

Министерство науки и высшего образования Российской Федерации
Казанский национальный исследовательский технический университет –
КАИ им. А.Н. Туполева

Институт компьютерных технологий и защиты информации
Отделение СПО ИКТЗИ «Колледж информационных технологий»

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЕ

Методические указания к лабораторной работе №10

Тема: «Внутренняя сортировка данных»

Казань 2022

Составитель преподаватель СПО ИКТЗИ Мингалиев Заид Зульфатович

Методические указания к лабораторным работам по дисциплине «ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЕ» предназначены для студентов направления подготовки 09.02.07 «Информационные системы и программирование»

КНИТУ-КАИ, 2022

ПРОЦЕСС СДАЧИ ВЫПОЛНЕННОЙ РАБОТЫ

По итогам выполнения работы студент:

1. демонстрирует преподавателю правильно работающие программы;
2. демонстрирует приобретённые знания и навыки отвечает на пару небольших вопросов преподавателя по составленной программе, возможностям её доработки;
3. демонстрирует отчет по выполненной лабораторной работе.

Итоговая оценка складывается из оценок по трем указанным составляющим.

ЛАБОРАТОРНАЯ РАБОТА №10

ТЕМА: «ВНУТРЕННЯЯ СОРТИРОВКА ДАННЫХ»

ЦЕЛЬ РАБОТЫ

Изучение простейших и улучшенных методов сортировки и особенностей их программной реализации.

ХОД РАБОТЫ

1. Метод обмена

Пусть имеется n элементов $a_1, a_2, a_3, \dots, a_n$, расположенных в ячейках массива. Для простоты будем считать, что сам элемент совпадает с его ключом. Алгоритм состоит в повторении $n - 1$ шага, на каждом из которых в оставшемся необработанном наборе за счет попарного сравнения соседних элементов отыскивается минимальный элемент.

Шаг 1. Сравниваем a_n с a_{n-1} и если $a_n < a_{n-1}$ то меняем их местами, потом сравниваем a_{n-1} с a_{n-2} и, возможно, переставляем их, сравниваем a_{n-2} и a_{n-3} и т.д. до сравнения и, возможно, перестановки a_2 и a_1 . В результате на первом месте в массиве оказывается самый минимальный элемент, который в дальнейшей сортировке не участвует

Шаг 2. Аналогично сравниваем a_n с a_{n-1} , a_{n-1} с a_{n-2} и т.д., a_3 с a_2 , в результате чего на месте a_2 оказывается второй наименьший элемент, который вместе с a_1 образует начальную часть упорядоченного массива

Шаг 3. Аналогичными сравнениями и перестановками среди элементов a_3, a_4, \dots, a_n находится наименьший, который занимает место a_3

...

Шаг $n-1$. К этому моменту первые $n - 2$ элемента в массиве уже упорядочены и остается «навести порядок» только между двумя последними элементами a_{n-1} и a_n . На этом сортировка заканчивается.

Программная реализация включает двойной цикл: внешний реализует основные шаги алгоритма, внутренний сравнивает и переставляет элементы, начиная с конца массива.

Пример программной реализации сортировки методом обмена представлен в листинге 10.1.

Листинг 10.1 – Сортировка методом обмена

1	static int[] BubbleSort(int[] mas)
2	{
3	int temp;
4	for (int i = 0; i < mas.Length; i++)
5	{
6	for (int j = i + 1; j < mas.Length; j++)
7	{
8	if (mas[i] > mas[j])
9	{
10	temp = mas[i];
11	mas[i] = mas[j];
12	mas[j] = temp;
13	}
14	}
15	}
16	return mas;
17	}

2. Метод вставок

Пусть имеется n элементов $a_1, a_2, a_3, \dots, a_n$, расположенных в ячейках массива. Сортировка выполняется за $(n-1)$ шаг, причем шаги удобно нумеровать от 2 до n . На каждом i -ом шаге обрабатываемый набор разбивается на 2 части:

- левую часть образуют уже упорядоченные на предыдущих шагах элементы $a_1, a_2, a_3, \dots, a_{i-1}$
- правую часть образуют еще не обработанные элементы $a_i, a_{i+1}, a_{i+2}, \dots, a_n$

На шаге i для элемента a_i находится подходящее место в уже отсортированной последовательности. Поиск подходящего места выполняется поэлементными сравнениями и перестановками по необходимости: сравниваем a_i с a_{i-1} , если $a_i < a_{i-1}$, то переставляем их, потом сравниваем a_{i-1} с a_{i-2} и т. д. Сравнения и, возможно, перестановки продолжаются до тех пор, пока не будет выполнено одно из 2-х следующих условий:

- в отсортированном наборе найден элемент, меньший a_i (все остальные не просмотренные элементы будут еще меньше);
- достигнут первый элемент набора a_1 , что произойдет в том случае, если a_i меньше всех элементов в отсортированном наборе и он должен занять первое место в массиве.

Программная реализация включает два вложенных цикла, но в отличие от предыдущего метода, внутренний цикл реализуется как **while** с возможностью остановки при обнаружении меньшего элемента.

Пример программной реализации сортировки методом вставок представлен в листинге 10.2.

Листинг 10.2 – Сортировка методом вставок

1	<code>void InsertionSort(int n, int[] mass)</code>
2	<code>{</code>
3	<code>int newElement, location;</code>
4	<code>for (int i = 1; i < n; i++)</code>
5	<code>{</code>

6	<code>newElement = mass[i];</code>
7	<code>location = i - 1;</code>
8	<code>while(location >= 0 && mass[location] > newElement)</code>
9	<code>{</code>
10	<code>mass[location + 1] = mass[location];</code>
11	<code>location = location - 1;</code>
12	<code>}</code>
13	<code>mass[location + 1] = newElement;</code>
14	<code>}</code>
15	<code>}</code>

3. Метод выбора

Пусть имеется n элементов $a_1, a_2, a_3, \dots, a_n$, расположенных в ячейках массива. Сортировка выполняется за $(n-1)$ шаг, пронумерованных от 1 до $n-1$.

1. На каждом i -ом шаге обрабатываемый набор разбивается на 2 части:

- левую часть образуют уже упорядоченные на предыдущих шагах элементы $a_1, a_2, a_3, \dots, a_{i-1}$
- правую часть образуют еще не обработанные элементы $a_i, a_{i+1}, a_{i+2}, \dots, a_n$

Суть метода состоит в том, что в необработанном наборе отыскивается наименьший элемент, который меняется местами с элементом a_i . На первом шаге (при $i = 1$), когда необработанным является весь исходный набор, это сводится к поиску наименьшего элемента в массиве и обмену его с первым элементом. Ясно, что поиск наименьшего элемента выполняется обычным попарным сравнением, но соседние элементы при этом не переставляются, что в целом уменьшает число пересылок.

Пример программной реализации сортировки методом выбора представлен в листинге 10.3.

Листинг 10.3 – Сортировка методом выбора

1	static int[] ViborSort(int[] mas)
2	{
3	for (int i = 0; i < mas.Length - 1; i++)
4	{
5	//поиск минимального числа
6	int min=i;
7	for (int j = i + 1; j < mas.Length; j++)
8	{
9	if (mas[j] < mas[min])
10	{
11	min = j;
12	}
13	}
14	//обмен элементов
15	int temp = mas[min];
16	mas[min] = mas[i];
17	mas[i] = temp;

18	}
19	return mas;
20	}

4. Метод Шелла

Метод Шелла является улучшенным вариантом метода вставок. Поскольку метод вставок дает хорошие показатели качества для небольших или почти упорядоченных наборов данных, метод Шелла использует эти свойства за счет многократного применения метода вставок.

Алгоритм метода Шелла состоит в многократном повторении двух основных действий:

- **объединение** нескольких элементов исходного массива по некоторому правилу
- **сортировка** этих элементов обычным методом вставок

Более подробно, на первом этапе **группируются** элементы входного набора с **достаточно большим шагом**. Например, выбираются все 1000-е элементы, т.е. создаются группы:

группа 1: 1, 1001, 2001, 3001 и т.д.

группа 2: 2, 1002, 2002, 3002 и т.д.

группа 3: 3, 1003, 2003, 3003 и т.д.

.....

группа 1000: 1000, 2000, 3000 и т.д.

Внутри каждой группы выполняется **обычная** сортировка вставками, что эффективно за счет **небольшого** числа элементов в группе.

На втором этапе выполняется группировка уже с **меньшим** шагом, например - все сотые элементы. В каждой группе опять выполняется обычная сортировка вставками, которая эффективна за счет того, что после первого этапа в каждой группе набор данных будет уже **частично отсортирован**.

На третьем этапе элементы группируются с еще меньшим шагом, например – все десятые элементы. Выполняется сортировка, группировка с еще меньшим шагом и т.д.

На последнем этапе сначала выполняется группировка с **шагом 1**, создающая единственный набор данных размерности n , а затем - сортировка **практически отсортированного** набора.

Пример программной реализации сортировки методом Шелла представлен в листинге 10.4.

Листинг 10.4 – Сортировка методом Шелла

1	static int[] ShellSort(int[] array)
2	{
3	//расстояние между элементами, которые сравниваются
4	var d = array.Length / 2;
5	while (d >= 1)
6	{
7	for (var i = d; i < array.Length; i++)
8	{
9	var j = i;
10	while ((j >= d) && (array[j - d] > array[j]))
11	{
12	Swap(ref array[j], ref array[j - d]);
13	j = j - d;
14	}
15	}
16	d = d / 2;
17	}
18	return array;
19	}

Здесь n – количество элементов в массиве, а $mass[]$ – упорядочиваемый массив элементов.

5. Метод быстрой сортировки

Данный метод в настоящее время считается наиболее быстрым универсальным методом сортировки. Как ни странно, он является обобщением самого плохого из простейших методов – обменного метода. Эффективность метода достигается тем, что перестановка применяется не для соседних элементов, а **отстоящих друг от друга на приличном расстоянии**.

Более конкретно, алгоритм быстрой сортировки заключается в следующем.

- пусть каким-то образом в исходном наборе выделен некий элемент **x**, который принято называть **опорным**. В простейшем случае в качестве опорного можно взять **серединный** элемент массива
- просматривается часть массива, расположенная **левее** опорного элемента и находится первый по порядку элемент $a_i > x$
- после этого просматривается часть массива, расположенная **правее** опорного элемента, причем - **в обратном порядке**, и находится первый по порядку (с конца) элемент $a_j < x$
- производится **перестановка** элементов a_i и a_j
- после этого в левой части, начиная с a_i отыскивается еще один элемент, больший **x**, а в правой части, начиная с a_j отыскивается элемент, меньший **x**
- эти два элемента меняются местами
- эти действия (поиск слева и справа с последующим обменом) продолжаются до тех пор, пока не будет достигнут опорный элемент **x**
- после этого слева от опорного элемента **x** будут находиться элементы, **меньшие опорного**, а справа – элементы, **большие опорного**. При этом обе половины скорее всего не будут отсортированными
- после этого массив разбивается на **правую** и **левую** части, и каждая часть обрабатывается **отдельно** по той же самой схеме: определение опорного элемента, поиск слева и справа соответствующих элементов и их перестановка и т.д.

Пример программной реализации метода быстрой сортировки представлен в листинге 10.5.

Листинг 10.5 – Метод быстрой сортировки

1	void QuickSort(int[] array, int a, int b)
2	{
3	int i = a;
4	int j = b;
5	int middle = array[(a + b)/2];
6	while(i <= j)
7	{
8	while(array[i] < middle)
9	{
10	i++;
11	}
12	while(array[j] > middle)
13	{
14	j--;
15	}
16	if(i <= j)
17	{
18	int temporaryVariable = array[i];
19	array[i] = array[j];
20	array[j] = temporaryVariable;
21	i++;
22	j--;
23	}
24	}
25	if (a < j)
26	{
27	QuickSort(array, a, j);
28	}
29	if (i < b)
30	{
31	QuickSort(array, i, b);
32	}
33	}

6. Метод поразрядной сортировки

Пусть известно, что каждый ключ является **k-разрядным целым числом**. Например, если $k = 4$, то все ключи находятся в диапазоне 0000 – 9999. Смысл поразрядной сортировки заключается в том, что k раз

повторяется карманная сортировка. На первом шаге все ключи группируются по **младшей цифре** (разряд единиц). Для этого в каждом ключе выделяется младшая цифра и элемент помещается в соответствующий список-карман для данной цифры. Потом все списки **объединяются** и создается новый массив, в котором элементы упорядочены по младшей цифре ключа. К этому массиву опять применяется карманная сортировка, но уже по **более старшей** цифре (разряд десятков): в каждом ключе выделяется вторая справа цифра и элементы распределяются по соответствующим спискам. Потом списки объединяются в массив, где элементы будут упорядочены уже **по двум младшим** цифрам. Процесс распределения по все более старшим цифрам с последующим объединением повторяется до старшей цифры (разряд k).

Поскольку цифр всего 10, то для реализации метода необходим вспомогательный массив из 10 ячеек для хранения адресов соответствующих списков.

Для программной реализации можно объявить десятиэлементный массив и ссылочный тип для организации списков. Удобно вместе с началом каждого списка хранить указатель на его последний элемент, что упрощает как добавление новых элементов в конец списка, так и объединение отдельных списков. Внешний цикл алгоритма повторяется k раз (разрядность ключа). Каждый раз внутри этого цикла необходимо:

- обнулить все 20 указателей;
- циклом по числу элементов (m) распределить элементы по своим спискам, выделяя в ключе необходимую цифру;
- циклом по 10 объединить списки в новый набор данных.

В качестве примера приведем реализацию функции поразрядной сортировки для целых чисел любой длины. Однако максимальная длина числа должна быть известна заранее. Она передается функции сортировки в качестве аргумента.

Пример программной реализации поразрядной сортировки представлен в листинге 10.6.

Листинг 10.6 – Поразрядная сортировка

В вышеописанном методе:

1	<code>public static void sorting(int[] arr, int range, int</code>
2	<code>length)</code>
3	<code>{</code>
4	<code>ArrayList[] lists = new ArrayList[range];</code>
5	<code>for(int i = 0; i < range; ++i)</code>
6	<code>lists[i] = new ArrayList();</code>
7	<code>for(int step = 0; step < length; ++step) {</code>
8	<code>//распределение по спискам</code>
9	<code>for(int i = 0; i < arr.Length; ++i) {</code>
10	<code>int temp = (arr[i] % (int)Math.Pow(range, step + 1))</code>
11	<code>/ (int)Math.Pow(range, step);</code>
12	<code>lists[temp].Add(arr[i]);</code>
13	<code>}</code>
14	<code>//сборка</code>
15	<code>int k = 0;</code>
16	<code>for(int i = 0; i < range; ++i) {</code>
17	<code>for(int j = 0; j < lists[i].Count; ++j) {</code>
18	<code>arr[k++] = (int)lists[i][j];</code>
19	<code>}</code>
20	<code>}</code>
21	<code>for(int i = 0; i < range; ++i)</code>
22	<code>lists[i].Clear();</code>
	<code>}</code>
	<code>}</code>

- `length` - максимальное количество разрядов в сортируемых величинах (например, при сортировке слов необходимо знать максимальное количество букв в слове),

- `range` - количество возможных значений одного разряда (при сортировке слов - количество букв в алфавите).

Количество проходов равно числу `length`.

7. Паттерн проектирования «Стратегия»

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

Стратегия — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Формально паттерн Стратегия можно выразить следующей схемой UML:

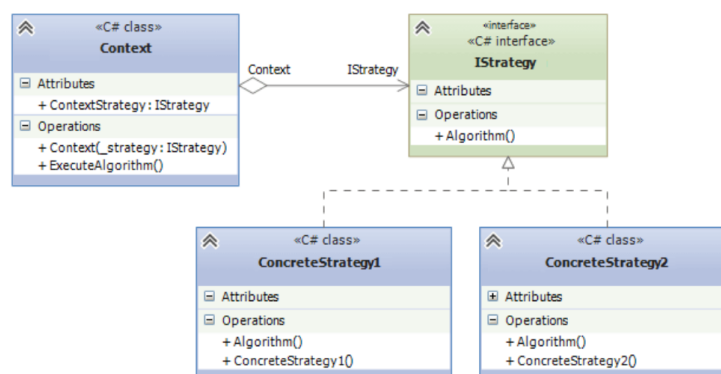


Рисунок 10.1 – Диаграмма классов паттерна «Стратегия»

Формальное определение паттерна на языке C# может выглядеть следующим образом:

Листинг 10.7 – Формальное определение паттерна «Стратегия»

1	public interface IStrategy
2	{
3	void Algorithm();
4	}
5	public class ConcreteStrategy1 : IStrategy
6	{
7	public void Algorithm()
8	{}
9	}
10	public class ConcreteStrategy2 : IStrategy
11	{
12	public void Algorithm()
13	{}
14	}
15	public class Context
16	{
17	public IStrategy ContextStrategy { get; set; }
18	
19	public Context(IStrategy _strategy)
20	{
21	ContextStrategy = _strategy;
22	}
23	public void ExecuteAlgorithm()
24	{
25	ContextStrategy.Algorithm();
26	}
27	}

Как видно из диаграммы, здесь есть следующие участники:

- Интерфейс **IStrategy**, который определяет метод **Algorithm()**. Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
- Классы **ConcreteStrategy1** и **ConcreteStrategy**, которые реализуют интерфейс **IStrategy**, предоставляя свою версию метода **Algorithm()**. Подобных классов-реализаций может быть множество.
- Класс **Context** хранит ссылку на объект **IStrategy** и связан с интерфейсом **IStrategy** отношением агрегации.

В данном случае объект `IStrategy` заключена в свойстве `ContextStrategy`, хотя также для нее можно было бы определить приватную переменную, а для динамической установки использовать специальный метод.

Для реализации методов сортировки для одного набора данных целесообразно будет использовать **паттерн проектирования «Стратегия»**.

Интерфейс `IStrategy` будет определять метод

```
int[] Algorithm(int[] mas, bool flag = true),
```

реализующий собственный метод сортировки. Метод будет принимать два параметра:

- `int[] mas` – целочисленный массив, представляющий собой исходный набор данных для сортировки;
- `flag` – необязательная логическая переменная-индикатор для переключения режимов работы.

Программная реализация интерфейса `IStrategy` представлена в листинге 10.8.

Листинг 10.8

1	<code>public interface IStrategy</code>
2	<code>{</code>
3	<code>int[] Algorithm(int[] mas, bool flag = true);</code>
4	<code>}</code>

Далее определим класс `Context`, который помимо ссылки на конкретный метод сортировки будет хранить исходный набор данных для сортировки. Программная реализация класса `Context` представлена в листинге 10.9.

Листинг 10.8

1	<code>public class Context</code>
2	<code>{</code>
3	<code>public IStrategy ContextStrategy;</code>
4	<code>public static int[] array;</code>
5	<code>public Context(IStrategy Strategy)</code>
6	<code>{</code>
7	<code>this.ContextStrategy = Strategy;</code>
8	<code>}</code>

9	public Context() {}
10	public void ExecuteAlgorithm(bool flag = true)
11	{
12	ContextStrategy.Algorithm(array, flag);
13	}
14	}

Класс хранит два свойства – ссылку на объект `IStrategy` (на конкретный метод сортировки) в строке 3 и целочисленный массив `array`, представляющий собой статическое свойство в строке 4. Определение метода сортировки для заданного контекста данных происходит в конструкторе инициализации класса (строка 5 – 8). За вызов метода сортировки для массива `array` отвечает метод `ExecuteAlgorithm` (строка 10 – 13).

В качестве классов `ConcreteStrategy`, то есть классов, реализующих интерфейс `IStrategy`, будут выступать классы, инкапсулирующие в себе поведение методов сортировки – метода обмена и метода Шелла.

Программная реализация метода обмена представлена в листинге 10.9.

Листинг 10.9

1	public class BubbleSort : IStrategy
2	{
3	public int iterationCount = 0;
4	public static Form1 form1;
5	public int[] Algorithm(int[] mas, bool flag = true)
6	{
7	if (flag)
8	{
9	IOFile.FillContent();
10	System.Diagnostics.Stopwatch myStopwatch = new System.Diagnostics.Stopwatch();
11	myStopwatch.Start();
12	int temp;
13	for (int i = 0; i < mas.Length; i++)
14	{
15	for (int j = i + 1; j < mas.Length; j++)
16	{
17	this.iterationCount++;
18	IOFile.content += this.iterationCount.ToString() + " итерация: " + '\n';
19	IOFile.InputInfoAboutComparison(mas[i], mas[j]);

20	ComparativeAnalysis.Comparison++;
21	if (mas[i] > mas[j])
22	{
23	IOFile.InputInfoAboutTransposition(mas[i], mas[j]);
24	temp = mas[i];
25	mas[i] = mas[j];
26	mas[j] = temp;
27	ComparativeAnalysis.NumberOfPermutations++;
28	IOFile.FillContent();
29	form1.AddItemListBox(mas[i], mas[j]);
30	}
31	}
32	}
33	myStopwatch.Stop();
34	var resultTime = myStopwatch.Elapsed;
35	string elapsedTime =
36	String.Format("{0:00}:{1:00}:{2:00}.{3:000}",
37	resultTime.Hours,
38	resultTime.Minutes,
39	resultTime.Seconds,
40	resultTime.Milliseconds);
41	form1.labelCountComparison.Text =
42	Convert.ToString(ComparativeAnalysis.Comparison);
43	form1.labelNumberOfPermutations.Text =
44	Convert.ToString(ComparativeAnalysis.NumberOfPermutat
45	ions);
46	form1.labelTimeSort.Text = elapsedTime;
	return mas;
	}
	}
	}

Объект `myStopwatch` класса `System.Diagnostics.Stopwatch` (строка 10) позволяет с помощью методов `myStopwatch.Start()` (строка 11) и `myStopwatch.Stop()` (строка 33) отследить время выполнения сортировки. Измеренное время выполнения блока кода сохраняется в свойство `Elapsed` объекта `myStopwatch` (строка 34), которое в дальнейшем можно преобразовать в строковый формат и сохранить в переменную типа `string` (строка 35), обращаясь отдельно к свойствам, отвечающим за хранение часов, минут, секунд и миллисекунд в измеренном промежутке времени (строка 35 –

39). В конечном счете время в формате «чч:мм:сс:мсмс» выводится в лейбл главной формы, где будет происходить визуализация пошаговой работы метода сортировки.

Статический метод `FillContent()` класса `IOFile` позволяет сохранить в буфер элементы массива для дальнейшего вывода в файл. Программная реализация метода представлена в листинге 10.10.

Листинг 10.10

1	<code>public static void FillContent()</code>
2	<code>{</code>
3	<code>foreach (var i in Context.array)</code>
4	<code>{</code>
5	<code>content += Convert.ToString(i) + ' ';</code>
6	<code>}</code>
7	<code>content += '\n';</code>
8	<code>}</code>

В вышеприведенном методе происходит перебор всех элементов массива из контекста данных проектируемого паттерна и добавление элементов в строковую переменную `content`, являющуюся буфером для дальнейшего вывода в файл.

Переменная `iterationCount` служит для хранения количество итераций при выполнении сортировки, статическое свойство `Comparison` класса `ComparativeAnalysis` – для хранения количества сравнений, статическое свойство `NumberOfPermutations` класса `ComparativeAnalysis` – для хранения количества перестановок.

Статические методы `InputInfoAboutComparison` и `InputInfoAboutTransposition` класса `IOFile` выводят информацию о произведенных операциях сортировки и перестановки. Их реализация представлена в листинге 10.11.

Листинг 10.11

1	<code>public static string InputInfoAboutComparison(int first, int second)</code>
2	<code>{</code>

3	<code>content += "Сравниваем " + Convert.ToString(first) + " и " + Convert.ToString(second) + '\n';</code>
4	<code>return "Сравниваем " + Convert.ToString(first) + " и " + Convert.ToString(second) + '\n';</code>
5	<code>}</code>
6	<code>public static string InputInfoAboutTransposition(int first, int second)</code>
7	<code>{</code>
8	<code>content += "Перестановка " + Convert.ToString(first) + " и " + Convert.ToString(second) + '\n';</code>
9	<code>return "Перестановка " + Convert.ToString(first) + " и " + Convert.ToString(second) + '\n';</code>
10	<code>}</code>

В строке 29 вызывается метод `AddItemsListBox(mas[i], mas[j])` для отображения на главной форме приложения пошагового процесса сортировки данных. В качестве параметров метода передаются элементы массива, которые участвуют на данной итерации в перестановке. Программная реализация метода представлена в листинге 10.12.

Листинг 10.12

1	<code>public void AddItemsListBox(int first = -1, int second = -1)</code>
2	<code>{</code>
3	<code>listBox1.Items.Add("");</code>
4	<code>foreach (var item in Context.array)</code>
5	<code>{</code>
6	<code>if (item == first item == second)</code>
7	<code>{</code>
8	<code>listBox1.Items[count] += '[' + Convert.ToString(item) + ']' + " ";</code>
9	<code>}</code>
10	<code>else</code>
11	<code>{</code>
12	<code>listBox1.Items[count] += Convert.ToString(item) + " ";</code>
13	<code>}</code>
14	<code>}</code>
15	<code>count++;</code>
16	<code>}</code>

В строке 8 вышеописанного метода элементы, участвующие в перестановке, выделяются квадратными скобками при выводе в `listBox`. Если элементы не участвуют в перестановке, то они выводятся последовательно через пробел.

Метод Шелла будет инкапсулироваться в класс по аналогичному сценарию.

Пример вызова метода сортировки для заданного контекста (через переключение `radioButton`) представлен в листинге 10.13.

Листинг 10.13

1	<code>private void buttonSort_Click(object sender, EventArgs</code>
2	<code>e)</code>
3	<code>{</code>
4	<code>if (Context.array != null)</code>
5	<code>{</code>
6	<code>if (radioButtonBubbleSort.Checked == true)</code>
7	<code>{</code>
8	<code>this.context = new Context(new BubbleSort());</code>
9	<code>context.ExecuteAlgorithm();</code>
10	<code>this.AddItemListBox();</code>
11	<code>IOFile.SaveData();</code>
12	<code>buttonSort.Enabled = false;</code>
13	<code>}</code>
14	<code>if (radioButtonShellSort.Checked == true)</code>
15	<code>{</code>
16	<code>this.context = new Context(new ShellSort());</code>
17	<code>context.ExecuteAlgorithm();</code>
18	<code>this.AddItemListBox();</code>
19	<code>IOFile.SaveData();</code>
20	<code>buttonSort.Enabled = false;</code>
21	<code>}</code>
22	<code>IOFile.content = "";</code>
23	<code>}</code>
24	<code>else</code>
25	<code>{</code>
26	<code>MessageBox.Show("Массив пуст, сортировка невозможна");</code>
27	<code>}</code>

Здесь в строке 5 и 13 идет проверка состояния переключателей. Если выбран переключатель `radioButtonBubbleSort`, то будет вызываться метод обмена для сортировки массива, хранимого в качестве свойства класса `Context`. В строке 7 клиентом, путем вызова конструктора и передачи в него объекта конкретной стратегии (метода сортировки), создается объект класса `Context`. Далее для созданного объекта вызывается метод выполнения сортировки (строка 8). Далее, после вывода всех итераций сортировки массива, выводится конечный отсортированный массив (строка 9) и происходит вызов метода `SaveData()` класса `IOFile` для сохранения данных о сортировке в файл (строка 10).

8. Файловый ввод данных

Для отображения диалогового окна, позволяющего пользователю открыть файл, через графический конструктор добавляем в главную форму элемент OpenFileDialog.

При выборе опции «Загрузить файл» в выпадающем меню будет вызываться метод LoadData(), представленный в листинге 10.14.

Листинг 10.14

1	public static void LoadData()
2	{
3	if(form1.openFileDialog1.ShowDialog() == DialogResult.Cancel)
4	return;
5	path = form1.openFileDialog1.FileName;
6	using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))
7	{
8	Separator(sr);
9	sr.Close();
10	}
11	}

Метод ShowDialog() открывает диалоговое окно выбора файла, являющегося источником данных для дальнейшей сортировки.

Метод Separator(sr) отвечает за посимвольную обработку данных из загружаемого файла и сохранения этих данных в массив из класса Context.

Листинг 10.15

1	private static void Separator(StreamReader streamReader)
2	{
3	int CurrentFilePosition = 0;
4	int TempMultiplicationResult = 0;
5	int LoadedArrayElement = 0;
6	int TempDigitCapacity = 0;
7	while (streamReader.Peek() != -1)
8	{
9	if (streamReader.Peek() == 32)
10	{
11	char[] vs = new char[2 * CurrentFilePosition];

12	streamReader.Read(vs, CurrentFilePosition, 1);
13	CurrentFilePosition++;
14	}
15	else if (streamReader.Peek() >= 48 && streamReader.Peek() <= 57)
16	{
17	do
18	{
19	if (streamReader.Peek() == -1)
20	{
21	break;
22	}
23	streamReader.Read(listChar, CurrentFilePosition, 1);
24	int.TryParse(Convert.ToString(listChar[CurrentFilePos ition]), out TempMultiplicationResult);
25	TempMultiplicationResult *= Convert.ToInt32(Math.Pow(10.0, TempDigitCapacity));
26	LoadedArrayElement += TempMultiplicationResult;
27	CurrentFilePosition++;
28	TempDigitCapacity++;
29	}
30	while (streamReader.Peek() != 32);
31	string output = new string(Convert.ToString(LoadedArrayElement).ToCharArr ay().Reverse().ToArray());
32	int.TryParse(output, out LoadedArrayElement);
33	arrayList.Add(Convert.ToString(LoadedArrayElement));
34	TempDigitCapacity = 0;
35	TempMultiplicationResult = 0;
36	LoadedArrayElement = 0;
37	}
38	else
39	{
40	MessageBox.Show("Некорректный формат загружаемого файла.");
41	break;
42	}
43	}
44	Context.array = new int[arrayList.Count];
45	for (int k = 0; k < arrayList.Count; k++)
46	{
47	int.TryParse(arrayList[k], out Context.array[k]);
48	}

49	<code>foreach (int j in Context.array)</code>
50	<code>{</code>
51	<code>content += Convert.ToString(j) + " ";</code>
52	<code>}</code>
53	<code>form1.listBox1.Items.Add(content);</code>
54	<code>form1.listBox1.Items.Add("");</code>
55	<code>}</code>

Метод `Separator` принимает в качестве параметра объект класса `StreamReader`, считывающий символы из потока байтов в определенной кодировке. Класс `StreamReader` позволяет нам легко считывать весь текст или отдельные строки из текстового файла.

Метод `Peek()` класса `StreamReader` возвращает следующий доступный символ в целочисленном представлении, не используя его. Если символов больше нет, то возвращает `-1`.

Весь процесс посимвольной обработки данных из файла инкапсулирован в цикл с предусловием, который прекращает свое выполнение при достижении конца файла (строка 7).

В строке 8 проверяется соответствие следующего символа из потока пробелу, который сопоставляется числовому коду, равному 32 (позиция в таблице ASCII). Пробел служит символом, разделяющим между собой элементы массива, хранимые в файле. Если введен пробел, то происходит считывание и возвращение следующего символа в численном представлении с помощью метода `Read` (строка 12), принимающего в качестве параметра `vs` – временный массив, куда считываются символы, `CurrentFilePosition` – индекс в массиве `vs`, начиная с которого записываются считываемые символы, и `count` – максимальное количество считываемых символов. в случае нашей программы мы на каждом шаге считываем по одному символу, поэтому `count` принимаем равным единице. После этого происходит инкрементирование счетчика `CurrentFilePosition`.

В строке 15 проверяется соответствие следующего символа из потока одной из цифр (0-9). При вхождении в данную ветку метода создается цикл с постусловием, прекращающий выполнение, если следующий символ в

файловом потоке – пробел. Цикл предназначен для обработки многозначных цифр в файле, то есть устраняется ситуация, когда каждая цифра многозначного числа сохраняется в отдельную ячейку сортируемого массива. В строке 23 вызывается метод `Read` для чтения одной цифры из файла и сохранения ее в символьный массив `listChar`. В строке 24 вызывается метод `int.TryParse()`, который преобразует строковое представление элемента символьного массива `listChar` из позиции `CurrentFilePosition` в эквивалентное ему целое число и сохраняет преобразованное значение в переменной `TempMultiplicationResult`, если оно выполнено успешно. После этого переменная умножается на 10 в степени, определяемой с помощью переменной `TempDigitCapacity`, определяющей разрядность числа. Результат умножения складывается со значением, хранимым в переменной `LoadedArrayElement`, которая в дальнейшем будет использоваться для размещения числа в массиве.

В строке 31 создается вспомогательная переменная `output`, куда сохраняется реверсивная строка `LoadedArrayElement`, так как в цикле она сохраняет загружаемое число в обратном порядке. Для вызова метода `Reverse()` предварительно целочисленную переменную необходимо преобразовать в строковой тип, а затем в массив символов с помощью метода `ToCharArray()`.

В строке 32 мы снова преобразуем строку `output` и сохраняем ее в целочисленную переменную `LoadedArrayElement`.

В строке 33 мы вызываем метод добавления нового обработанного числа из загружаемого файла в контейнер `arrayList`.

После обработки последнего символа из файла происходит выход из внешнего цикла и выполнение последующих инструкций:

- Вызов конструктора для массива `array` класса `Context` с размерностью, равной количеству элементов, хранимому в контейнере `arrayList` (строка 44);

- Копирование всех элементов контейнера в `arrayList` массив `Context.array` (строка 45-48);
- Сохранение в строковую переменную `content` элементов массива из загружаемого файла для дальнейшей визуализации в списке (строка 49-54).

9. Файловый вывод данных результатов сортировки

Для отображения диалогового окна, позволяющего пользователю сохранить файл, через графический конструктор добавляем в главную форму элемент `SaveFileDialog`.

Программная реализация метода сохранения представлена в листинге 10.16.

Листинг 10.16

1	<code>public static void SaveData()</code>
2	<code>{</code>
3	<code>if (form1.saveFileDialog1.ShowDialog() == DialogResult.Cancel)</code>
4	<code>return;</code>
5	<code>path = form1.saveFileDialog1.FileName;</code>
6	<code>System.IO.File.WriteAllText(path, IOFile.content);</code>
7	<code>}</code>

Для открытия диалогового окна, позволяющего пользователю выбрать местоположение для сохранения файла, вызывается метод `ShowDialog()`. В переменную `path` сохраняется путь для сохранения файла. Значение пути хранится в свойстве `FileName` объекта `saveFileDialog1`. Метод `WriteAllText` класса `File` позволяет создать новый файл, записать в него указанную строку и затем закрыть файл. `path` – файл, в который осуществляется запись, а `IOFile.content` – строка, которую нужно записать в файл.

В диалоговых окнах можно задать фильтр файлов, благодаря чему в диалоговом окне можно отфильтровать файлы по расширению. Фильтр задается в следующем формате `Название_файлов|*.расширение`. Например, `Текстовые файлы(*.txt)|*.txt`. Можно задать сразу несколько фильтров, для этого они разделяются вертикальной линией `|`. Например, `Bitmap files (*.bmp)|*.bmp|Image files (*.jpg)|*.jpg`. В разрабатываемой программе будет добавлен фильтр для текстовых файлов, как это показано в листинге 10.17.

Листинг 10.17

1	<code>saveFileDialog1.Filter = "Text files(*.txt) *.txt All files(*.*) *.*"</code>
2	<code>openFileDialog1.Filter = "Text files(*.txt) *.txt All files(*.*) *.*"</code>

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Реализовать программу, объединяющую улучшенные и простейшие методы сортировки массивов.

Систему взаимодействия алгоритмов сортировки и сортируемого набора данных представить в виде паттерна «Стратегия».

Каждый метод сортировки инкапсулируется в свой класс, добавляемый в основной проект по мере разработки. Кроме того, необходим вспомогательный метод генерации исходного массива случайных целых чисел с заданным числом элементов и выводом этого массива на экран.

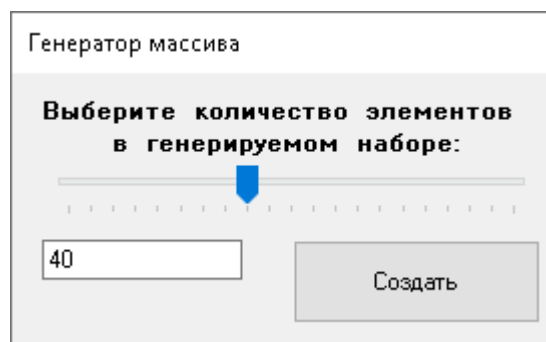


Рисунок 10.2 – Форма «Генератор массива»

С помощью визуальных компонент в панели элементов создать форму для пошаговой демонстрации работы алгоритма сортировки. На каждом шаге сортировки необходимо отметить сравниваемые элементы (например, вставить их в квадратные скобки или отметить цветом). Исходный массив должен обрабатываться методом сортировки, выбранным пользователем, с подсчетом и выводом фактического числа выполненных сравнений и пересылок.

где будет указываться количество сравнений, перестановок и время сортировки по каждому методу с определенным количеством элементов.

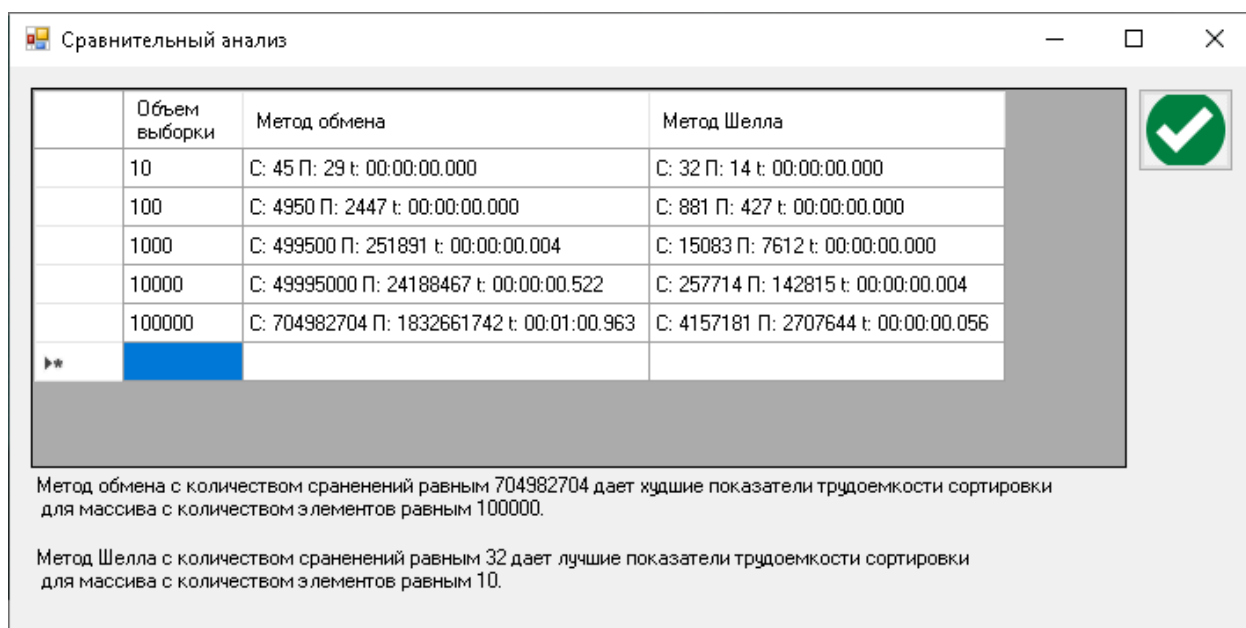


Рисунок 10.5 – Форма «Сравнительный анализ методов сортировки»

Предусмотреть возможность файлового ввода-вывода неотсортированного и сортированного массива. При выводе отсортированного набора данных выводить в файл дополнительно информацию о производимых на каждом шаге сортировки сравнениях и перестановках.

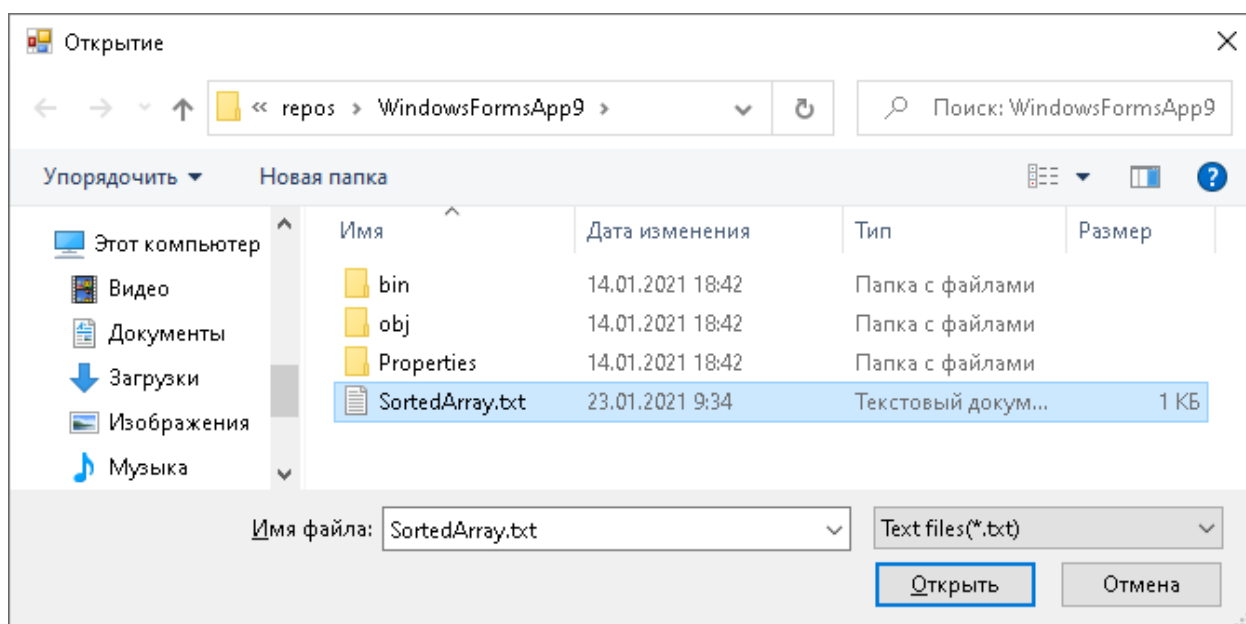


Рисунок 10.6 – Диалоговое окно для открытия файла

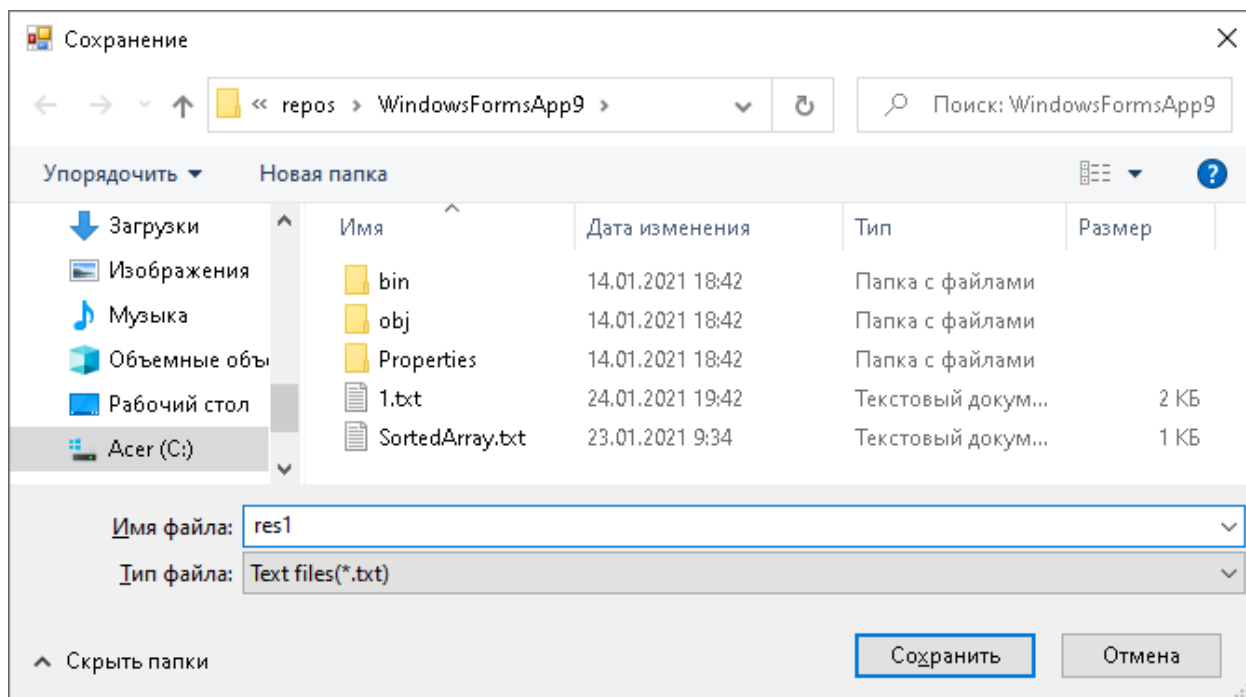
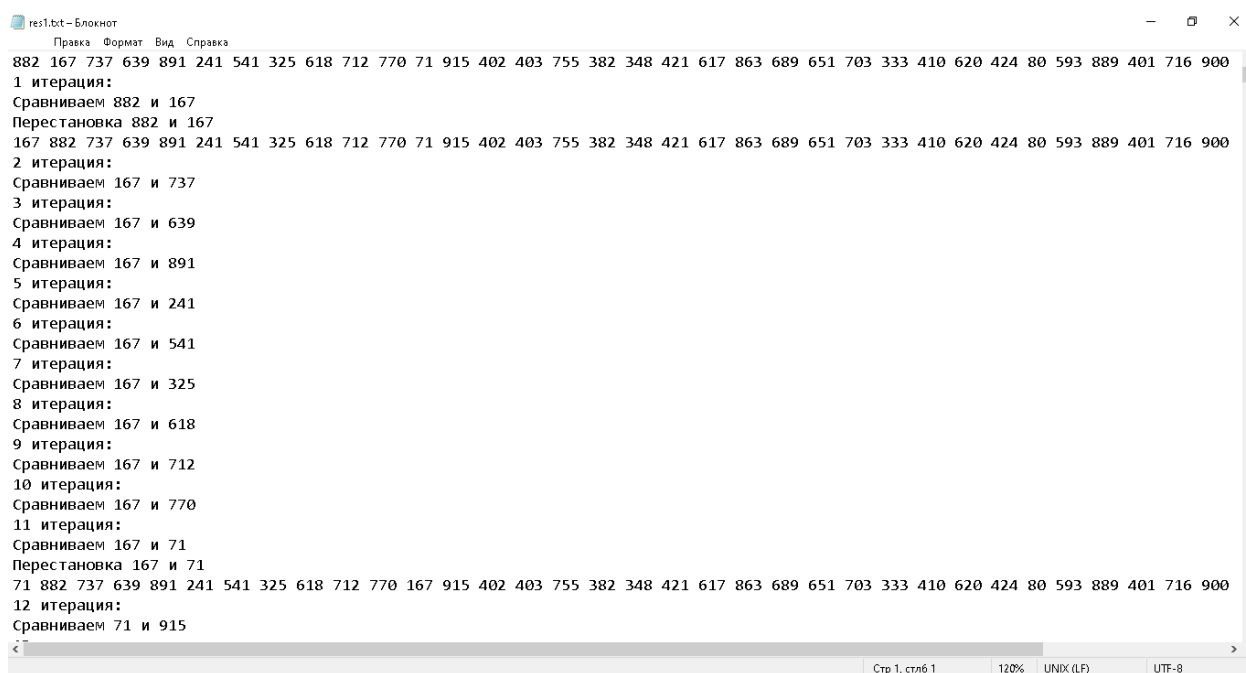


Рисунок 10.7 – Диалоговое окно для сохранения файла



ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ

Вариант	Простейшая сортировка	Улучшенная сортировка	Символ- разделитель при файловой вводе данных	Порядок сортировки

1	Метод обмена	Метод быстрой сортировки	Пробел (‘ ’)	По возрастанию
2	Метод вставок	Метод быстрой сортировки	Пробел (‘ ’)	По возрастанию
3	Метод выбора	Метод быстрой сортировки	Пробел (‘ ’)	По возрастанию
4	Метод обмена	Метод Шелла	Пробел (‘ ’)	По возрастанию
5	Метод вставок	Метод Шелла	Пробел (‘ ’)	По возрастанию
6	Метод выбора	Метод Шелла	Пробел (‘ ’)	По возрастанию
7	Метод обмена	Метод поразрядной сортировки	Пробел (‘ ’)	По возрастанию
8	Метод вставок	Метод поразрядной сортировки	Пробел (‘ ’)	По возрастанию
9	Метод выбора	Метод поразрядной сортировки	Пробел (‘ ’)	По возрастанию
10	Метод обмена	Метод быстрой сортировки	Запятая (‘,’)	По возрастанию

11	Метод вставок	Метод быстрой сортировки	Запятая (‘,’)	По возрастанию
12	Метод выбора	Метод быстрой сортировки	Запятая (‘,’)	По возрастанию
13	Метод обмена	Метод Шелла	Запятая (‘,’)	По возрастанию
14	Метод вставок	Метод Шелла	Запятая (‘,’)	По возрастанию
15	Метод выбора	Метод Шелла	Запятая (‘,’)	По возрастанию
16	Метод обмена	Метод поразрядной сортировки	Запятая (‘,’)	По возрастанию
17	Метод вставок	Метод поразрядной сортировки	Запятая (‘,’)	По возрастанию
18	Метод выбора	Метод поразрядной сортировки	Запятая (‘,’)	По возрастанию
19	Метод обмена	Метод быстрой сортировки	Пробел (‘ ’)	По возрастанию
20	Метод вставок	Метод быстрой сортировки	Пробел (‘ ’)	По возрастанию

21	Метод выбора	Метод быстрой сортировки	Пробел (‘ ’)	По возрастанию
22	Метод обмена	Метод Шелла	Пробел (‘ ’)	По возрастанию
23	Метод вставок	Метод Шелла	Пробел (‘ ’)	По возрастанию
24	Метод выбора	Метод Шелла	Пробел (‘ ’)	По возрастанию
25	Метод обмена	Метод поразрядной сортировки	Пробел (‘ ’)	По возрастанию
26	Метод вставок	Метод поразрядной сортировки	Пробел (‘ ’)	По возрастанию
27	Метод выбора	Метод поразрядной сортировки	Пробел (‘ ’)	По возрастанию
28	Метод обмена	Метод быстрой сортировки	Запятая (‘,’)	По возрастанию
29	Метод вставок	Метод быстрой сортировки	Запятая (‘,’)	По возрастанию
30	Метод выбора	Метод быстрой сортировки	Запятая (‘,’)	По возрастанию
31	Метод обмена	Метод Шелла	Запятая (‘,’)	По возрастанию

32	Метод вставок	Метод Шелла	Запятая (‘,’)	По возрастанию
33	Метод выбора	Метод Шелла	Запятая (‘,’)	По возрастанию
34	Метод обмена	Метод поразрядной сортировки	Запятая (‘,’)	По возрастанию
35	Метод вставок	Метод поразрядной сортировки	Запятая (‘,’)	По возрастанию
36	Метод выбора	Метод поразрядной сортировки	Запятая (‘,’)	По возрастанию