# Санкт-Петербургский государственный политехнический университет

Кафедра “Прикладная математика и информатика”

Отчет по лабораторной работе 2 (вариант 10 (11))

“Алгоритмы и структуры данных”

Студент группы № 5030102/20001

ФИО: Дрекалов Никита Сергеевич

Выполнил 05.10.2023

Оглавление

[Санкт-Петербургский государственный политехнический университет 1](#_Toc147435906)

[**Постановка задачи** 3](#_Toc147435907)

[**Описание алгоритма** 3](#_Toc147435908)

[Алгоритм выполнения 15 варианта 3](#_Toc147435909)

[**Текст программы** 3](#_Toc147435910)

[main.cpp 4](#_Toc147435911)

[list.hpp 4](#_Toc147435912)

[**Описание тестирования** 3](#_Toc147435913)

# Постановка задачи

В лабораторной работе требуется на написанной ранее списке реализовать два способа представления графа – как матрицу смежностей и как список – и написать функцию постройки матрицы смежностей на основе списка смежностей (и наоборот…).

Пользователь программы может делать следующие запросы:

* Запустить сам алгоритм. Пользователю предложит ввести путь к файлу с матрицей/список смежностей и путь, в который будет записан обратный вариант.
* Вывести все запросы (на случай, если забудется одна из множества команд)
* Завершить программу.

Количество запросов со стороны пользователя неограниченно.

# Описание алгоритма

### Алгоритм построения списка смежностей из матрицы (10):

1. Считываем матрицу смежностей в переменную типа List<List<int>> с помощью функции readAdjMatrix().
2. Строим список смежностей с помощью функции asAdjList(). Для того, чтобы матрицу смежностей превратить в список, достаточно пройтись по строкам -> столбцам и добавлять в элемент, соответствующий столбцу, n раз элемент строки, где n – значение текущего элемента матрицы.
3. Записываем список в файл с помощью функции writeAdjList();

### Алгоритм построения матрицы смежностей из списка (10):

1. Считываем список смежностей в переменную типа List<List<int>> с помощью функции readAdjList().
2. Строим список смежностей с помощью функции asAdjMatrix(). Для того, чтобы список смежностей превратить в матрицу, достаточно пройтись по строкам -> столбцам матрицы и добавлять число, равное количеству встречаемости в списке у элемента, соответствующего столбцу, связей с элементом, соответствующего строке.
3. Записываем матрицу в файл с помощью функции writeAdjMatrix();

# Текст программы

Проект можно найти и посмотреть тут: [GitHub](https://github.com/Forthey/algorithms_ds/tree/main/task1)

### main.cpp

#include <fstream>

#include "list.hpp"

#pragma execution\_character\_set("utf-8")

int compareStrings(const std::string& s1, const std::string& s2) {

    return s1.compare(s2);

}

void test() {

    using namespace std;

    cout << "Тестовая программа считывает слова из words1.data и words2.data остортированно в списки, \nа затем записывает остортированный результирующий список в файл words12.data" << endl;

    cout << "Считывание списков в программу..." << endl;

    Alg::List<string> list1, list2;

    ifstream file1("files/words1.data"), file2("files/words2.data");

    if (!file1.is\_open() || !file2.is\_open()) {

        cout << "Не нашлись файлы \*.data для теста, завершение..." << endl;

        return;

    }

    list1.readSorted(file1, compareStrings);

    list2.readSorted(file2, compareStrings);

    file1.close();

    file2.close();

    cout << "Списки считаны" << endl;

    cout << "Список 1 отсортирован?: " << (list1.sorted(compareStrings) ? "true" : "false") << endl;

    cout << "Список 2 отсортирован?: " << (list2.sorted(compareStrings) ? "true" : "false") << endl;

    Alg::List<std::string> list12 = (list1 + list2).sort(compareStrings);

    cout << "Результирующий список отсортирован?: " << (list12.sorted(compareStrings) ? "true" : "false") << endl;

    cout << "Записывание результирующего списка в файл..." << endl;

    ofstream file("files/words12.data");

    if (!file.is\_open()) {

        cout << "Что - то помешало открыть файл words12.data, завершение..." << endl;

        return;

    }

    list12.display(file);

    file.close();

    cout << "Список записан" << endl;

}

void customFile() {

    using namespace std;

    string filename;

    cout << "Введите путь к первому файлу: ";

    cin >> filename;

    ifstream file1(filename);

    cout << "Введите путь ко второму файлу: ";

    cin >> filename;

    ifstream file2(filename);

    if (!file1.is\_open() || !file2.is\_open()) {

        cout << "Один из файлов не существует, выход..." << endl;

        return;

    }

    cout << "Считывание списков в программу..." << endl;

    Alg::List<string> list1, list2;

    list1.readSorted(file1, compareStrings);

    list2.readSorted(file2, compareStrings);

    file1.close();

    file2.close();

    cout << "Списки считаны" << endl;

    cout << "Список 1 отсортирован?: " << (list1.sorted(compareStrings) ? "true" : "false") << endl;

    cout << "Список 2 отсортирован?: " << (list2.sorted(compareStrings) ? "true" : "false") << endl;

    Alg::List<std::string> list12 = (list1 + list2).sort(compareStrings);

    cout << "Результирующий список отсортирован?: " << (list12.sorted(compareStrings) ? "true" : "false") << endl;

    cout << "display - вывести результирующий список" << endl;

    cout << "write - записать список в файл" << endl;

    cout << "back - выход" << endl;

    std::string response;

    cout << "> ";

    cin >> response;

    if (response == "display") {

        list12.display();

    }

    else if (response == "write") {

        cout << "Введите имя файла: ";

        cin >> filename;

        ofstream file(filename);

        if (!file.is\_open()) {

            cout << "Что-то помешало открыть файл, завершение..." << endl;

            return;

        }

        cout << "Записывание результирующего списка в файл..." << endl;

        list12.display(file);

        file.close();

        cout << "Список записан" << endl;

    }

    else if (response == "back") {

        return;

    }

}

void customConsole() {

    using namespace std;

    Alg::List<string> list1, list2;

    size\_t size;

    cout << "Создание первого списка" << endl;

    cout << "Введите число слов в списке: ";

    cin >> size;

    cout << "Введите " << size << " слов списка:" << endl;

    list1.readSorted(size, std::cin, compareStrings);

    cout << "Создание второго списка" << endl;

    cout << "Введите число слов в списке: ";

    cin >> size;

    cout << "Введите " << size << " слов списка : " << endl;

    list2.readSorted(size, std::cin, compareStrings);

    cout << "Списки считаны" << endl;

    cout << "Список 1 отсортирован?: " << (list1.sorted(compareStrings) ? "true" : "false") << endl;

    cout << "Список 2 отсортирован?: " << (list2.sorted(compareStrings) ? "true" : "false") << endl;

    Alg::List<std::string> list12 = (list1 + list2).sort(compareStrings);

    cout << "Результирующий список отсортирован?: " << (list12.sorted(compareStrings) ? "true" : "false") << endl;

    cout << "display - вывести результирующий список" << endl;

    cout << "write - записать список в файл" << endl;

    cout << "back - выход" << endl;

    std::string response;

    cout << "> ";

    cin >> response;

    if (response == "display") {

        list12.display();

    }

    else if (response == "write") {

        std::string filename;

        cout << "Введите имя файла: ";

        cin >> filename;

        ofstream file(filename);

        if (!file.is\_open()) {

            cout << "Что-то помешало открыть файл, завершение..." << endl;

            return;

        }

        cout << "Записывание результирующего списка в файл..." << endl;

        list12.display(file);

        file.close();

        cout << "Список записан" << endl;

    }

    else if (response == "back") {

        return;

    }

}

void custom() {

    using namespace std;

    cout << "file - считывать файлов" << endl;

    cout << "console - считать из консоли" << endl;

    cout << "back - вернуться в основное меню" << endl;

    std::string response;

    cout << "> ";

    cin >> response;

    if (response == "file") {

        customFile();

    }

    else if (response == "console") {

        customConsole();

    }

    else if (response == "back") {

        return;

    }

    else {

        cout << "Неизвестная команда" << endl;

    }

}

void printMenuHelp() {

    using namespace std;

    cout << "test - Запустить тестовую программу" << endl;

    cout << "custom - Использовать собственные параметры" << endl;

    cout << "help - вывести это окно помощи" << endl;

    cout << "exit - Выйти" << endl;

}

void menu() {

    using namespace std;

    cout << "<< N1. Работа со списком (вариант 15) >>" << endl;

    printMenuHelp();

    bool isOpened = true;

    std::string response;

    while (isOpened) {

        cout << "> ";

        cin >> response;

        if (response == "test") {

            test();

        }

        else if (response == "custom") {

            custom();

        }

        else if (response == "help") {

            printMenuHelp();

        }

        else if (response == "exit") {

            isOpened = false;

        }

        else {

            cout << "Неизвестная команда (\"help\" - вывести список всех команд)" << endl;

        }

    }

}

int main() {

    std::locale::global(std::locale("ru\_RU.utf8"));

    menu();

    return 0;

}

### list.hpp

#pragma once

#include <iostream>

#include <string>

namespace Alg {

    /\*

     \* Default functions

     \*/

    template<typename Data>

    int defaultCompareFunc(const Data &data1, const Data &data2) {

        return data1 - data2;

    }

    template<typename Data>

    void defaultDisplayFunc(std::ostream &out, const Data &data) {

        out << data;

    }

    template<typename Data>

    Data defaultParseFunc(std::istream& in) {

        Data data;

        in >> data;

        return data;

    }

    /\*

     \* Class List declaration

     \*/

    template <typename Data>

    class List {

    public:

        explicit List();

        List(const List<Data> &list);

        List<Data>& pushBack(const Data& data);

        List<Data>& pushBack(const List<Data>& list);

        List<Data>& pushFront(const Data& data);

        List<Data>& pushFront(const List<Data>& list);

        List<Data>& pushAt(size\_t index, const Data& data);

        List<Data>& pushAt(size\_t index, const List<Data>& list);

        List<Data>& pushSorted(const Data& data, int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc);

        List<Data>& pushSorted(const List<Data>& list, int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc);

        List<Data>& popBack();

        List<Data>& popBack(Data& oldData);

        List<Data>& popFront();

        List<Data>& popFront(Data& oldData);

        List<Data>& popAt(size\_t index);

        List<Data>& popAt(size\_t index, Data& oldData);

        void clear();

        bool empty() const;

        size\_t length() const;

        bool sorted(int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc) const;

        void display(std::ostream& out = std::cout, void(&displayFunc)(std::ostream &, const Data &) = defaultDisplayFunc) const;

        List<Data>& read(const size\_t size, std::istream& in = std::cin, Data(&parseFunc)(std::istream&) = defaultParseFunc);

        List<Data>& readSorted(const size\_t size, std::istream& in = std::cin, int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc, Data(&parseFunc)(std::istream&) = defaultParseFunc);

        List<Data>& read(std::istream& in = std::cin, Data(&parseFunc)(std::istream&) = defaultParseFunc);

        List<Data>& readSorted(std::istream& in = std::cin, int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc, Data(&parseFunc)(std::istream&) = defaultParseFunc);

        size\_t find(const Data& data) const;

        // List<Data>& findAll(const Data& data);

        Data& min(int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc) const;

        Data& max(int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc) const;

        List<Data>& sort(int(&compareFunc)(const Data&, const Data&) = defaultCompareFunc);

        List<Data>& operator<<(const Data& data);

        List<Data>& operator<<(const List<Data>& list);

        List<Data>& operator>>(const Data& data);

        List<Data> operator+(const List<Data>& list) const;

        Data& operator[](size\_t index) const;

        ~List();

    private:

        class ListElement {

        public:

            explicit ListElement(Data data, ListElement\* next) : data(data), next(next) {};

            Data data;

            ListElement\* next;

        } \*head;

        size\_t len;

        Data& min(int(&compareFunc)(const Data&, const Data&), size\_t a, size\_t b);

        Data& max(int(&compareFunc)(const Data&, const Data&), size\_t a, size\_t b);

        void swap(Data& data1, Data& data2);

    };

    /\*

     \* Class List's functions definitions

     \*/

    template<typename Data>

    List<Data>::List() {

        head = nullptr;

        len = 0;

    }

    template<typename Data>

    List<Data>::List(const List<Data> &list) {

        head = nullptr;

        len = 0;

        ListElement\* listElement = list.head;

        for (size\_t i = 0; i < list.len; i++) {

            pushBack(listElement->data);

            listElement = listElement->next;

        }

    }

    template<typename Data>

    List<Data>::~List() {

        clear();

    }

    template<typename Data>

    List<Data>& List<Data>::pushBack(const Data& data) {

        ListElement\* newListElement = new ListElement(data, nullptr);

        if (newListElement == nullptr)

            return \*this;

        if (head == nullptr) {

            head = newListElement;

            len++;

            return \*this;

        }

        ListElement\* listElement = head;

        for (size\_t i = 0; i < len - 1; i++)

            listElement = listElement->next;

        listElement->next = newListElement;

        len++;

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::pushBack(const List<Data>& list) {

        ListElement\* listElement = list.head;

        for (size\_t i = 0; i < list.len; i++) {

            (\*this).pushBack(listElement->data);

            listElement = listElement->next;

        }

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::pushFront(const Data& data) {

        ListElement\* newListElement = new ListElement(data, head);

        if (newListElement == nullptr)

            return \*this;

        len++;

        head = newListElement;

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::pushFront(const List<Data>& list) {

        ListElement\* listElement = list.head;

        for (size\_t i = 0; i < list.len; i++) {

            (\*this).pushAt(i, listElement->data);

            listElement = listElement->next;

        }

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::pushAt(size\_t index, const Data& data) {

        if (head == nullptr) {

            return pushBack(data);

        }

        if (index >= len) {

            index = len - 1;

        }

        if (index == 0) {

            return pushFront(data);

        }

        ListElement\* listElement = head;

        for (size\_t i = 0; i < index - 1; i++) {

            listElement = listElement->next;

        }

        ListElement\* newListElement = new ListElement(data, listElement->next);

        if (newListElement == nullptr)

            return \*this;

        listElement->next = newListElement;

        len++;

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::pushAt(size\_t index, const List<Data>& list) {

        ListElement\* listElement = list.head;

        for (size\_t i = 0; i < list.len; i++) {

            (\*this).pushAt(i + index, listElement->data);

            listElement = listElement->next;

        }

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::pushSorted(const Data& data, int(&compareFunc)(const Data&, const Data&)) {

        if (head == nullptr) {

            return pushBack(data);

        }

        ListElement\* listElement = head;

        for (size\_t i = 0; i < len; i++) {

            if (compareFunc(listElement->data, data) > 0) {

                return pushAt(i, data);

            }

            listElement = listElement->next;

        }

        return pushBack(data);

    }

    template<typename Data>

    List<Data>& List<Data>::pushSorted(const List<Data>& list, int(&compareFunc)(const Data&, const Data&)) {

        ListElement\* listElement = list.head;

        for (size\_t i = 0; i < list.len; i++) {

            pushSorted(listElement->data, compareFunc);

            listElement = listElement->next;

        }

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::popBack(Data& oldData) {

        if (len == 0)

            return \*this;

        ListElement\* listElement = head;

        ListElement\* previousListElement = nullptr;

        for (size\_t i = 0; i < len - 1; i++) {

            previousListElement = listElement;

            listElement = listElement->next;

        }

        oldData = listElement->data;

        delete listElement;

        if (previousListElement != nullptr)

            previousListElement->next = nullptr;

        else

            head = nullptr;

        len--;

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::popBack() {

        Data tmp;

        return popBack(tmp);

    }

    template<typename Data>

    List<Data>& List<Data>::popFront(Data& oldData) {

        if (head == nullptr)

            return \*this;;

        ListElement\* oldElement = head;

        oldData = oldElement->data;

        head = head->next;

        delete oldElement;

        len--;

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::popFront() {

        Data tmp;

        return popFront(tmp);

    }

    template<typename Data>

    List<Data>& List<Data>::popAt(size\_t index, Data& oldData) {

        if (head == nullptr)

            return \*this;

        if (index >= len)

            index = len - 1;

        if (len == 1)

            return popFront(oldData);

        ListElement\* listElement = head;

        for (size\_t i = 0; i < index - 1; i++)

            listElement = listElement->next;

        ListElement\* oldElement = listElement->next;

        listElement->next = oldElement->next;

        delete oldElement;

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::popAt(size\_t index) {

        Data tmp;

        return popAt(index, tmp);

    }

    template<typename Data>

    void List<Data>::clear() {

        while (!popFront().empty());

    }

    template<typename Data>

    bool List<Data>::empty() const {

        return len == 0;

    }

    template<typename Data>

    size\_t List<Data>::length() const {

        return len;

    }

    template<typename Data>

    bool List<Data>::sorted(int(&compareFunc)(const Data&, const Data&)) const {

        ListElement\* listElement = head;

        for (size\_t i = 0; i < len - 1; i++) {

            if (compareFunc(listElement->data, listElement->next->data) > 0) {

                return false;

            }

            listElement = listElement->next;

        }

        return true;

    }

    template<typename Data>

    void List<Data>::display(std::ostream& out, void(&displayFunc)(std::ostream&, const Data&)) const {

        ListElement\* element = head;

        if (len == 0) {

            out << "<Empty list>" << std::endl;

        }

        for (size\_t i = 0; i < len && element != nullptr; i++) {

            displayFunc(out, element->data);

            out << std::endl;

            element = element->next;

        }

        out << std::endl;

    }

    template<typename Data>

    List<Data>& List<Data>::read(const size\_t size, std::istream& in, Data(&parseFunc)(std::istream&)) {

        for (size\_t i = 0; i < size; i++) {

            \*this << parseFunc(in);

        }

        in.eof();

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::readSorted(const size\_t size, std::istream& in,int(&compareFunc)(const Data&, const Data&), Data(&parseFunc)(std::istream&)) {

        for (size\_t i = 0; i < size; i++) {

            Data data = parseFunc(in);

            pushSorted(data, compareFunc);

        }

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::read(std::istream& in, Data(&parseFunc)(std::istream&)) {

        if (in.eof())

            return \*this;

        do {

            \*this << parseFunc(in);

        } while (!in.eof());

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::readSorted(std::istream& in, int(&compareFunc)(const Data&, const Data&), Data(&parseFunc)(std::istream&)) {

        if (in.eof())

            return \*this;

        do {

            Data data = parseFunc(in);

            pushSorted(data, compareFunc);

        } while (!in.eof());

        return \*this;

    }

    template<typename Data>

    size\_t List<Data>::find(const Data& data) const {

        ListElement\* listElement = head;

        size\_t index;

        for (index = 0; index < len; index++) {

            if (listElement->data == data)

                break;

            listElement = listElement->next;

        }

        return index;

    }

    template<typename Data>

    Data& List<Data>::min(int(&compareFunc)(const Data&, const Data&)) const {

        return min(compareFunc, 0, len);

    }

    template<typename Data>

    Data& List<Data>::max(int(&compareFunc)(const Data&, const Data&)) const {

        return max(compareFunc, 0, len);

    }

    template<typename Data>

    Data& List<Data>::min(int(&compareFunc)(const Data&, const Data&), size\_t a, size\_t b) {

        if (len == 0 || a >= b)

            throw std::runtime\_error("Cannot find the max element of an empty list");

        ListElement\* listElement = head;

        for (size\_t i = 0; i < a; i++)

            listElement = listElement->next;

        Data\* min = &listElement->data;

        listElement = listElement->next;

        for (size\_t i = a + 1; i < b; i++) {

            if (compareFunc(listElement->data, \*min) < 0)

                min = &listElement->data;

            listElement = listElement->next;

        }

        return \*min;

    }

    template<typename Data>

    Data& List<Data>::max(int(&compareFunc)(const Data&, const Data&), size\_t a, size\_t b) {

        if (len == 0 || a >= b)

            throw std::runtime\_error("Cannot find the max element of an empty list");

        ListElement\* listElement = head;

        for (size\_t i = 0; i < a; i++)

            listElement = listElement->next;

        Data\* max = &listElement->data;

        listElement = listElement->next;

        for (size\_t i = a + 1; i < b; i++) {

            if (compareFunc(listElement->data, \*max) > 0)

                max = &listElement->data;

            listElement = listElement->next;

        }

        return \*max;

    }

    template<typename Data>

    void List<Data>::swap(Data& data1, Data& data2) {

        Data tmp = data1;

        data1 = data2;

        data2 = tmp;

    }

    template<typename Data>

    List<Data>& List<Data>::sort(int(&compareFunc)(const Data&, const Data&)) {

        if (len == 0)

            return \*this;

        // Selection sort

        ListElement\* listElement = head;

        for (size\_t i = 0; i < len; i++) {

            swap(listElement->data, min(compareFunc, i, len));

            listElement = listElement->next;

        }

        return \*this;

    }

    template<typename Data>

    List<Data>& List<Data>::operator<<(const Data& data) {

        return pushBack(data);

    }

    template<typename Data>

    List<Data>& List<Data>::operator<<(const List<Data>& list) {

        return pushBack(list);

    }

    template<typename Data>

    List<Data>& List<Data>::operator>>(const Data& data) {

        return popBack(data);

    }

    template<typename Data>

    List<Data> List<Data>::operator+(const List<Data>& list) const {

        List<Data> resultList(\*this);

        resultList.pushBack(list);

        return resultList;

    }

    template<typename Data>

    Data& List<Data>::operator[](size\_t index) const {

        if (index >= len)

            index = len - 1;

        ListElement\* listElement = head;

        for (size\_t i = 0; i < index; i++)

            listElement = listElement->next;

        return listElement->data;

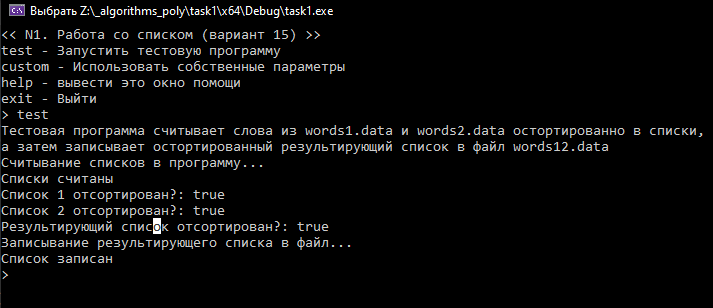
    }

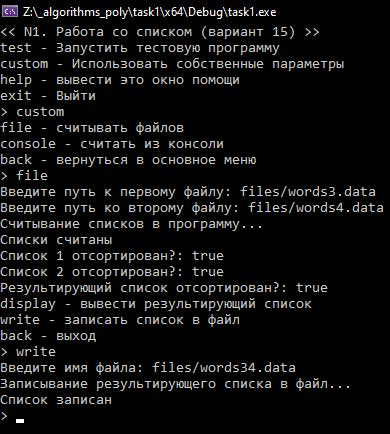
}

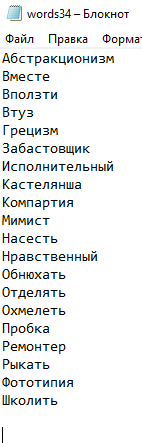
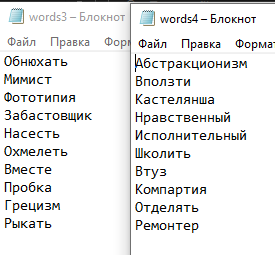
# Описание тестирования

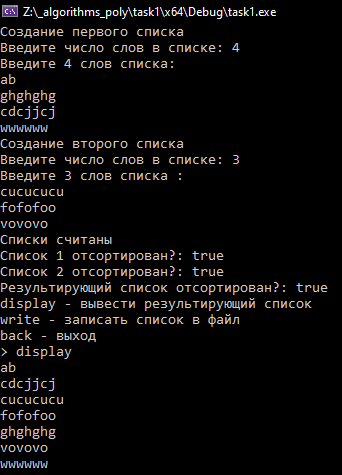
Протестировать успешное выполнение программы без вывода списка и визуального оценивания помогает метод sorted(). Для тестирования данной программы были созданы 2 файла: words1.data на 1000 слов и words2.data на 1330 слов, а результат выводился в файл words12.data. Также функционал позволял вводить данные из других файлов и их консоли, что позволило на более простых примерах еще раз проверить работоспособность программы. Проверялось:

* Стабильность работы программы при одинаковых входных данных;
* Отсутствие “падений” и “зависаний”;
* Корректное выполнение всех заявленных процедур;
* Корректное завершение программы;

Приведем пример тестирования на скриншотах, данных ниже







При проведении тестирования такого рода никаких проблем обнаружено не было, что позволяет судить о корректности работы программы в целом.