Содержание

Введение

Данная работа направлена на практическое применение основных принципов программирования, которые помогут создавать чистый эффективный код, а также научат правильным методам разработки программного обеспечения.

Алгоритмы сортировки являются важным инструментом при работе с коллекциями данных, позволяя упорядочить элементы по определенному критерию. Они используются во множестве приложений, начиная от поиска, баз данных, алгоритмов машинного обучения до обработки изображений.

В основе любого алгоритма сортировки лежит стремление упорядочить элементы списка или массива. Существует множество различных подходов и стратегий для этого.

Некоторые алгоритмы сортировки работают, сравнивая элементы и меняя их местами при необходимости. Например, один из таких алгоритмов сортировки сравнивает пары элементов и, если они находятся в неправильном порядке, меняет их местами. Этот процесс повторяется до тех пор, пока весь список не будет упорядочен.

Другие алгоритмы используют метод "разделяй и властвуй", который разбивает список на более мелкие части, сортирует их отдельно, а затем объединяет в один отсортированный список. Этот подход обычно применяется для работы с большими наборами данных и обладает более эффективной сложностью.

Выбор алгоритма сортировки зависит от множества факторов, таких как размер данных, доступная память, тип элементов и требуемая эффективность. Некоторые алгоритмы лучше подходят для маленьких списков, в то время как другие могут эффективно работать с большими объемами данных.

Изучение и понимание различных алгоритмов сортировки позволяет разработчикам выбирать наиболее подходящий алгоритм для конкретной

задачи и улучшать производительность своих программ.

1. Постановка задачи

Необходимо разработать такую программу, которая позволит пользователю вводить массив для сортировки из файла и выводить отсортированный массив в файл. Пользователь сможет выбирать тип массива в консольном меню: случайный, отсортированный, в обратном порядке. Также можно будет выбирать размер массива, он будет неограничен. Необходимо разработать алгоритм быстрой сортировки, который будет сортировать массив чисел.

Алгоритм быстрой сортировки, или "Quicksort", - один из самых широко используемых алгоритмов сортировки. Он основан на стратегии "разделяй и властвуй", где исходный массив разбивается на меньшие подмассивы, которые сортируются рекурсивно. Алгоритм быстрой сортировки был первоначально предложен Тони Хоаром (TonyHoare) в 1960 году и до сих пор остается одним из самых эффективных алгоритмов сортировки.

Основная идея алгоритма быстрой сортировки заключается в том, чтобы выбрать опорный элемент из массива и разделить массив на две части: элементы, меньшие опорного, и элементы, большие опорного. Затем каждую из этих частей необходимо отсортировать рекурсивно. После сортировки каждой части массива, элементы объединяются вместе, чтобы получить отсортированный массив.

Выбор опорного элемента является ключевым шагом в алгоритме быстрой сортировки. Хороший выбор опорного элемента может значительно ускорить сортировку, в то время как плохой выбор может привести к значительному замедлению. Обычно для выбора опорного элемента используются различные стратегии, включая выбор случайного элемента, выбор среднего элемента или выбор первого, последнего или центрального элементов.

Алгоритм быстрой сортировки имеет несколько вариаций, которые могут быть использованы для улучшения его производительности в различных ситуациях. Одна из таких вариаций - это трехэтапный алгоритм быстрой сортировки, который использует сортировку вставками для подмассивов меньшего размера, чтобы избежать лишних вызовов рекурсивной функции сортировки.

1.1 Достоинства алгоритма

а) высокая скорость сортировки: Быстрая сортировка имеет среднюю сложность, что делает её одним из самых быстрых алгоритмов сортировки. Это означает, что время выполнения быстрой сортировки растет не так быстро, как у других алгоритмов сортировки, когда размер массива увеличивается. Это делает быструю сортировку отличным выбором для сортировки больших массивов.

б) сортировка на месте: Быстрая сортировка выполняется в том же массиве, что и изначальный. Это означает, что не нужно дополнительной памяти для создания нового массива, что делает ее эффективной для больших массивов. Кроме того, сортировка на месте означает, что она не изменяет порядок элементов в других массивах или структурах данных, которые могут ссылаться на те же объекты.

в) универсальность: Быстрая сортировка может быть использована для сортировки различных типов данных, включая числа, строки и объекты. Это делает ее универсальным алгоритмом сортировки, который может быть использован в различных приложениях.

г) легко реализуется: Алгоритм быстрой сортировки довольно прост в реализации, и его можно легко адаптировать для различных языков программирования. Кроме того, быстрая сортировка имеет множество оптимизаций и вариаций, которые могут быть использованы для улучшения ее производительности в конкретных случаях. Это делает быструю

сортировку популярным выбором для многих приложений, где требуется быстрая и эффективная сортировка.

1.2 Недостатки алгоритма

а) неустойчивость: Быстрая сортировка не гарантирует, что порядок элементов с одинаковыми значениями не изменится. Это может быть проблемой при сортировке объектов с несколькими свойствами. Например, если в массиве есть несколько объектов с одинаковым значением свойства, то после сортировки быстрой сортировкой порядок этих объектов может быть изменен.

б) худшая производительность в худшем случае: Если выбор опорного элемента происходит неудачно, то быстрая сортировка может работать очень медленно, что приводит к увеличенному времени выполнения. Это происходит, когда опорный элемент является самым большим или самым маленьким элементом в массиве, или когда массив уже отсортирован в обратном порядке. В таких случаях быстрая сортировка может стать медленнее других алгоритмов сортировки.

в) сложность для сортировки упорядоченных массивов: Если исходный массив уже отсортирован или почти отсортирован, то быстрая сортировка может замедлиться. Это происходит, потому что опорный элемент может быть выбран неудачно, что приводит к разделению массива на две части неравной длины. Это создаёт дополнительные сложности, если массивы, которые нужно отсортировать уже отсортированы.

г) требования к памяти: В худшем случае быстрая сортировка может потребовать дополнительной памяти для хранения стека вызовов рекурсивных функций. Это может быть проблемой для больших массивов или при ограниченно доступной памяти. Кроме того, быстрая сортировка не всегда является лучшим выбором для сортировки больших файлов или

потоков данных, поскольку она требует доступа ко всему файлу или потоку одновременно, чтобы разбить его на подмассивы.

1.3 Типичные сценарии применения

а) сортировка массивов данных: QuickSort является очень быстрым алгоритмом сортировки массивов данных и может использоваться во многих приложениях, где требуется сортировка больших объемов данных, таких как базы данных, поисковые системы и многие другие.

б) сортировка файлов: QuickSort может использоваться для сортировки больших файлов, где данные не могут поместиться в память целиком. Алгоритм может разбить файл на множество меньших частей, которые могут быть отсортированы в памяти, а затем объединены обратно в исходный файл.

в) поиск медианы: QuickSort может использоваться для поиска медианы (элемента, который разделяет массив на две равные части). Это может быть полезно, например, при определении цены на товары на основе их стоимости в разных магазинах.

г) поиск k-го наименьшего элемента: QuickSort также может использоваться для поиска k-го наименьшего элемента в массиве. Это может быть полезно, например, в медицине для поиска k-го наименьшего значения измерений пациента.

д) генерация случайных чисел: QuickSort может использоваться для генерации случайных чисел. Алгоритм может сортировать массив случайных чисел, а затем выбрать определенное количество элементов из отсортированного массива.

е) сжатие данных: QuickSort может использоваться для сжатия данных. Например, при сортировке цветов в изображении можно сгруппировать

пиксели одного цвета вместе и затем заменить их на один общий цвет, что может уменьшить объем данных.

ж) анализ данных: QuickSort может использоваться для анализа данных, например, для определения статистики на основе большого количества данных. Например, можно использовать QuickSort для сортировки массива продаж за определенный период и затем найти медиану, среднее значение или другие статистические показатели.

з) поиск дубликатов: QuickSort может использоваться для поиска дубликатов в массиве данных. Для этого массив сначала сортируется, а затем сканируется на наличие повторяющихся элементов.

и) обработка текстовых данных: QuickSort может использоваться для обработки текстовых данных, например, для сортировки слов в алфавитном порядке или для поиска наиболее часто встречающихся слов в тексте.

2. Выбор решения

Разработка велась в среде Microsoft Visual Studio на языке C.

Visual Studio — это интегрированная среда разработки (IDE) от компании Microsoft, предназначенная для создания различных типов приложений, включая веб-приложения, настольные приложения, мобильные приложения и игры. Она предоставляет разработчикам широкий набор инструментов и функций для упрощения процесса разработки и повышения производительности.

Visual Studio имеет большое количество плюсов, например, возможность устанавливать дополнительные расширения, которые позволяют расширить функциональность, поддержка многих языков программирования, интегрируемость с другими сервисами Microsoft.

Для разработки программы было решено выбрать язык C. Язык программирования C — это один из самых популярных языков программирования. Язык C, благодаря своей эффективности, часто используется для разработки приложений, требующих высокую производительность, например, игры и графические приложения.

3. Описание программы

Для проведения качественной тестировки необходимо точно измерить время выполнения программы. Оно измеряется с помощью функций из библиотеки time.h.

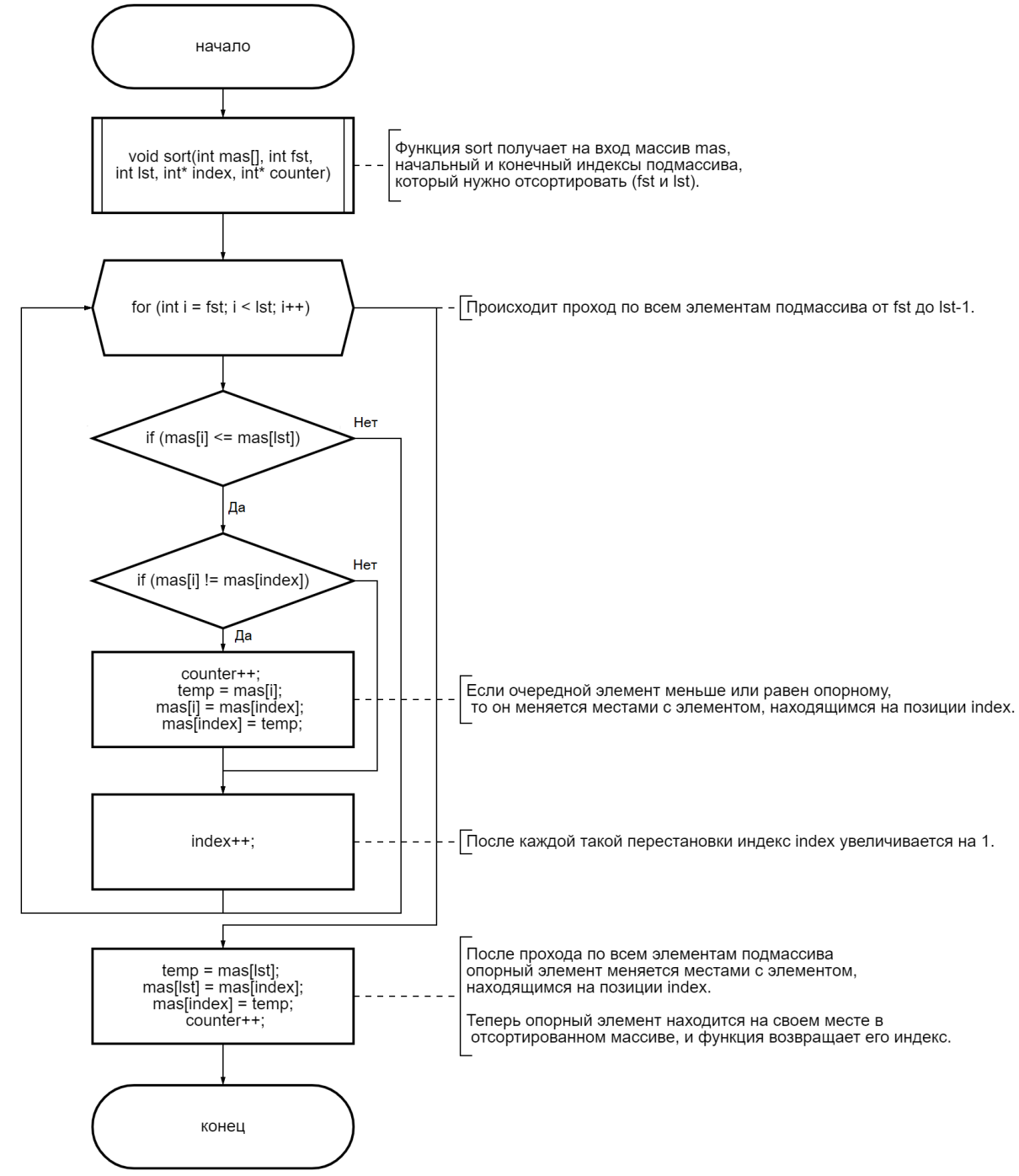
В начале функции start() вызывается функция clock(), которая возвращает количество тактов процессора, прошедших с начала работы программы. Затем выполняется алгоритм быстрой сортировки, после чего функция clock() вызывается снова, чтобы получить количество тактов, прошедших после выполнения сортировки. Вычисляется разница между двумя значениями, полученными от функций clock(), и делится на константу CLOCKS_PER_SEC, которая определяет количество тактов процессора в секунду. Таким образом, получается время выполнения алгоритма быстрой сортировки в секундах с точностью до миллисекунд.

```
void start() {
	int type = 4;
	int size = 1;
	int *arr;
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	printf("Введите кол-во элементов: ");
	scanf("%d", &size);
	do {
		printf("1 - случайный\n2 - отсортированный\n3 - в обратном порядке\nВыберите вид входного массива : ");
```

```
            scanf("%d", &type);
        } while (type > 3 && type < 1);

        arr = (int*)malloc(size * sizeof(int));

        createFile(size, type);
        scanFile(arr, size);

        clock_t start = clock();

        quick(arr, 0, size - 1);

        clock_t end = clock();
        double seconds = (double)(end - start) / CLOCKS_PER_SEC;

        writeFile(arr, size);
        free(arr);
        printf("Время выполнения %f секунд\nОпераций перестановок: %d\n",
seconds, counter);

    }
```

4. Схемы программы

4.1 Блок-схема алгоритма



В функции quick вызывается функция sort для подмассива от fst до index-1
и для подмассива от index+1 до lst.

Рекурсивные вызовы функции quick повторяются до тех пор,
пока размер подмассива не станет меньше или равен 1.

Рисунок 1 - Схема алгоритма

4.2 Блок-схема программы

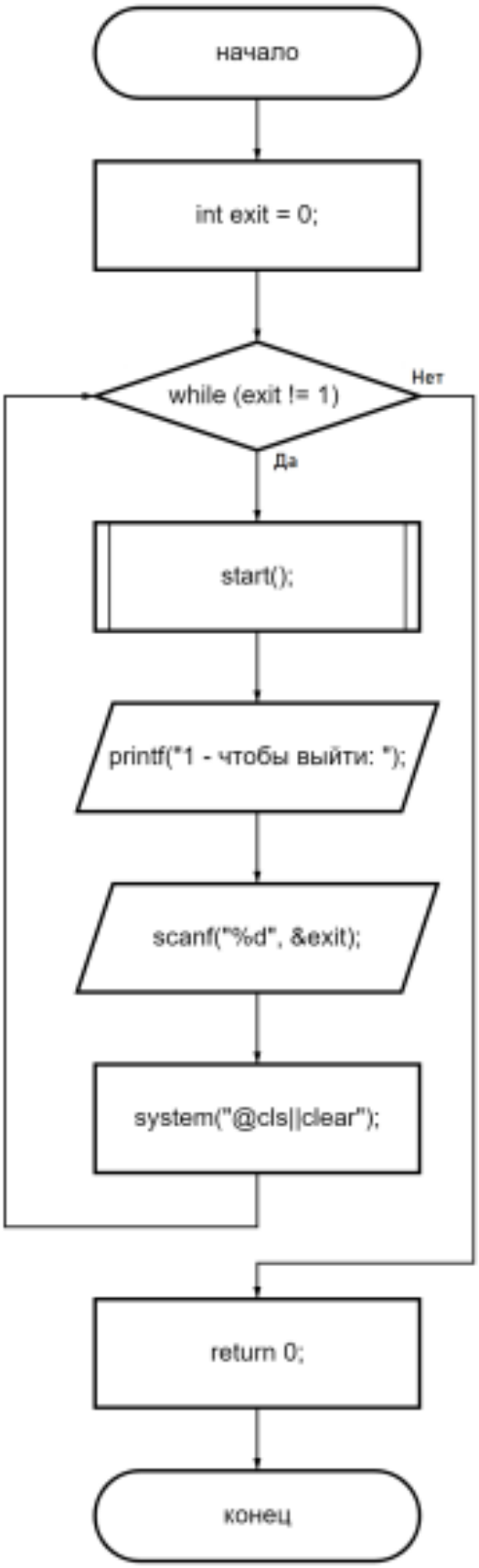


Рисунок 2 - Схема программы

5. Тестирование на разных наборах данных

Тестирование программы проводилось на случайном, отсортированном и отсортированном в обратном порядке наборах данных. Результаты тестирования представлены в таблицах и графиках, а также в приложении А на рисунках 6-11.

Таблица 1 – Тестирование со случайным набором данных

Число элементов	Время работы
10.000	0.001
20.000	0.002
30.000	0.004
40.000	0.006
50.000	0.007
60.000	0.009
70.000	0.011
80.000	0.013
90.000	0.015
100.000	0.018
110.000	0.02

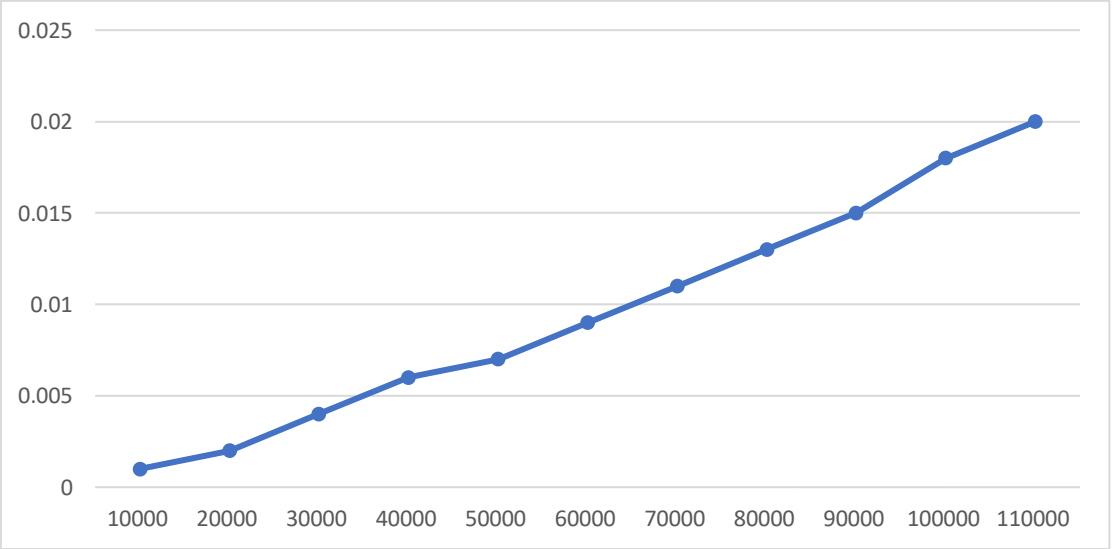


Таблица 2 – Тестирование с отсортированным набором данных

Число элементов	Время работы
10.000	0.134
20.000	0.535
30.000	1.198
40.000	2.137
50.000	3.370
60.000	4.847
70.000	6.527
80.000	8.541
90.000	10.806
100.000	13.314
110.000	15.988

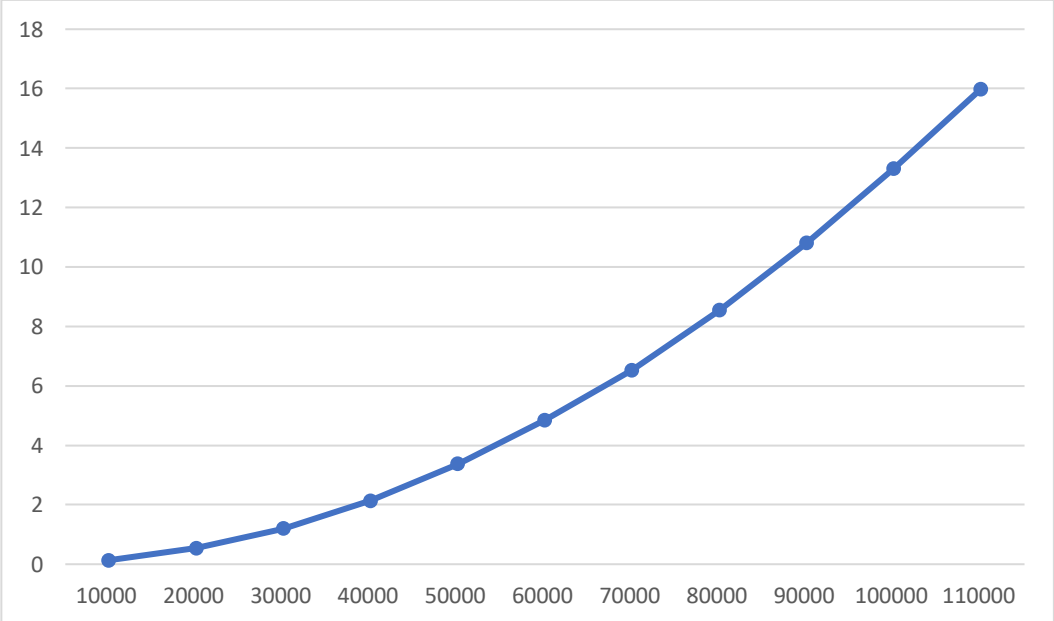
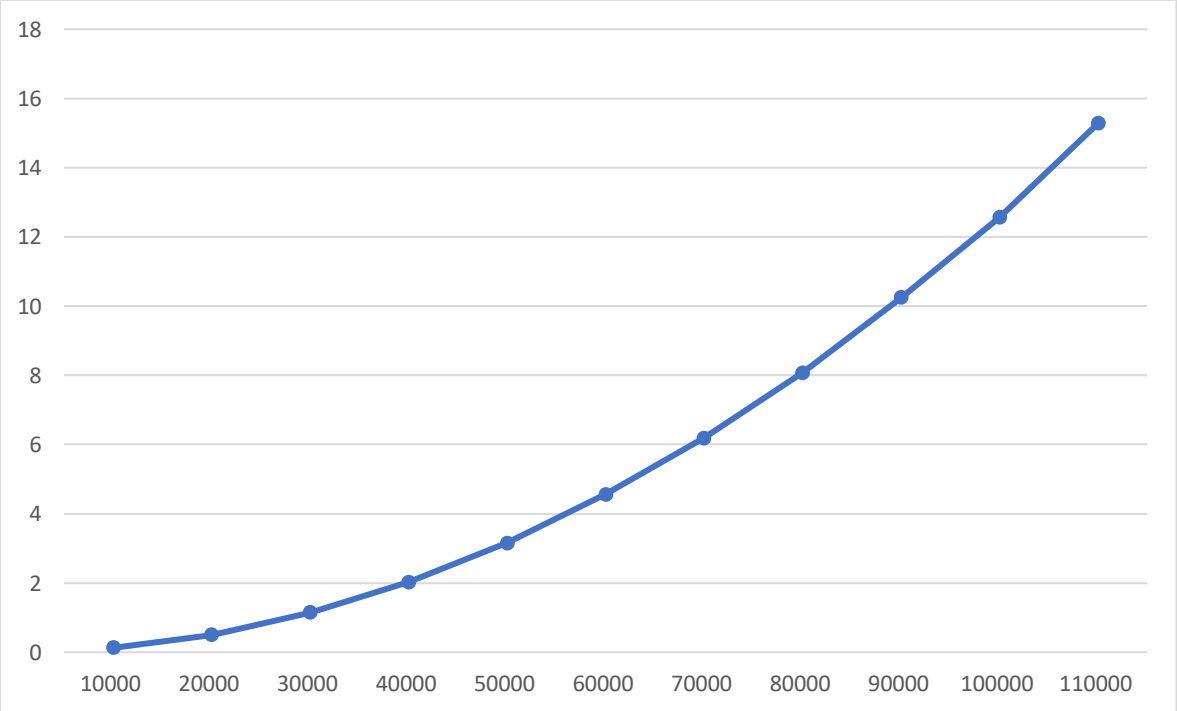


Таблица 3– Тестирование с набором данных, отсортированным в обратном порядке

Число элементов	Время работы
10.000	0.127
20.000	0.499
30.000	1.145
40.000	2.024
50.000	3.152
60.000	4.558
70.000	6.185
80.000	8.078
90.000	10.251
100.000	12.556
110.000	15.277



6. Отладка

Отладка программы была проведена с использованием средств, предоставляемых интегрированной средой разработки Microsoft Visual Studio 2022. В ходе отладки применялись такие возможности, как точка остановки, выполнение кода по шагам, анализ содержимого локальных и глобальных переменных.

В ходе тестирования программа подверглась изменению для того, чтобы в ней не было глобальных переменных. Также обнаружилась критическая ошибка переполнения стека. Было рассмотрено несколько способов решения проблемы, включая оптимизацию алгоритма быстрой сортировки, увеличение размера стека и использование других структур данных для сортировки, из них был выбран оптимальный.

После исправления всех выявленных ошибок и проведения дополнительных тестов можно считать, что программа работает корректно и соответствует поставленным требованиям.

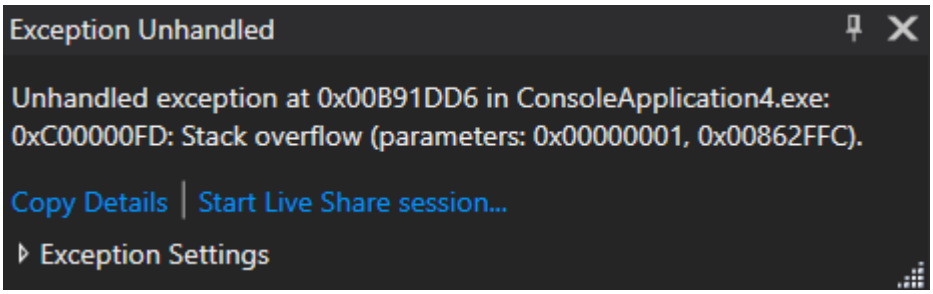


Рисунок 3 – Ошибка переполнения стека

7. Совместная работа

Работа над программой осуществлялась совместно, с использованием удаленного репозитория GitHub (github.com/Sabmyusel/Sort).

Алгоритм сортировки был реализован Черниковым К.А.

Алгоритм работы с файлами (создание файла с разными видами массивов, считывание из файла в массив и запись в файл) был написан Пяткиным М.Э.

Тестирование и замеры времени работы программы проводил Матросов А.М.

После написания алгоритма сортировки он был загружен на удаленный репозиторий:

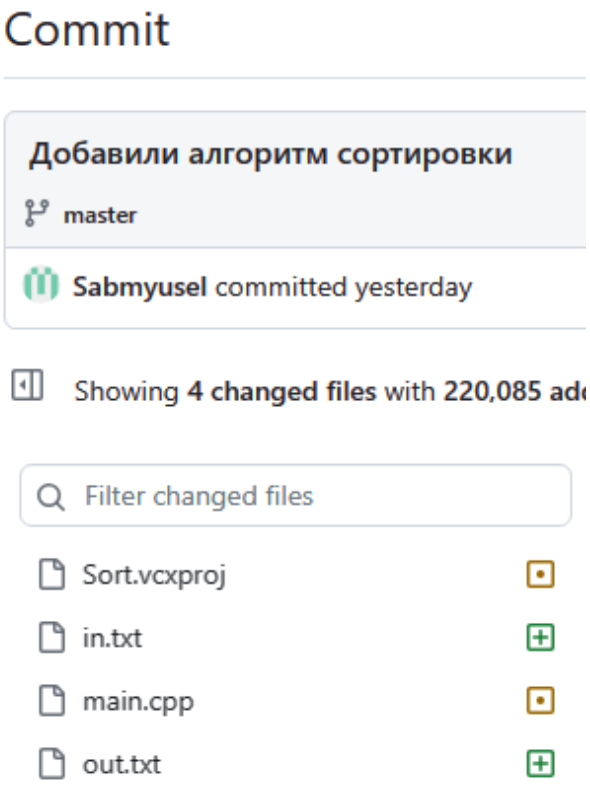


Рисунок 4 - Первый коммит

Далее, Пяткин М.Э. загрузил файлы проекта на компьютер и добавил функции для работы с файлами. После чего загрузил обновленный проект на репозиторий GitHub:

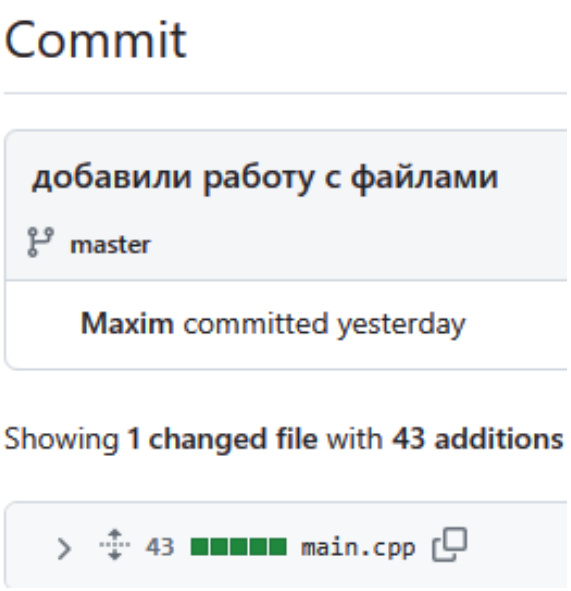


Рисунок 5 - Добавление работы с файлами

Матросов А.М. внес правки в финальный код программы, убрал глобальные переменные, поменял формат файлов ввода-вывода, изменил название массива.

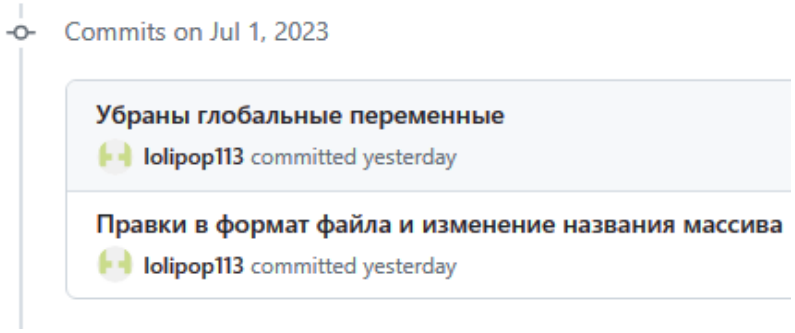


Рисунок 6 – финальные правки

Заключение

При выполнении работой были получены навыки разработки алгоритмов сортировки, навыки совместной работы с использованием удаленного репозитория GitHub. Был изучен алгоритм быстрой сортировки.

Я проводил тестирование алгоритма быстрой сортировки на различных наборах данных и анализировал результаты. Также я старался найти ошибки и дефекты в алгоритме, чтобы обеспечить его более эффективную работу.

В дальнейшем программу можно улучшить, добавив графический интерфейс и оптимизировав сортировку для малых наборов данных.

Список используемой литературы

1. И.С. Солдатенко. "Основы программирования на языке С" 2017 г.
2. Б.В. Керниган, Д.М. Ричи. “Язык программирования С”
3. Быстрая сортировка [Электронный ресурс] – URL: https://ru.wikipedia.org (дата обращения: 01.07.2023 г)
4. Роберт Седжвик. “Фундаментальные алгоритмы на С. Анализ. Структуры данных. Сортировка. Поиск. Алгоритмы на графах” (2003)

Приложение А. Тестирование программы

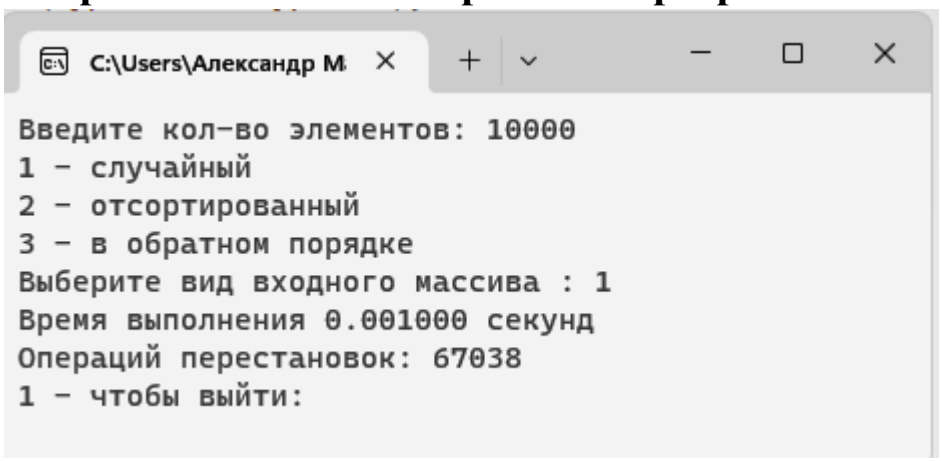


Рисунок 7 - Тестирование

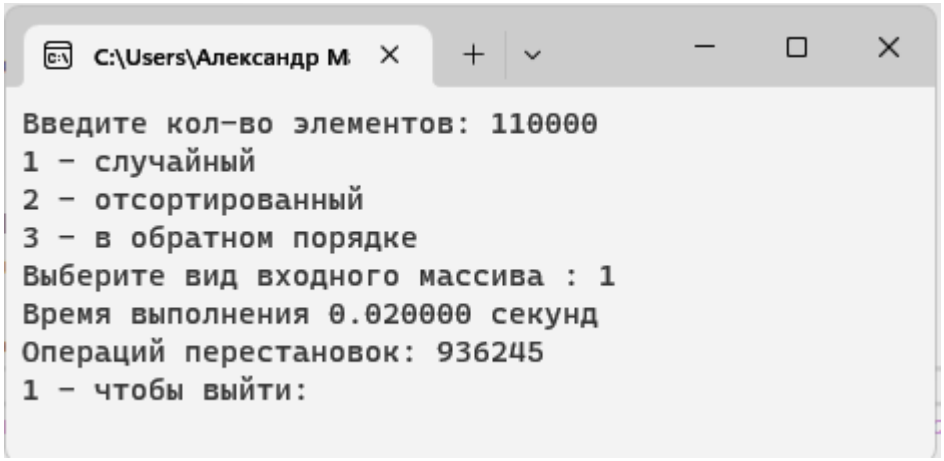


Рисунок 8 – Тестирование

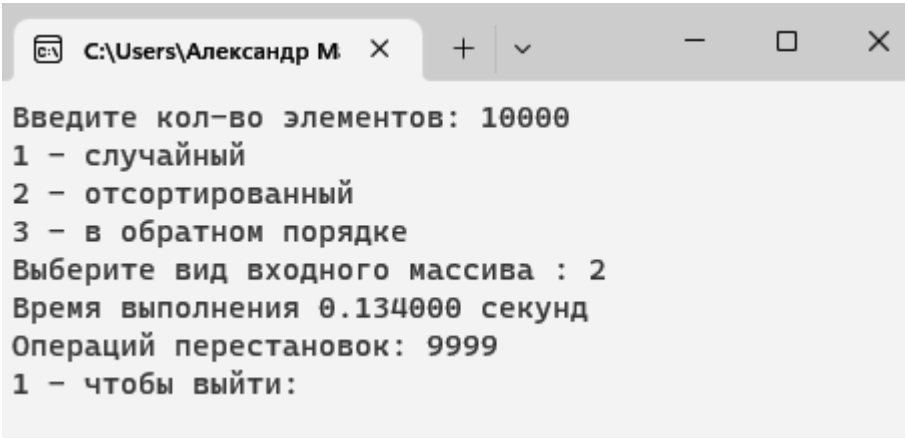


Рисунок 9 – Тестирование

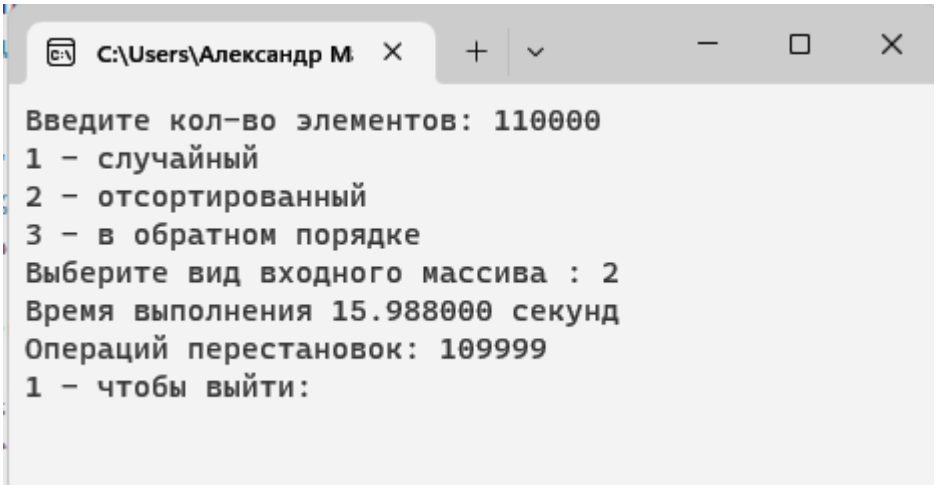


Рисунок 10 – Тестирование

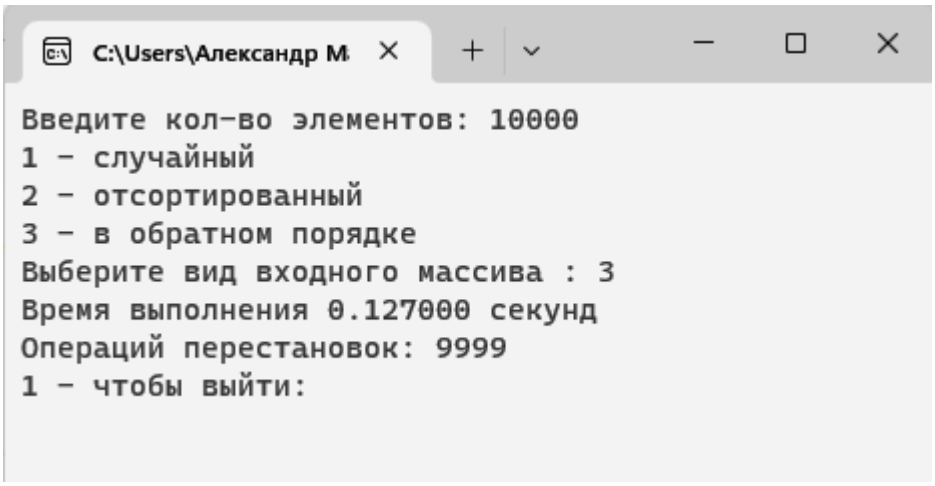


Рисунок 11 – Тестирование

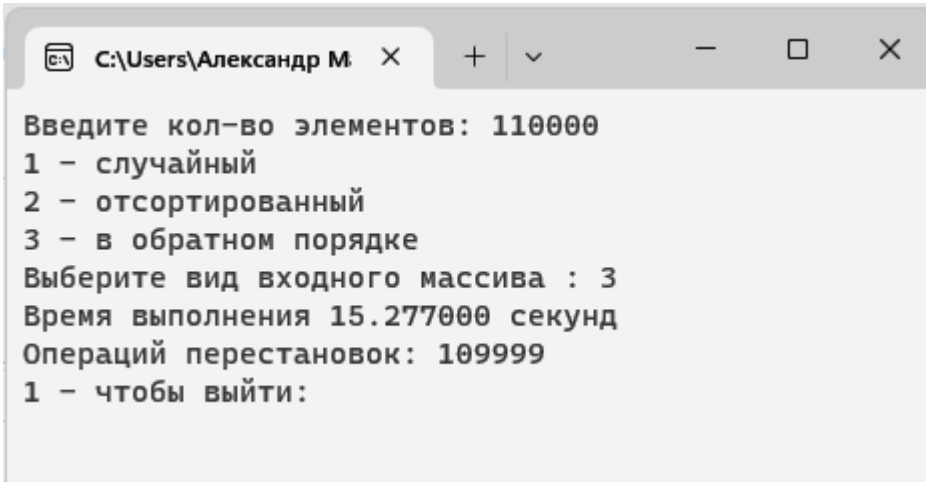


Рисунок 12 – Тестирование

Приложение Б. Листинг программы

```
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
void createFile(int amount, int type) {
	FILE* input = fopen("in.csv", "w");
	switch (type) {
	case 1:
		srand(time(NULL));
		for (int i = 0; i< amount; i++)
			fprintf(input, "%d\n", (rand() % 2001) - 1000);
		break;
	case 2:
		for (int i = 0; i< amount; i++)
			fprintf(input, "%d\n", i);
		break;
	case 3:
		int k = amount;
		for (int i = 0; i< amount; i++) {
			fprintf(input, "%d\n", k);
			k--;
		}
		break;
	}
	fclose(input);
}

void scanFile(int arr[], int amount) {
	FILE* input = fopen("in.csv", "r");

	for (int i = 0; i< amount; i++)
		fscanf(input, "%d\n", &arr[i]);

	fclose(input);
```

```
}

void writeFile(int arr[], int amount) {
	FILE* output = fopen("out.csv", "w");

	for (int i = 0; i< amount; i++)
		fprintf(output, "%d\n", arr[i]);

	fclose(output);
}

static int counter = 0;

int sort(int arr[], int fst, int lst) {

	int index = fst;
	int temp;

	for (int i = fst; i<lst; i++) {
		if (arr[i] <= arr[lst]) {
			if (arr[i] != arr[index])
			{
				counter++;
				temp = arr[i];
				arr[i] = arr[index];
				arr[index] = temp;
			}
			index++;
		}
	}
	temp = arr[lst];
	arr[lst] = arr[index];
	arr[index] = temp;
	counter++;
	return index;
}
```

```
void quick(int arr[], int fst, int lst) {



	if (fst>= lst)

		return;


	int index = sort(arr, fst, lst);


	quick(arr, fst, index - 1);

	quick(arr, index + 1, lst);

}


void start() {

	int type = 4;

	int size = 1;

	int *arr;

	SetConsoleCP(1251);

	SetConsoleOutputCP(1251);

	printf("Введитекол-воэлементов: ");

	scanf("%d", &size);

	do {

		printf("1 - случайный\n2 - отсортированный\n3 - в обратном
порядке\nВыберите вид входного массива : ");

		scanf("%d", &type);

	} while (type > 3 && type < 1);


	arr = (int*)malloc(size * sizeof(int));


	createFile(size, type);

	scanFile(arr, size);


	clock_t start = clock();


	quick(arr, 0, size - 1);


	clock_t end = clock();

	double seconds = (double)(end - start) / CLOCKS_PER_SEC;
```

```
	writeFile(arr, size);
	free(arr);
	printf("Время выполнения %f секунд\nОпераций перестановок: %d\n", seconds, counter);

}

int main() {
	int exit = 0;
	while (exit != 1) {
		start();
		printf("1 - чтобывыйти: ");
		scanf("%d", &exit);
		system("@cls||clear");
	}
	return 0;
}
```