Минобрнауки РФ

ФГБОУ ВО Тверской Государственный Технический Университет

Кафедра  “Программное Обеспечение”

Лабораторная работа N5

По дисциплине “Структуры и алгоритмы обработки данных”

Выполнил: студент группы Б.ПИН.РИС-22.06

Иванов Алексей Михайлович

Проверил: Мальков Александр Анатольевич

г. Тверь, 2023

Оглавление

[**Часть 1. Постановка задачи.** 3](#_Toc145406338)

[**Часть 2. Алгоритм решения.** 3](#_Toc145406339)

[**Часть 3. Оценка временной эффективности в терминах О-функций.** 4](#_Toc145406340)

[**Часть 4. Программный код.** 5](#_Toc145406341)

[**Часть 5. Тестирование и результаты работы.** 9](#_Toc145406342)

# **Часть 1. Постановка задачи.**

Алгоритмы сортировки - по 2 из каждой группы(стратегии).

- реальные данные

- сравнение времени работы сортировок (график)

на одном наборе данных, но разной длины.

# **Часть 2. Алгоритм решения.**

# В данной лабораторной работе будут рассмотрены следующие алгоритмы поиска сортировки в массиве:

1. Сортировка пузырьком
2. Сортировка простыми вставками
3. Сортировка выбором
4. Сортировка Шелла
5. Быстрая сортировка Хоара
6. Пирамидальная сортировка

Сортировка пузырьком – это алгоритм сортировки, который проходит по массиву несколько раз и на каждой итерации сравнивает два соседних элемента. Если первый элемент больше второго, то они меняются местами. Таким образом, на каждой итерации самый большой элемент "всплывает" на правильную позицию, как пузырек в воде. Далее действия повторяются для следующего элемента, и он всплывает до предпоследней позиции, и так далее.

Сортировка простыми вставками - это алгоритм сортировки, который проходит по массиву и на каждой итерации выбирает элемент и вставляет его на правильную позицию в уже отсортированной части массива. На первом проходе первый элемент массива считается отсортированным. Затем берется второй элемент и сравнивается с первым. Если он меньше первого, то они меняются местами. Затем берется третий элемент и сравнивается с предыдущими. Если он меньше предыдущих, то он сдвигается влево до тех пор, пока не будет найдена правильная позиция для него. И так далее, пока массив не будет отсортирован.

Сортировка выбором заключается в том, что каждый раз выбирается наименьший элемент из оставшихся и меняется его местами с текущим элементом. Текущий элемент заключается в цикл, проходящий по всем элементам.

Сортировка Шелла – переработка сортировки вставками. Она сводится к сортировке вставками не сразу всего элемента а каждого n-ого элемента, и последующее уменьшение n до единицы. Это позволяет увеличить скорость «перемещения» слишком выделяющихся элементов по массиву. А в конце сортировка вставками досортировывает массив, она эффективна, когда массив почти упорядочен. В моей реализации, классической, n изначально равно длине массива пополам, и с каждым проходом, n уменьшается вдвое.

Основная идея пирамидальной сортировки заключается в том, что мы сначала создаем кучу из элементов неотсортированного массива, а затем постепенно извлекаем наибольший элемент из кучи и помещаем его в конец отсортированной части массива. После каждого извлечения элемента из кучи, она перестраивается, чтобы сохранить свойство "максимальный элемент находится в корне".

1. Построение кучи:

- Начинаем с последнего элемента массива и движемся в обратном порядке к первому элементу.

- Для каждого элемента проверяем, является ли он максимальным элементом в поддереве, корнем которого он является.

- Если элемент не является максимальным, то он меняется местами с наибольшим из своих потомков.

- После этого переходим к следующему элементу и повторяем процесс до тех пор, пока не дойдем до корня кучи.

2. Сортировка:

- После построения кучи, максимальный элемент будет находиться в корне кучи.

- Меняем местами этот элемент с последним элементом в массиве (который еще не отсортирован).

- Уменьшаем размер кучи на 1 (так как последний элемент уже отсортирован) и перестраиваем кучу.

- Повторяем этот процесс до тех пор, пока размер кучи не станет равен 1.

Алгоритм быстрой сортировки Хоара заключается в следующем: сначала выбирается опорный элемент из массива(классическая реализация это когда он это центр). Далее элементы по обе стороны от опорного сравниваются с опорным и меняются местами так, чтобы с каждой из сторон от опорного элементы одинаково относились к опорному (то бишь, были меньше либо больше). Далее функция вызывается рекурсивно на каждый из полученных подмассивов. Массив, состоящий из одного элемента, считается по умолчанию отсортированым.

# **Часть 3. Оценка временной эффективности в терминах О-функций.**

Составим сводную таблицу эффективности для матрицы в обоих типах представлений.

|  |  |
| --- | --- |
|  | Эффективность (n длина массива) |
| Пузырьковая | O(n^2) |
| Выбором | O(n^2) |
| Вставками | O(n^2) |
| Быстрая | O(n log(n)) |
| Шелла | от O(n log^2(n)) до O(n^2) |
| Пирамидальная | O(n log(n)) |

# **Часть 4. Программный код.**

Итоговая библиотека классов имеет следующий вид:

/// <summary>

/// Класс-обертка для массива, добавляющий

/// методы сортировки для

/// массивов произвольного типа Т,

/// реализующего интерфейс IComparable

/// </summary>

/// <typeparam name="T">Тип данных в сортируемых массивах</typeparam>

public class Sorter<T> : ICloneable, IList<T>,

ICollection<T>, IEnumerable<T>, IStructuralEquatable, IStructuralComparable

where T : IComparable<T>

{

/// <summary>

/// Поле, отображающее массив

/// </summary>

protected T[] arr;

/// <summary>

/// Длина массива

/// </summary>

protected int len;

public int Count => ((ICollection<T>)arr).Count;

public bool IsReadOnly => ((ICollection<T>)arr).IsReadOnly;

/// <summary>

/// Конструктор, создающий объект на основе массива

/// </summary>

/// <param name="arr">Массив</param>

public Sorter(T[] arr)

{

this.arr = arr;

this.len = arr.Length;

}

/// <summary>

/// Конструктор, создающий объект на основе IList

/// </summary>

/// <param name="arr">IList</param>

public Sorter(IList<T> arr)

{

this.arr = arr.ToArray();

this.len = arr.Count;

}

/// <summary>

/// Индексатор, позволяющий работать с элементами массива по индексу

/// </summary>

/// <param name="i">Индекс элемента в обернутом массиве</param>

/// <returns>Элемент массива по данному индексу</returns>

public T this[int i]

{

get { return arr[i]; }

set { arr[i] = value; }

}

/// <summary>

/// Позволяет поменять местами элементы по ссылке

/// </summary>

/// <param name="elem1">Первый элемент</param>

/// <param name="elem2">Второй элемент</param>

private void Swap(ref T elem1, ref T elem2)

{

T temp = elem1;

elem1 = elem2;

elem2 = temp;

}

/// <summary>

/// Позволяет поменять элементы местами по индексу

/// </summary>

/// <param name="index1">Первый индекс</param>

/// <param name="index2">Второй индекс</param>

private void Swap(int index1, int index2)

{

T temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

/// <summary>

/// Осуществляет сортировку выбором

/// </summary>

public void SelectSort()

{

for (int i = 0; i < len; i++)

{

int min = i;

for (int j = i; j < len; j++)

{

if (arr[min].CompareTo(arr[j]) > 0)

min = j;

}

Swap(i, min);

}

}

/// <summary>

/// Осуществляет сортировку пузырьком

/// </summary>

public void BubbleSort()

{

for (int i = 0; i < len; i++)

for (int j = i; j < len; j++)

if (arr[j].CompareTo(arr[i]) < 0)

Swap(ref arr[j], ref arr[i]);

}

/// <summary>

/// Осуществляет сортировку Шелла

/// </summary>

public void ShellSort()

{

int d = len / 2;

while (d >= 1)

{

for (int i = 0; i < len; i+=d)

{

int j = i;

while (j > 0 && arr[j].CompareTo(arr[j - 1]) <= 0)

{

Swap(j - 1, j);

j-=d;

}

}

d /= 2;

}

}

/// <summary>

/// Осуществляет сортировку вставками

/// </summary>

public void InsertSort()

{

for (int i = 0; i < len; i++)

{

int j = i;

while (j > 0 && arr[j].CompareTo(arr[j - 1]) <= 0)

{

Swap(j - 1, j);

j--;

}

}

}

/// <summary>

/// Осуществляет быструю сортировку Хоара

/// </summary>

public void FastSort()

{

FastSortRec(0, len - 1);

}

/// <summary>

/// Вспомогательная рекурсивная функция

/// для быстрой сортировки Хоара

/// </summary>

/// <param name="start">Начало сортируемого отрезка</param>

/// <param name="end">Конец сортируемого отрезка</param>

private void FastSortRec(int start, int end)

{

int i = start, j = end;

T pivot = arr[(start + end) / 2];

while (i <= j)

{

while (arr[i].CompareTo(pivot) < 0)

{

i++;

}

while (arr[j].CompareTo(pivot) > 0)

{

j--;

}

if (i <= j)

{

Swap(i, j);

i++;

j--;

}

}

if (start < j)

{

FastSortRec(start, j);

}

if (i < end)

{

FastSortRec(i, end);

}

}

/// <summary>

/// Осуществляет пирамидальную сортировку

/// </summary>

public void HeapSort()

{

HeapSort(len);

}

/// <summary>

/// Вспомогательная функция для пирамидальной сортировки

/// </summary>

private void HeapSort(int N)

{

for (int i = N / 2 - 1; i >= 0; i--)

{

Heapify(arr, N, i);

}

for (int i = N - 1; i >= 0; i--)

{

Swap(ref arr[0], ref arr[i]);

Heapify(arr, i, 0);

}

return;

}

/// <summary>

/// Вспомогательная функция для пирамидальной сортировки,

/// приводящая массив на отрезке к виду бинарной кучи

/// </summary>

private void Heapify(T[] M, int N, int i)

{

int iM = i;

int L = 2 \* i + 1, R = 2 \* i + 2;

if (L < N && (M[L].CompareTo(M[iM]) > 0))

{

iM = L;

}

if (R < N && (M[R].CompareTo(M[iM]) > 0))

{

iM = R;

}

if (i != iM)

{

Swap(ref M[i], ref M[iM]);

Heapify(M, N, iM);

}

return;

}

/// <summary>

/// Статический метод, создающий случаный целочисленный массив

/// </summary>

/// <param name="start">Начало отрезка случайных чисел</param>

/// <param name="end">Конец отрезка случайных чисел</param>

/// <param name="N">Длина массива</param>

/// <returns>Целочисленный случайный массив</returns>

public static int[] CreateRandom(int start, int end, int N)

{

Random R = new Random();

int[] res = new int[N];

for (int i = 0; i < N; i++)

{

res[i] = R.Next(start, end);

}

return res;

}

/// <summary>

/// Осуществляет перемешку текущего массива

/// </summary>

public void Shuffle()

{

Random R = new Random();

for (int i = 0; i < len; i++)

{

this.Swap(i, R.Next(0, len));

}

}

/// <summary>

/// Проверяет, отсортирован ли данный массив по возрастанию

/// </summary>

/// <param name="arr">Проверяемый массив</param>

/// <returns>True: массив отсортирован по возрастанию

/// False: все иные случаи</returns>

public static bool CheckIfSorted<P>(P[] arr) where P : IComparable<P>

{

for (int i = 0; i < arr.Length - 1; i++)

{

if (arr[i].CompareTo(arr[i + 1]) > 0)

return false;

}

return true;

}

public int IndexOf(T item)

{

return ((IList<T>)arr).IndexOf(item);

}

public void Insert(int index, T item)

{

((IList<T>)arr).Insert(index, item);

}

public void RemoveAt(int index)

{

((IList<T>)arr).RemoveAt(index);

}

public void Add(T item)

{

((ICollection<T>)arr).Add(item);

}

public void Clear()

{

((ICollection<T>)arr).Clear();

}

public bool Contains(T item)

{

return ((ICollection<T>)arr).Contains(item);

}

public void CopyTo(T[] array, int arrayIndex)

{

((ICollection<T>)arr).CopyTo(array, arrayIndex);

}

public bool Remove(T item)

{

return ((ICollection<T>)arr).Remove(item);

}

public IEnumerator<T> GetEnumerator()

{

return ((IEnumerable<T>)arr).GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return arr.GetEnumerator();

}

public bool Equals(object other, IEqualityComparer comparer)

{

return ((IStructuralEquatable)arr).Equals(other, comparer);

}

public int GetHashCode(IEqualityComparer comparer)

{

return ((IStructuralEquatable)arr).GetHashCode(comparer);

}

public int CompareTo(object other, IComparer comparer)

{

return ((IStructuralComparable)arr).CompareTo(other, comparer);

}

public object Clone()

{

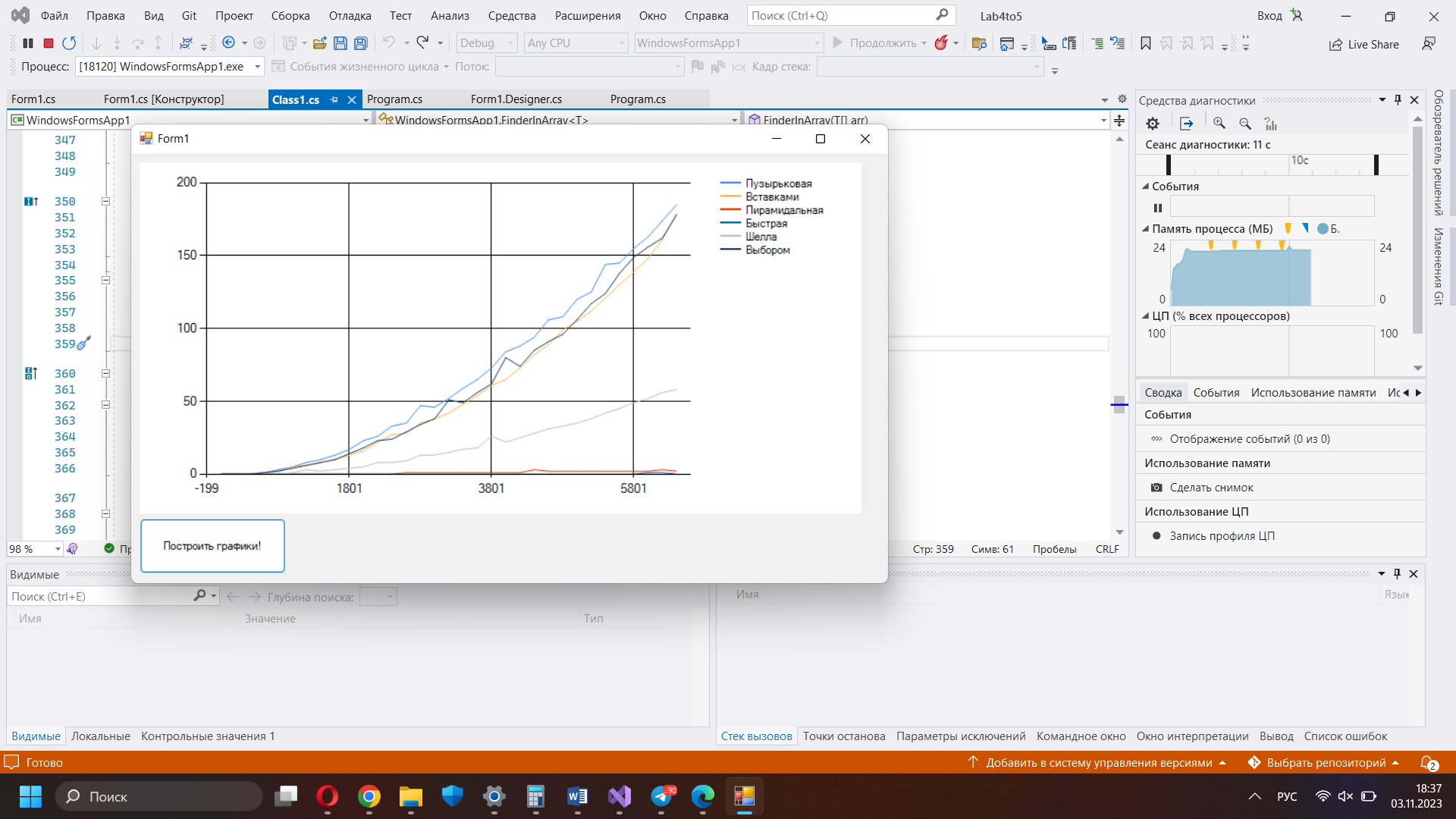
return arr.Clone();

}

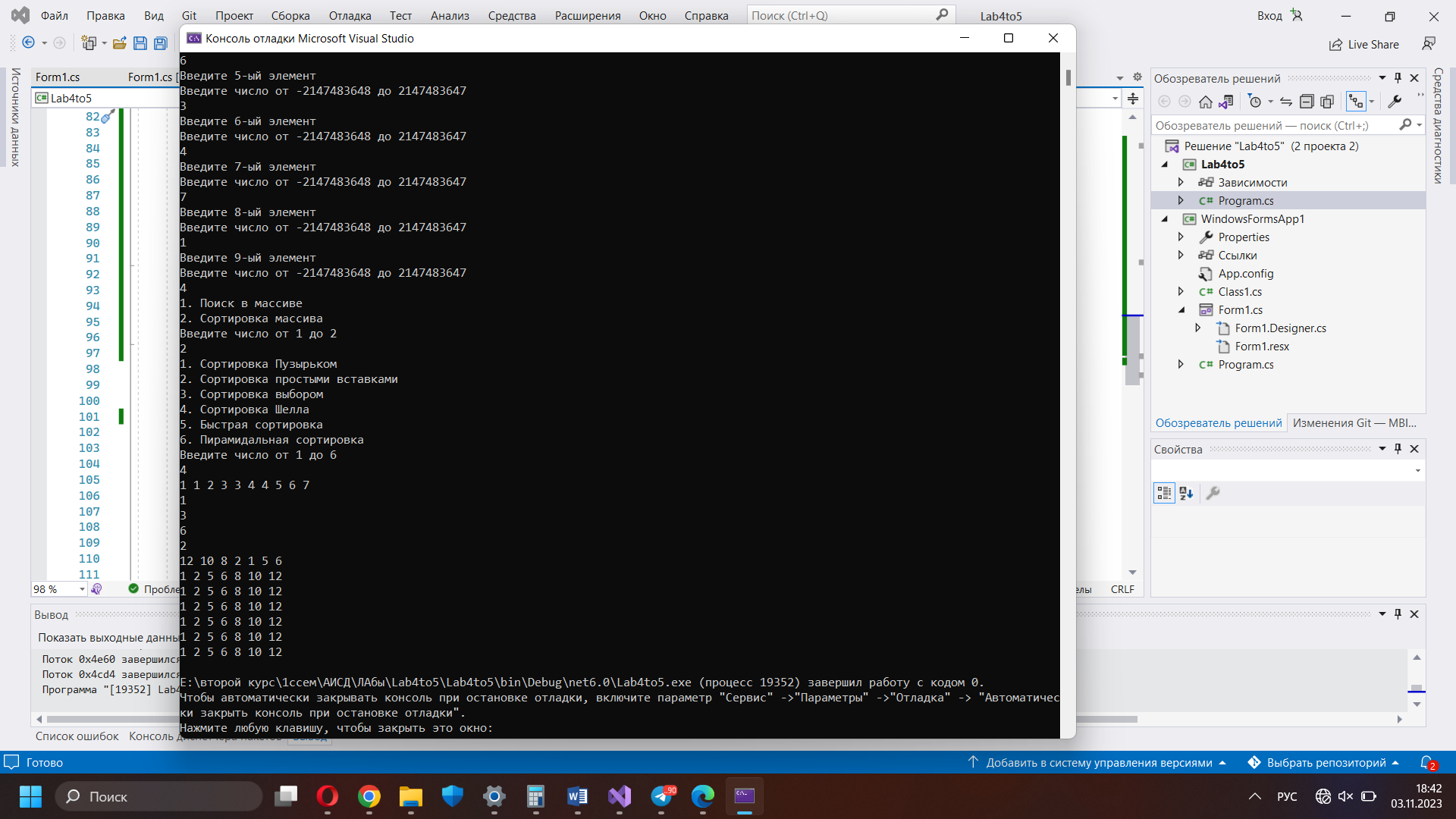
}

**Часть 5. Тестирование и результаты работы.**

Для тестирования было разработано консольное приложение, проверяющее корректность алгоритмов, а также проект WINDOWS FORMS, позволяющий отследить динамику роста времени сортировки на объемах данных. Он подтверждает теортические выкладки



Пример, вызывающий для массива каждую из сортировок.



Консольное приложение, которое позволяет пользователю ввести массив самостоятельно его заполняя.

