышать количество элементов.\n");
    free(queue.buffer);
    return 1;
}
double *result = malloc(n * sizeof(double));
if (!result) {
    printf("Ошибка: Не удалось выделить память для результата\n");
    free(queue.buffer);
    return 1;
}
pthread_t *threads;
ThreadData *thread_data;
if (!initialize_threads(p, &threads, &thread_data)) {
    free(queue.buffer);
    free(result);
    return 1;
}
int base_size = n / p, remainder = n % p, offset = 0;
// Разделение массива на части для каждого потока
for (int i = 0; i < p; ++i) {
    int local_size = base_size + (i < remainder);
    thread_data[i] = (ThreadData){ .array = numbers + offset, .size = local_size, .id = i, .queue
    = &queue };
    offset += local_size;
    pthread_create(&threads[i], NULL, local_sort, &thread_data[i]); // Создание потоков для
    локальной сортировки
}
for (int i = 0; i < p; ++i) pthread_join(threads[i], NULL);
// Выборка для разделителей
int sample_count = p - 1;
double *samples = malloc(p * sample_count * sizeof(double));
choose_samples(thread_data, p, sample_count, samples);
// Выбор разделителей
splitters = malloc((p - 1) * sizeof(double));
chooseSplitters(samples, p * sample_count, p);
free(samples);
// Перераспределение данных по корзинам
double **buckets = malloc(p * sizeof(double*));
partition_sizes = calloc(p, sizeof(int));
redistribute_data(p, numbers, n, buckets, partition_sizes);
// Финальная сортировка для каждого потока
for (int i = 0; i < p; ++i) {
    thread_data[i] = (ThreadData){ .array = buckets[i], .size = partition_sizes[i], .id = i };
    pthread_create(&threads[i], NULL, final_sort, &thread_data[i]);
}
for (int i = 0; i < p; ++i) pthread_join(threads[i], NULL);
printf("Сортировка для каждого потока выполнена. Объединение результатов...\n");
// Объединение результатов из всех корзин
offset = 0;
for (int i = 0; i < p; ++i) {
    memcpy(result + offset, buckets[i], partition_sizes[i] * sizeof(double));
    offset += partition_sizes[i];
    free(buckets[i]);
}
// Запись результата в файл
write_result_to_file("sorted_result.txt", result, n);
free_memory(numbers, result, threads, thread_data);
free(splitters);
free(partition_sizes);
free(buckets);
return 0;
}

```

4 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Программа скомпилирована с использованием следующих флагов:

```
gcc -O3 main.c psrs.c -o psrs -lpthread -lm -Wall -Wextra -Werror
```

В процессе тестирования не было обнаружено утечек памяти или других проблем (примеры ниже).

4.1 Скриншоты тестирования



```
(kali㉿kali)-[~/Desktop/algos2]
$ ./psrs file.txt 6
Ошибка: Некорректное значение "asd" в файле

(kali㉿kali)-[~/Desktop/algos2]
$ ./psrs file.txt
Использование: ./psrs <путь к файлу> <количество потоков>

(kali㉿kali)-[~/Desktop/algos2]
$ ./psrs file.txt -5
Ошибка: Неверное количество потоков. Должно быть больше 1 и не больше 6.

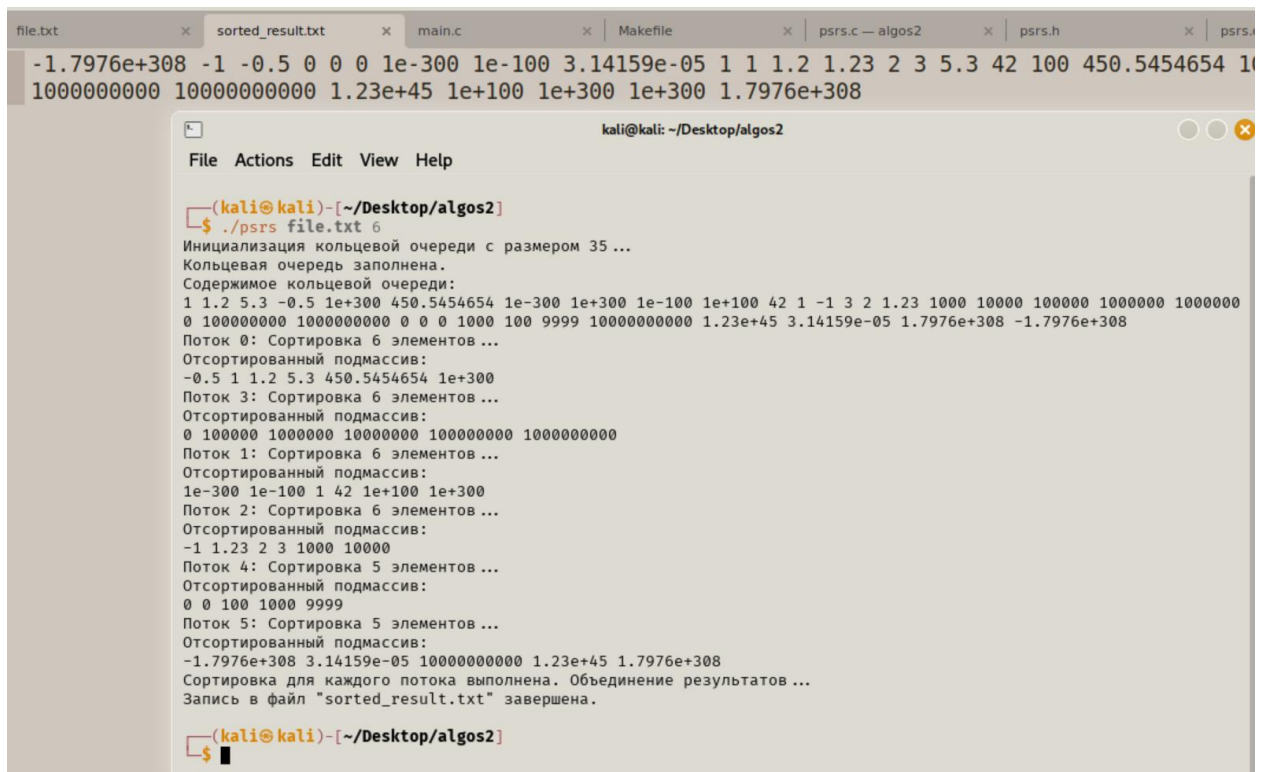
(kali㉿kali)-[~/Desktop/algos2]
$ ./psrs file1.txt 5
Ошибка: Не удалось открыть файл file1.txt

(kali㉿kali)-[~/Desktop/algos2]
$ ./psrs file1.txt 6
Ошибка: Не удалось открыть файл file1.txt

(kali㉿kali)-[~/Desktop/algos2]
$ ./psrs file.txt 6
Ошибка: Значение "inf" в файле недопустимо

(kali㉿kali)-[~/Desktop/algos2]
$
```

Рисунок 4 – Обработка ошибок



```
file.txt x sorted_result.txt x main.c x Makefile x psrs.c — algos2 x psrs.h x psrs.
-1.7976e+308 -1 -0.5 0 0 0 1e-300 1e-100 3.14159e-05 1 1 1.2 1.23 2 3 5.3 42 100 450.5454654 1
1000000000 1000000000 1.23e+45 1e+100 1e+300 1e+300 1.7976e+308

kali㉿kali: ~/Desktop/algos2
File Actions Edit View Help

(kali㉿kali)-[~/Desktop/algos2]
$ ./psrs file.txt 6
Инициализация кольцевой очереди с размером 35 ...
Кольцевая очередь заполнена.
Содержимое кольцевой очереди:
1 1.2 5.3 -0.5 1e+300 450.5454654 1e-300 1e+300 1e-100 1e+100 42 1 -1 3 2 1.23 1000 10000 100000 1000000 1000000
0 100000000 1000000000 0 0 0 1000 100 9999 10000000000 1.23e+45 3.14159e-05 1.7976e+308 -1.7976e+308
Поток 0: Сортировка 6 элементов ...
Отсортированный подмассив:
-0.5 1 1.2 5.3 450.5454654 1e+300
Поток 3: Сортировка 6 элементов ...
Отсортированный подмассив:
0 100000 1000000 10000000 100000000 1000000000
Поток 1: Сортировка 6 элементов ...
Отсортированный подмассив:
1e-300 1e-100 1 42 1e+100 1e+300
Поток 2: Сортировка 6 элементов ...
Отсортированный подмассив:
-1 1.23 2 3 1000 10000
Поток 4: Сортировка 5 элементов ...
Отсортированный подмассив:
0 0 100 1000 9999
Поток 5: Сортировка 5 элементов ...
Отсортированный подмассив:
-1.7976e+308 3.14159e-05 10000000000 1.23e+45 1.7976e+308
Сортировка для каждого потока выполнена. Объединение результатов ...
Запись в файл "sorted_result.txt" завершена.

(kali㉿kali)-[~/Desktop/algos2]
$
```

Рисунок 5 – Работа программы со всеми допустимыми значениями

5 ОЦЕНКА СЛОЖНОСТИ

- **Начальная сортировка:** Для каждого из p подмассивов выполняется быстрая сортировка (лемма: Время работы алгоритма быстрой сортировки равно $O(n \log n)$). На каждом шаге быстрой сортировки происходит сравнение двух элементов, что можно считать одной элементарной операцией), что дает сложность

$$O\left(\frac{n}{p} \log \frac{n}{p}\right)$$

для каждого подмассива. То есть на одном процессоре или потоке мы сортируем подмассив из $\frac{n}{p}$ элементов.

- **Выбор порядковых элементов:** Для каждого процессора выбирается p элементов для дальнейшего слияния. Эта выборка выполняется за время

$$O(p),$$

так как мы просто выбираем элемент из отсортированных подмассивов.

- **Слияние выбранных элементов в каждом процессе:** В процессе выбора разделителей требуется сортировка p элементов, и эта операция будет требовать $O(p \log p)$ времени. Но для корректного слияния данных, необходимо выполнить несколько операций слияния. Так как на каждом шаге слияния участвуют p разделителей, итоговая сложность слияния с учётом обмена данных и сортировки разделителей будет:

$$O(p^2 \log p)$$

- **Слияние подмассивов:** После того как каждый процессор обработает данные и выберет нужные элементы, необходимо произвести слияние этих подмассивов. Количество элементов для слияния в каждом подмассиве $\frac{n}{p^2}$, так как каждый процессор работает с подмассивом длины $\frac{n}{p}$, и данные для слияния собираются по этому принципу.

Сложность слияния всех подмассивов (при равномерном распределении данных) можно записать как:

$$O\left(\sum_{k=2}^p \frac{kn}{p^2}\right) = O\left(\frac{n}{p^2} \cdot \sum_{k=2}^p k\right)$$

Сумма $\sum_{k=2}^p k$ – это просто сумма натуральных чисел от 2 до p , которая может быть выражена через формулу для суммы арифметической прогрессии:

$$\sum_{k=2}^p k = \frac{p(p+1)}{2} - 1$$

Однако для больших p эта сумма будет асимптотически равна $O(p^2)$, так как $\frac{p(p+1)}{2} \sim \frac{p^2}{2}$.

$$O\left(\frac{n}{p^2} \cdot \sum_{k=2}^p k\right) = O(n)$$

Это означает, что для слияния всех подмассивов потребуется линейное время $O(n)$, так как процесс слияния сводится к линейному объединению всех данных.

- **Суммарная сложность:** Теперь можно подытожить все этапы алгоритма:

$$O\left(\frac{n}{p} \log \frac{n}{p}\right) + O(p^2 \log p) + O(n) + O(p)$$

Преобладающим вкладом будет $O\left(\frac{n}{p} \log \frac{n}{p}\right)$, так как для больших n и p остальные части оказываются значительно меньше, чем основной вклад от разделения и сортировки подмассивов. Итоговая сложность PSRS-сортировки будет:

$$O\left(\frac{n}{p} \log \frac{n}{p}\right)$$

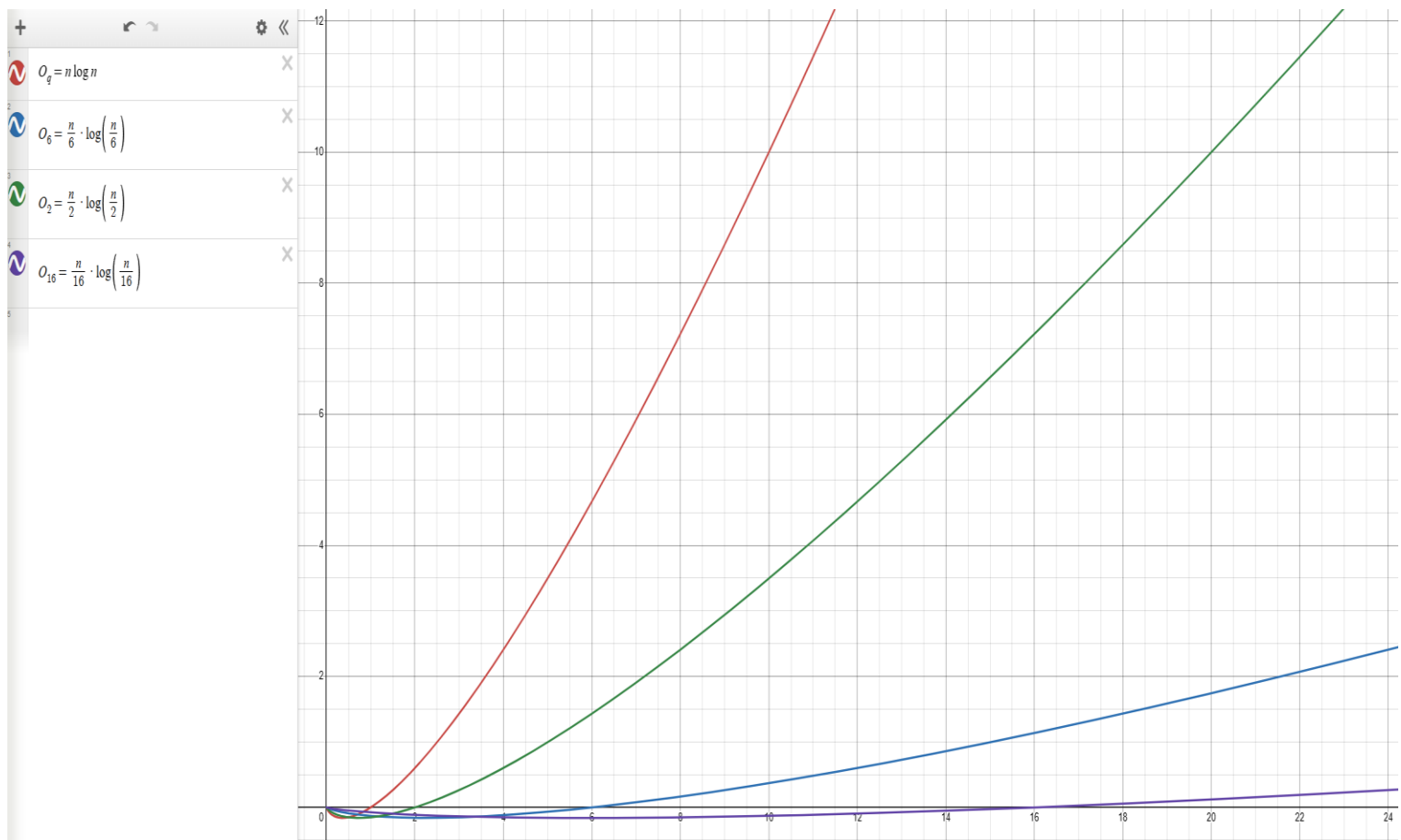


Рисунок 6 – Графики быстрой сортировки и PSRS-сортировок

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы была разработана программа для реализации PSRS-сортировки чисел, записанных в кольцевую очередь на базе массива, а также проведена оценка сложности алгоритма.

Алгоритм PSRS эффективно распределяет данные между несколькими потоками, снижая время сортировки за счёт параллельной обработки. Оценка сложности показала, что алгоритм имеет сложность $O\left(\frac{n}{p} \log \frac{n}{p}\right)$, что является хорошим результатом для алгоритмов сортировки в многозадачных системах.

Выполненная работа позволила освоить использование кольцевой очереди на базе массива и оценивать сложность алгоритмов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. [Circular buffer - Wikipedia](#) – 2025.
2. [Circular Array Implementation of Queue | GeeksforGeeks](#) – 2025.
3. [Круговая очередь - CodeChick](#) – 2025.
4. [PSRS-сортировка — Викиконспекты](#) – 2022.
5. H. Shi and J. Schaeffer. "Parallel Sorting by Regular Sampling. Journal of Parallel and Distributed Computing," 14(4):361--372, 1992. – URL : [psrs1.pdf - Yandex Documents](#)
6. .Stack Overflow – 2022. – URL : [Which parallel sorting algorithm has the best average case performance? - Stack Overflow](#)