`

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

СЕМЕСТРОВАЯ РАБОТА
по дисциплине «Алгоритмы и анализ сложности»
«Экспериментальный анализ различных методов сортировки»

Обучающийся: Шарифуллин Ринас Рамилевич гр. 09–131
(ФИО студента) (Группа)

Руководитель: к.ф.-м.н., доцент КСАИТ, А. В. Васильев

Казань – 2023

Содержание

Введение

Сортировка является одним из основных алгоритмов компьютерной науки, который позволяет упорядочить элементы в некотором наборе данных. На сегодняшний день существует большое количество методов сортировки, каждый из которых имеет свои преимущества и недостатки в зависимости от типа данных и объема сортируемых элементов. В данной работе будут рассмотрены такие методы сортировки, как:

- Сортировка пузырьком;
- Сортировка вставками;
- Сортировка выбором;
- Шейкерная сортировка;
- Сортировка Шелла (с делением на 2, с последовательностью Хиббарда);
- Поразрядная сортировка;
- Сортировка слиянием;
- Сортировка кучей;
- Быстрая сортировка;
- Встроенная сортировка.

Будет проведен анализ сортировок, для которых использовались такие типы данных, как:

- byte;
- int;
- string;
- DateTime.

Методика проведения эксперимента

Для начала мною было принято решение разбить сортировки по видам и сравнивать их внутри каждого:

1. Простые схемы – это сортировки, которые могут быть легко реализованы и понятны. Однако, они не всегда являются наиболее эффективными и быстрыми.
2. Продвинутые схемы — это более сложные алгоритмы сортировки, которые обычно имеют более высокую производительность и эффективность. Эти алгоритмы могут работать с большими объемами данных и обрабатывать их быстрее, чем простые методы сортировки. Однако, их реализация может быть более сложной и требовательной к ресурсам компьютера.
3. Быстрые схемы – это сортировки, которые являются самыми эффективными по времени выполнения.

Сравнения происходили на таких типах данных, как byte, int, string (длина строки – 20 символов), DateTime, с различными длинами массивов. Также для анализа использовались по-разному порождённые массивы: случайно сгенерированные, отсортированные в обратном направлении и с большой долей одинаковых элементов.

Работа проводилась на ноутбуке HONOR MagicBook Pro с процессором AMD Ryzen 5 4600H, с оперативной памятью 16 Гб.

Полученные результаты

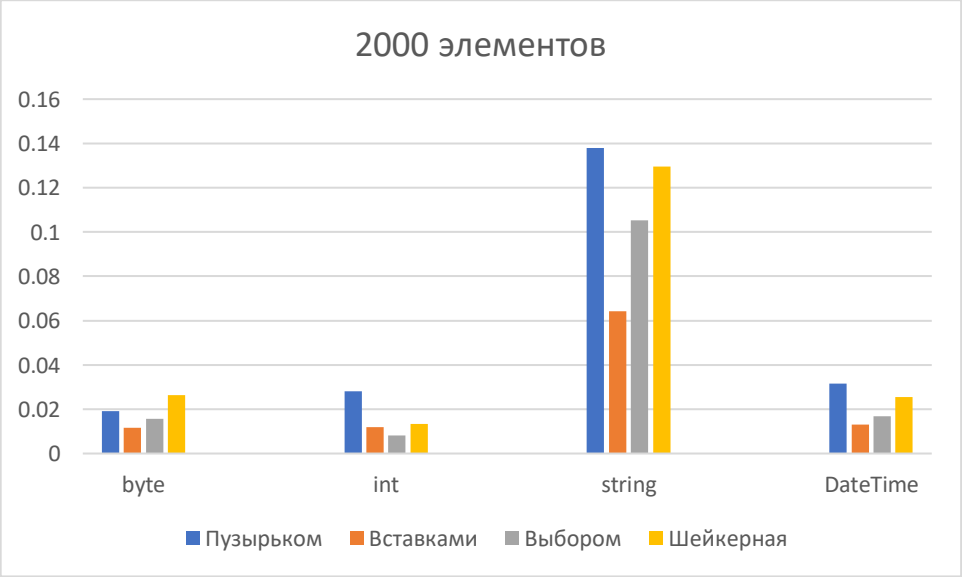
Замечание. Во всех представленных ниже гистограммах шкала слева представлена в секундах!

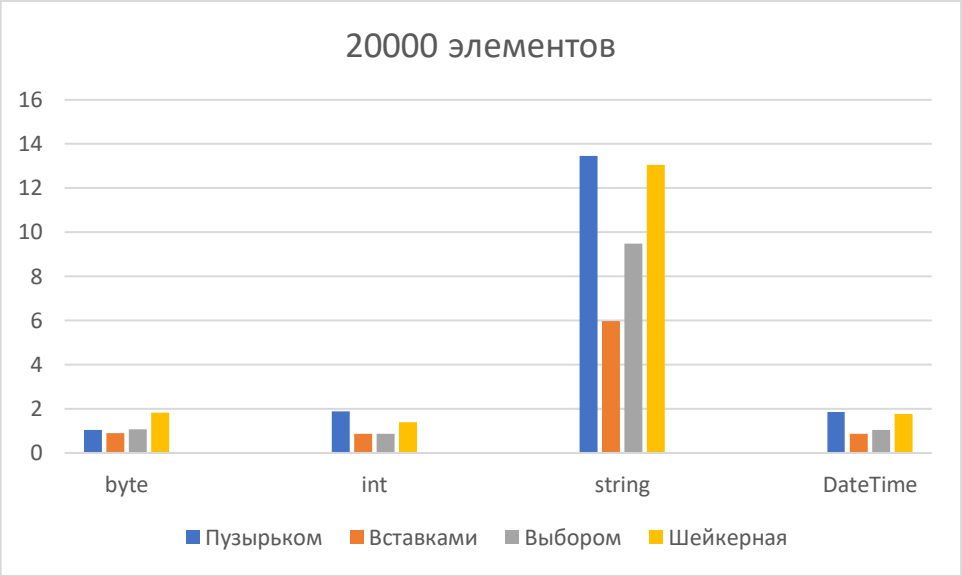
Сравнение сортировок внутри своего вида

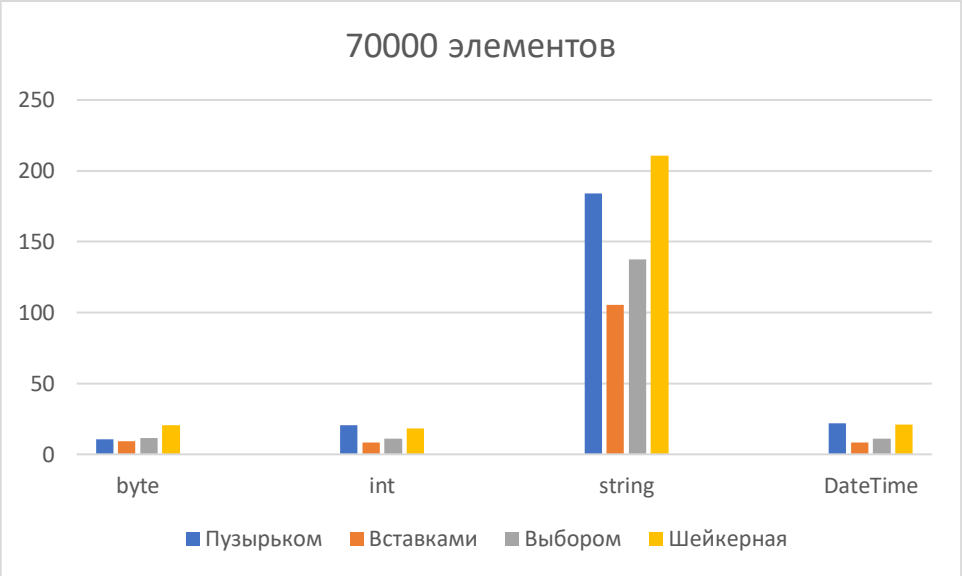
Простые схемы

- Сортировка пузырьком;
- Сортировка вставками;
- Сортировка выбором;
- Сортировка перемешиванием (Шейкерная сортировка).

Случайно сгенерированные массивы





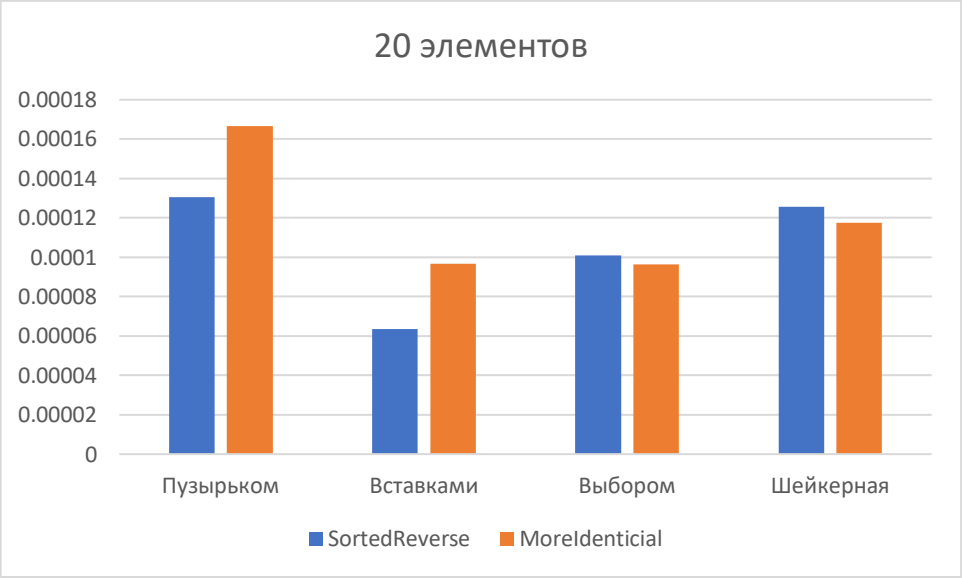


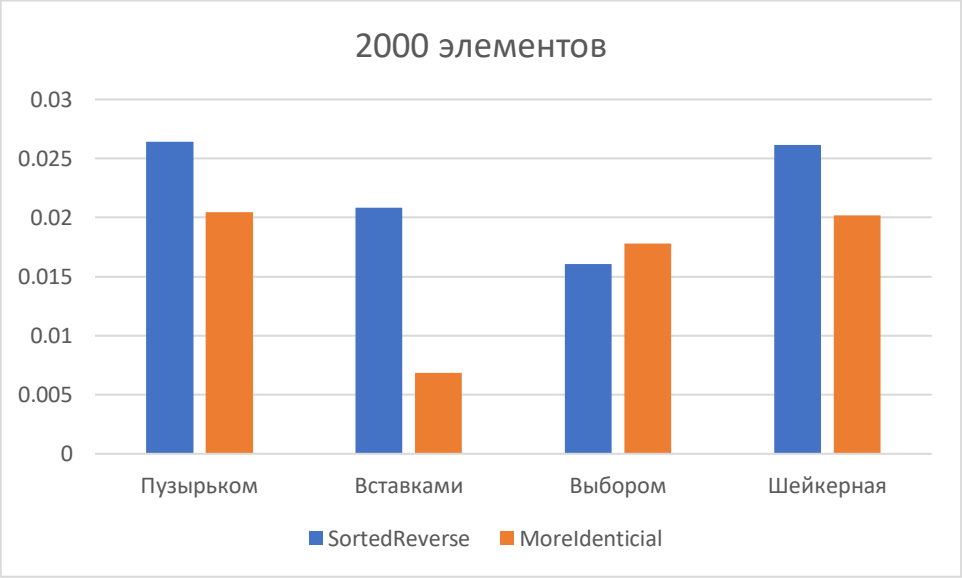
Результаты: как бы это очевидно не было, но с увеличением количества элементов массива страдает время выполнения сортировки. Проанализировав гистограммы, могу сказать, что при малом количестве данных (20 элементов) сортировки показывают почти одинаковые результаты для всех типов, кроме DateTime. С 2000 элементов начинается виднеться разница во времени выполнения алгоритмов. «Пузырьковая» и «Шейкерная» сортировки оказались медленными и неэффективными. Получше остальных себя проявляет сортировка вставками. Особенно это становится хорошо заметно с значительным увеличением количества элементов (20000, 70000). Также можно сделать вывод о том, как максимально неэффективны простые сортировки для типа данных string.
Таким образом, если расположить алгоритмы по быстроте выполнения сортировки массива, получается:

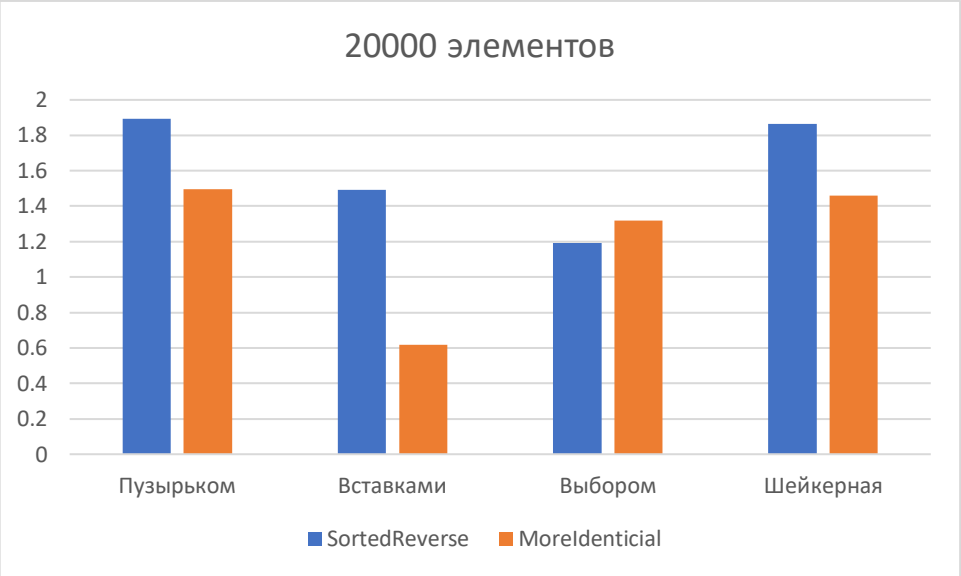
1-ое место: сортировка вставками,
2-ое место: сортировка выбором,
3-ое место: сортировка пузырьком и шейкерная

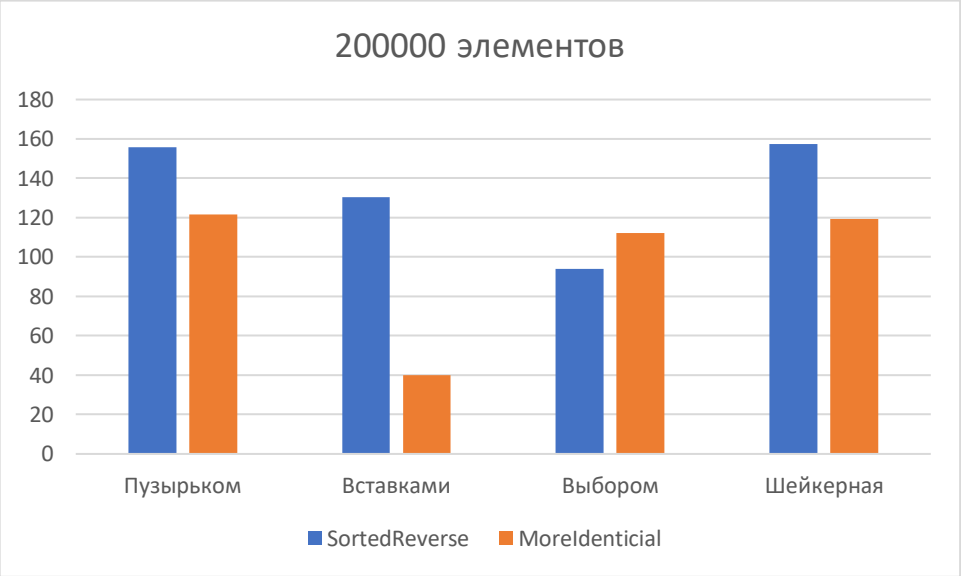
Отсортированные в обратном направлении и с большой долей одинаковых элементов массивы

Замечание. Такого рода порождённые массивы рассматривались только для типа данных int.







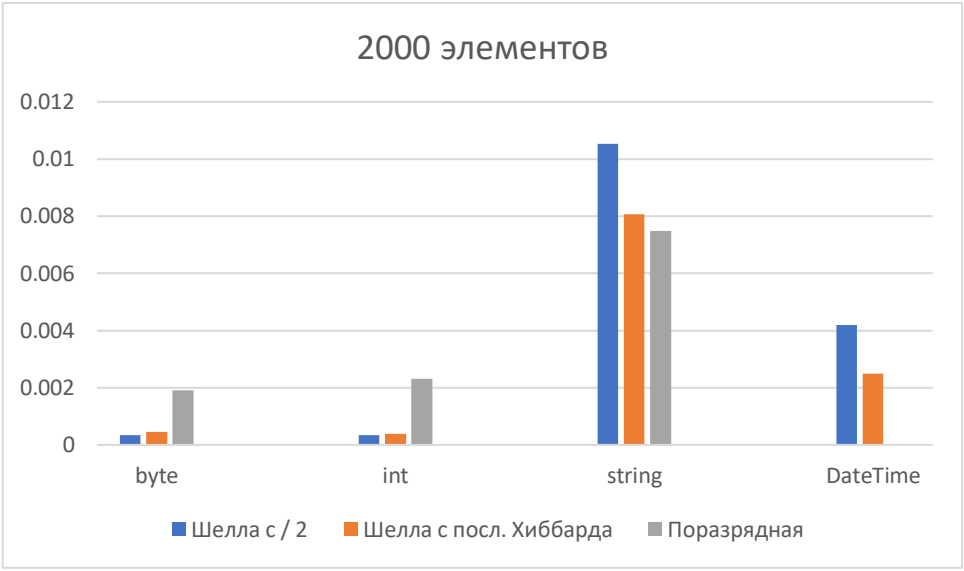


Результаты: при использовании обратно отсортированных массивов результаты сильно не изменились. Аутсайдерами все также остаются «Пузырьковая» и «Шейкерная». Могу только сказать, что сортировка вставками уступила место по времени выполнения сортировке выбором. При большой доле одинаковых элементов результаты оказались идентичными, как и при случайно сгенерированных массивах.

Продвинутые схемы

- Сортировка Шелла с делением на 2;
- Сортировка Шелла с последовательностью Хиббарда;
- Поразрядная сортировка.

Случайно сгенерированные массивы


2000 элементов
0.012
0.01
0.008
0.006
0.004
0.002
0
byte
int
string
DateTime
Шелла с / 2
Шелла с посл. Хиббарда
Поразрядная

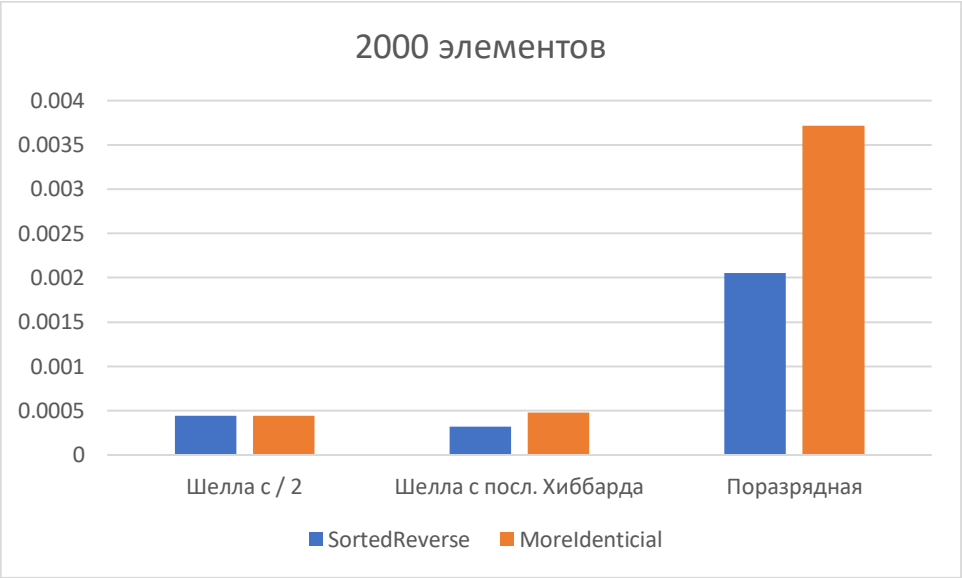

20000 элементов
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
byte
int
string
DateTime
Шелла с / 2
Шелла с посл. Хиббарда
Поразрядная

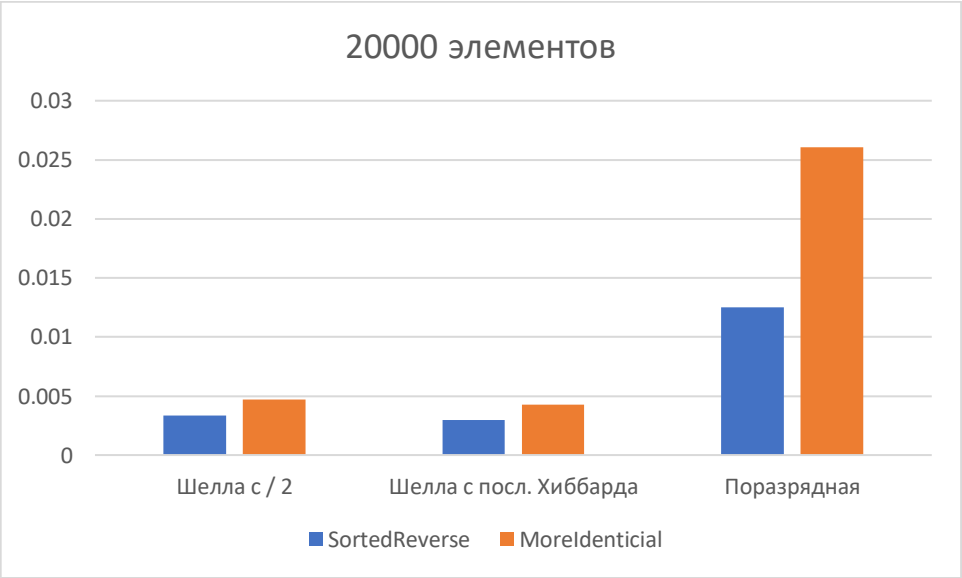

200000 элементов
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
byte
int
string
DateTime
Шелла с / 2
Шелла с посл. Хиббарда
Поразрядная

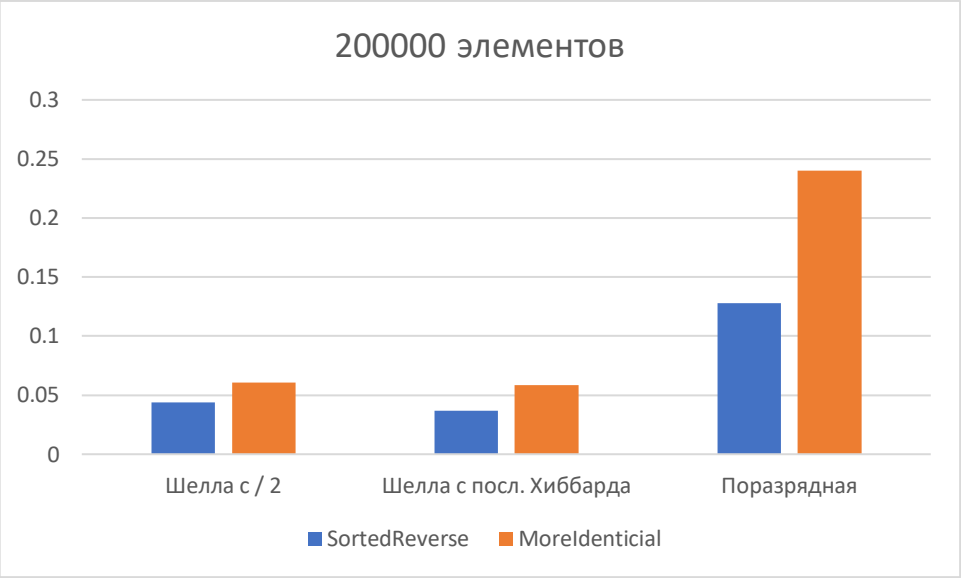
Результаты: по результатам данных можно сказать, что сортировки Шелла (с / 2 и с посл. Хиббарда) для различных типов и разного количества данных оказались примерно одинаковыми. Сортировка Шелла с / 2 получше введет себя при очень большом количестве элементов, а сортировка Шелла с посл. Хиббарда при меньшом. Если сравнивать эти две сортировки с поразрядной, можно убедиться в неэффективности последней при малых размерах массива (<2000000). Однако с увеличением количества элементов (≥ 2000000) картина меняется с точностью наоборот. Поразрядная сортировка становится эффективнее, т.к. это алгоритм с линейной сложностью, то есть время выполнения не зависит от размера выходных данных. Для типа данных string без сомнений фаворитом является поразрядная сортировка.

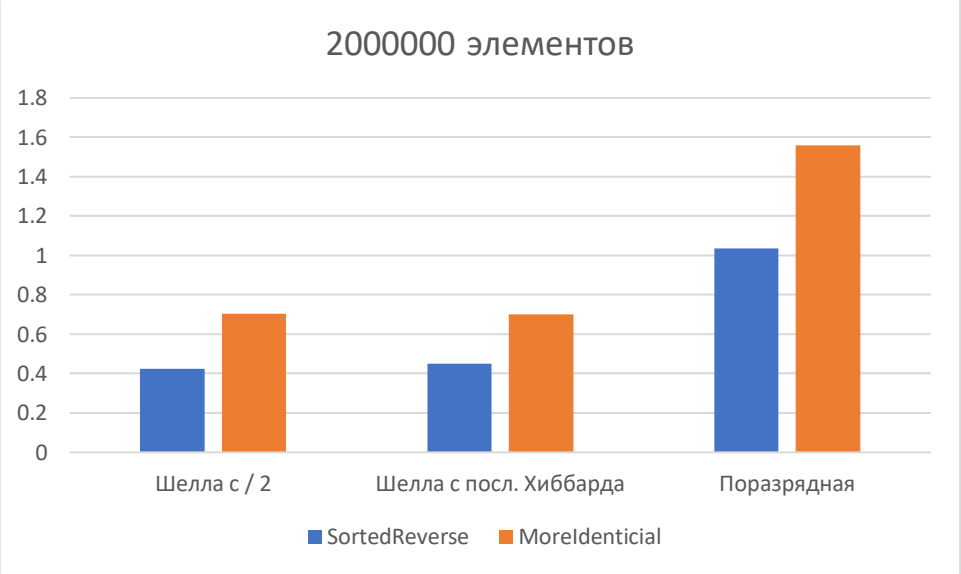
Отсортированные в обратном направлении и с большой долей одинаковых элементов массивы

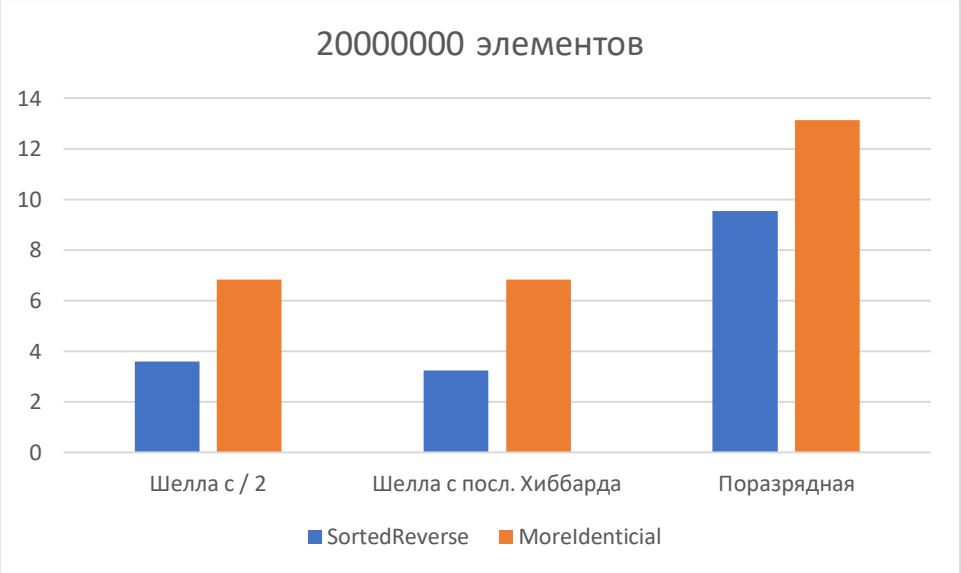
Замечание. Такого рода порождённые массивы рассматривались только для типа данных int.






200000 элементов
0.3
0.25
0.2
0.15
0.1
0.05
0
Шелла с / 2
Шелла с посл. Хиббарда
Поразрядная
SortedReverse
MoreIdenticial


2000000 элементов
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Шелла с / 2
Шелла с посл. Хиббарда
Поразрядная
SortedReverse
MoreIdenticial

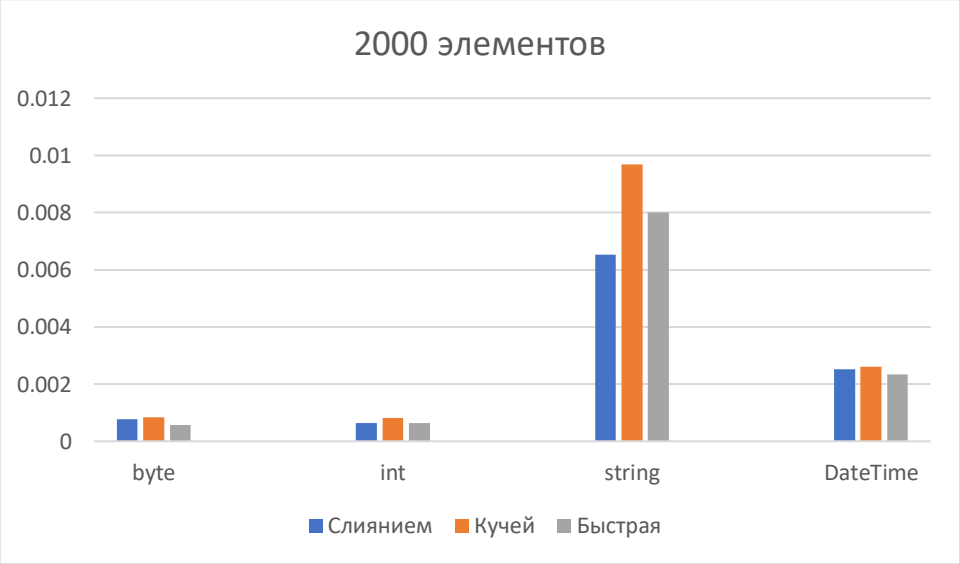

20000000 элементов
14
12
10
8
6
4
2
0
Шелла с / 2
Шелла с посл. Хиббарда
Поразрядная
SortedReverse
MoreIdenticial

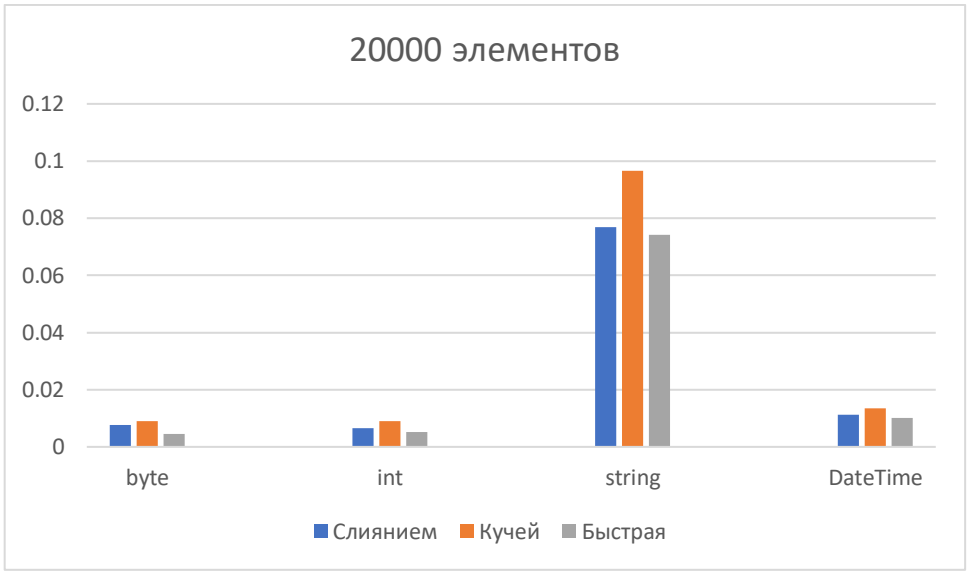
Результаты: при использовании обратно отсортированных и с большой долей одинаковых элементов массивов для сравнения сортировок Шелла и поразрядной результаты оказались немножко другими. Сортировки Шелла с / 2 и посл. Хиббарда показали максимально схожие результаты, а вот поразрядная, при таких порождённых массивах, сильно уступила им даже при большом количестве элементов. Причина этому такая, что сортировка Шелла меняет элементы только в том случае, когда это необходимо, в отличие от поразрядной.
Таким образом, точно определить победителя невозможно.

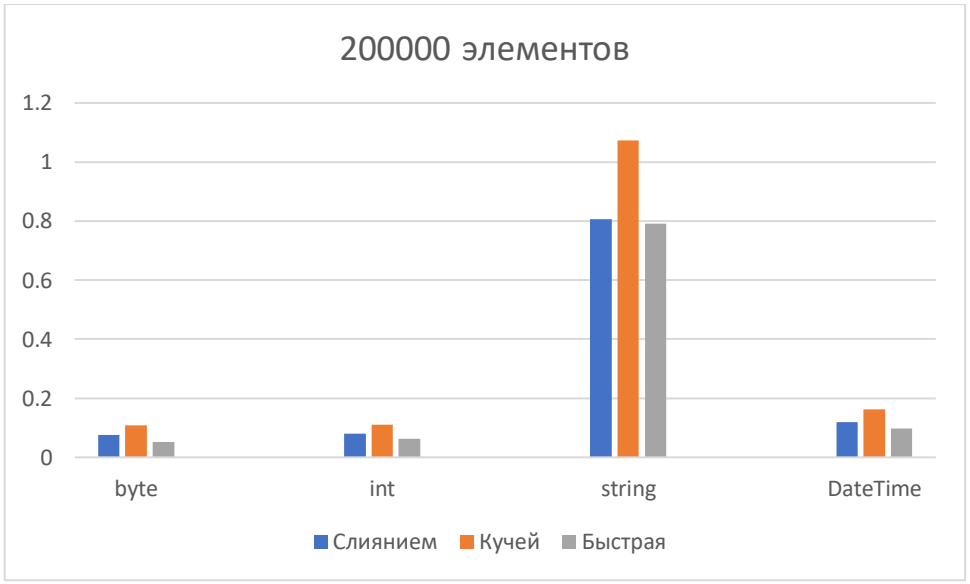
Быстрые схемы

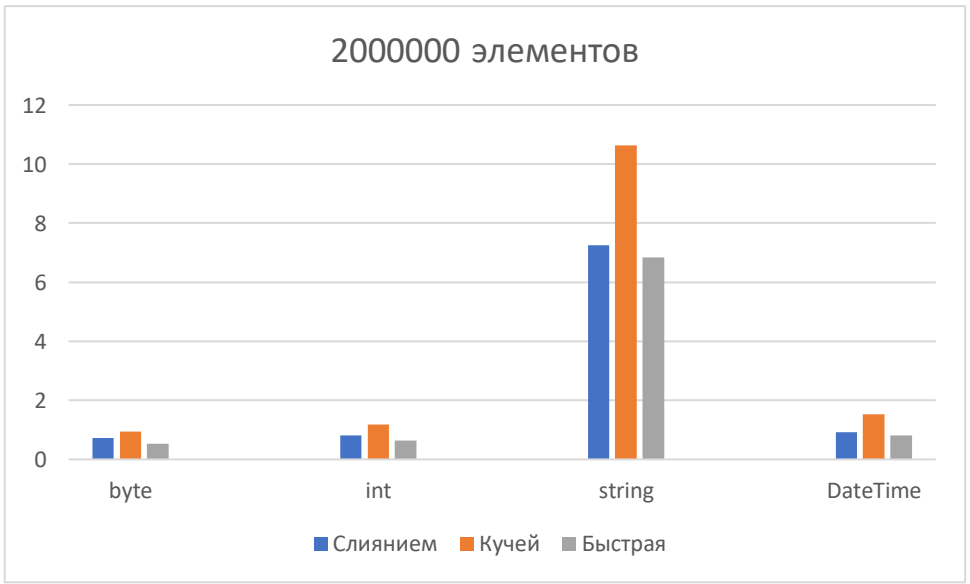
- Сортировка слиянием;
- Сортировка кучей;
- Быстрая сортировка.

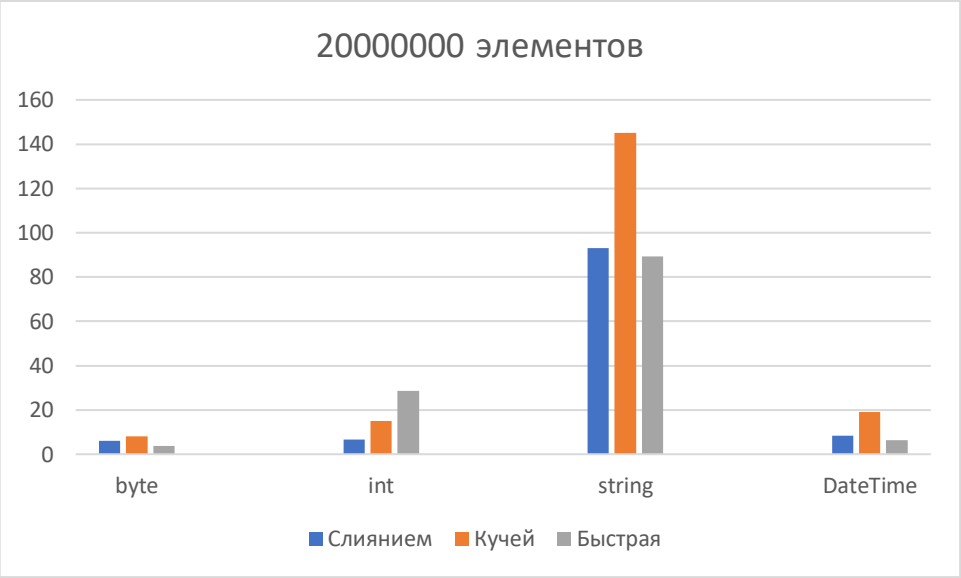
Случайно сгенерированные массивы




20000 элементов
0.12
0.1
0.08
0.06
0.04
0.02
0
byte
int
string
DateTime
Слиянием
Кучей
Быстрая


200000 элементов
1.2
1
0.8
0.6
0.4
0.2
0
byte
int
string
DateTime
Слиянием
Кучей
Быстрая

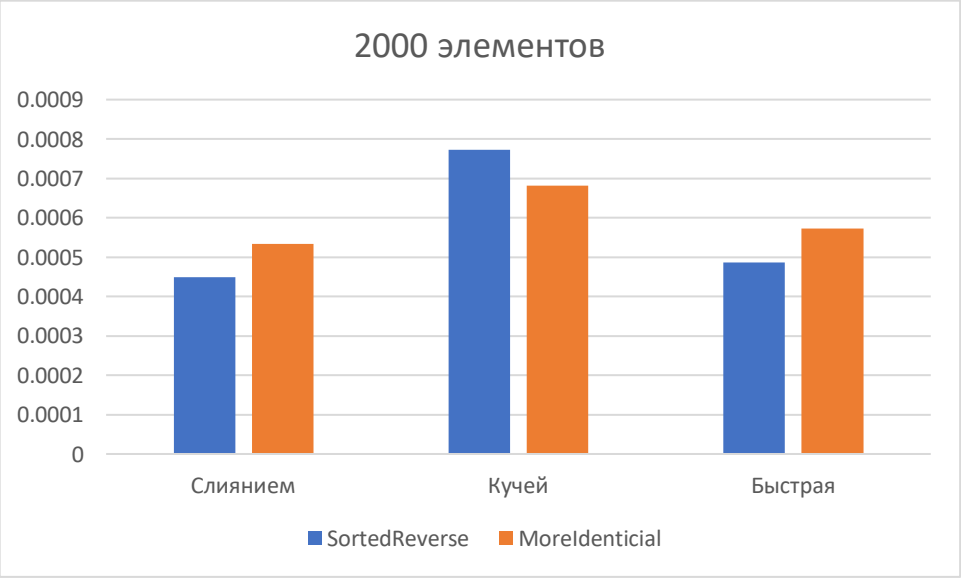

2000000 элементов
12
10
8
6
4
2
0
byte
int
string
DateTime
Слиянием
Кучей
Быстрая

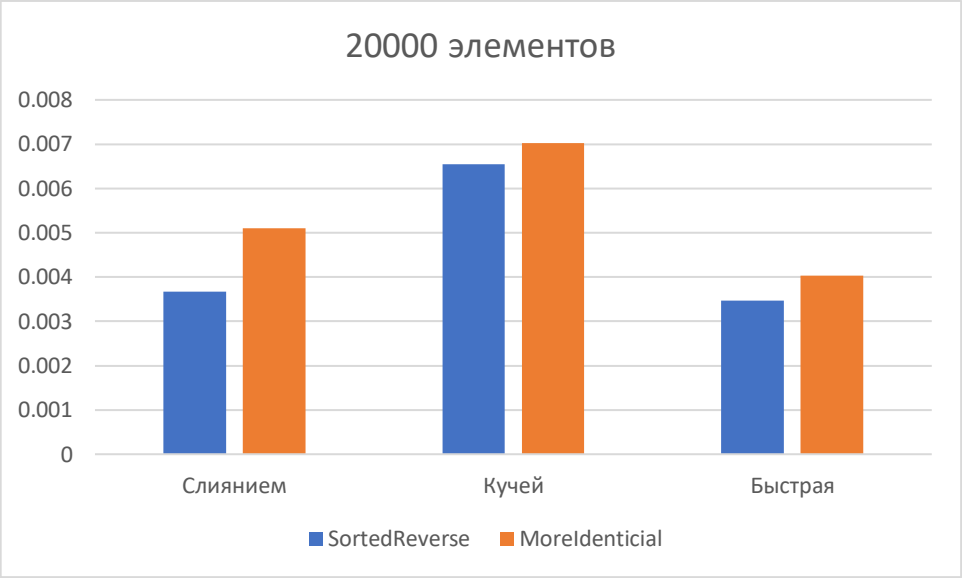


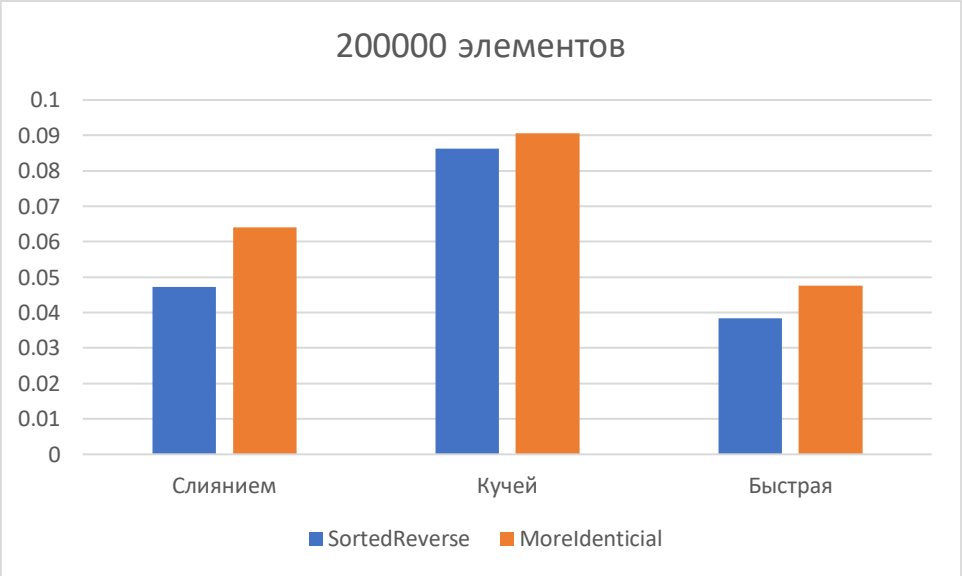
Результаты: пройдясь по гистограммам хотелось бы сказать, что данные сортировки не просто так считаются эффективными и быстрыми. Все показали примерно похожие результаты. При 2000 элементах существенных отличий между алгоритмами сортировок нет. А уже с 20000 элементов можно заметить, как сортировка кучей начинает понемножку отставать от двух других. Лидером из этих трёх алгоритмов является быстрая сортировка.

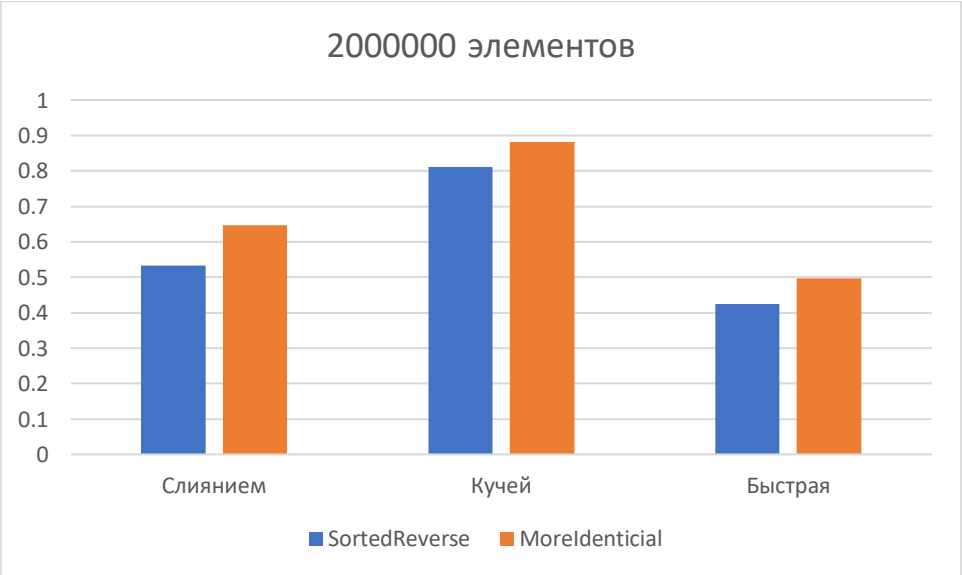
Отсортированные в обратном направлении и с большой долей одинаковых элементов массивы

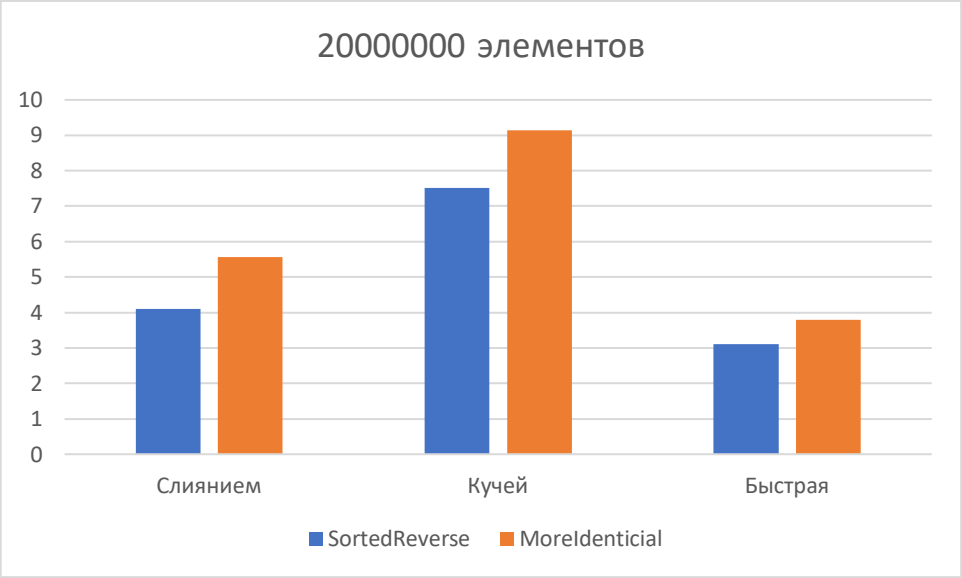
Замечание. Такого рода порождённые массивы рассматривались только для типа данных int.




20000 элементов
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
Слиянием
Кучей
Быстрая
SortedReverse
MoreIdenticial


200000 элементов
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
Слиянием
Кучей
Быстрая
SortedReverse
MoreIdenticial


2000000 элементов
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Слиянием
Кучей
Быстрая
SortedReverse
MoreIdenticial

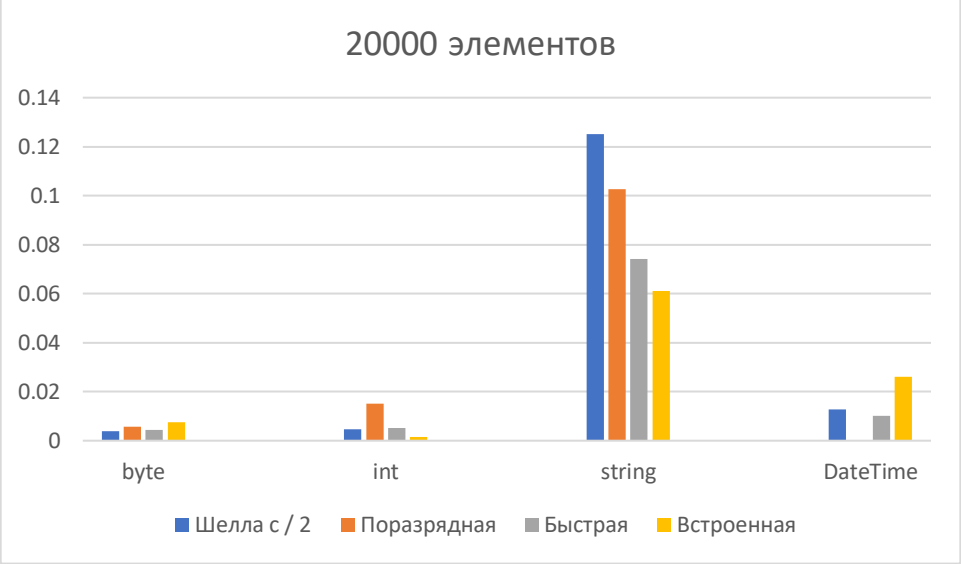


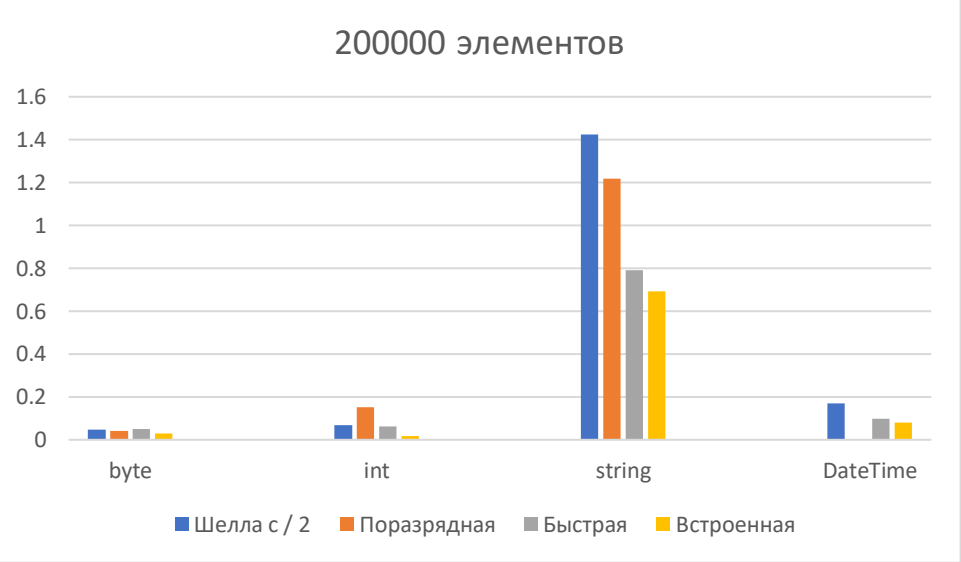
Результаты: итог очевиден, даже при таких порождённых массивах, лучшим остается быстрая сортировка в независимости от размера массива. На втором месте сортировка слиянием, которая тоже очень даже неплоха. Ну и на третьем месте находится сортировка кучей, которая стабильно хуже двух предыдущих.

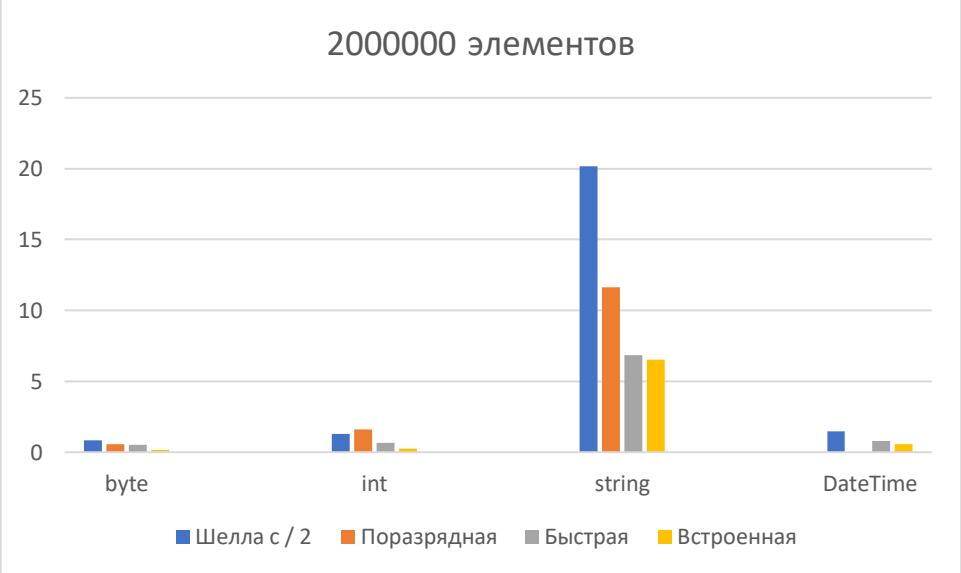
Сравнение лучших сортировок, взятых из каждого вида

Для сравнения лучших сортировок из простых схем была взята сортировка вставками, из продвинутых схем сортировка Шелла с / 2 и поразрядная, из быстрых схем быстрая сортировка, а также встроенная сортировка Array.Sort() для C#.

Случайно сгенерированные массивы

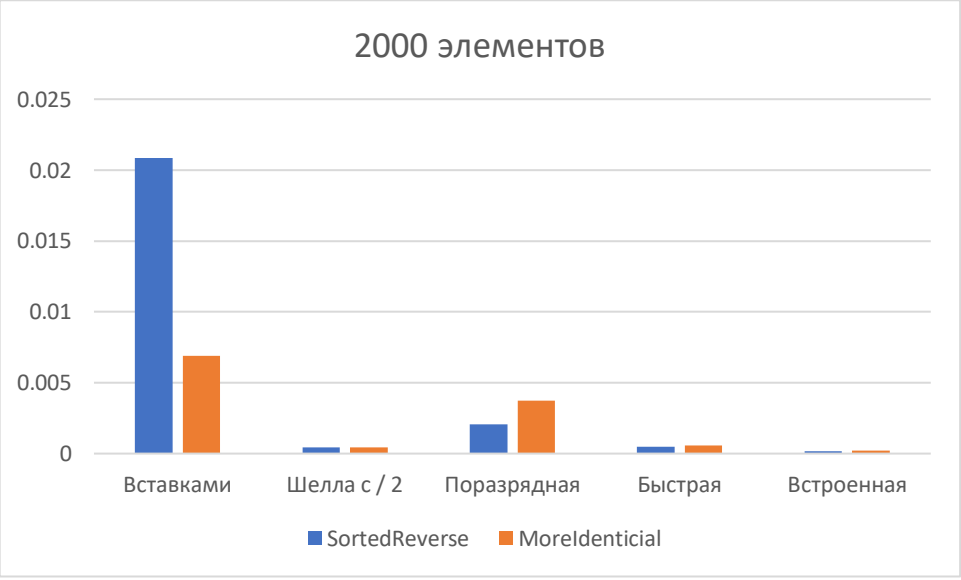

20000 элементов
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
byte
int
string
DateTime
Шелла с / 2
Поразрядная
Быстрая
Встроенная

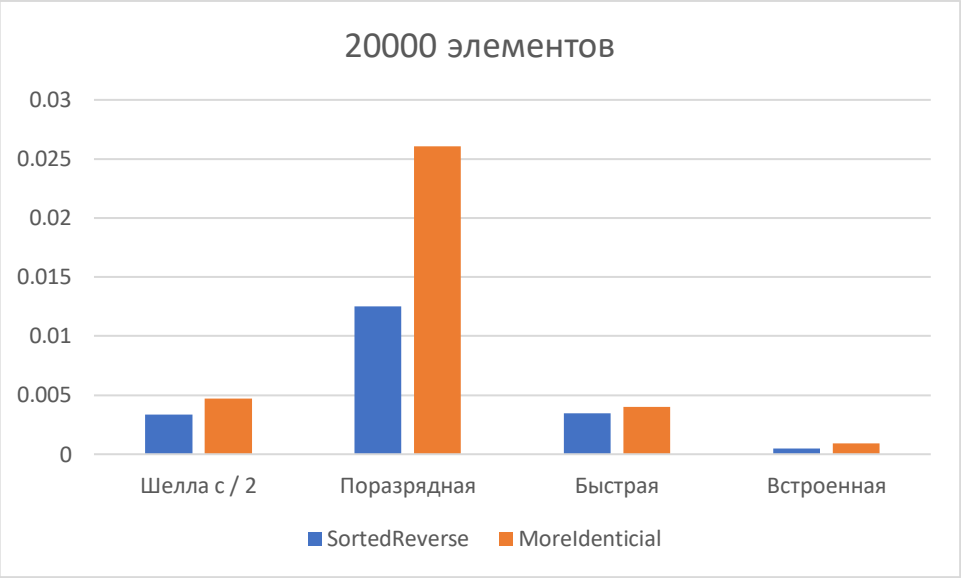

200000 элементов
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
byte
int
string
DateTime
Шелла с / 2
Поразрядная
Быстрая
Встроенная

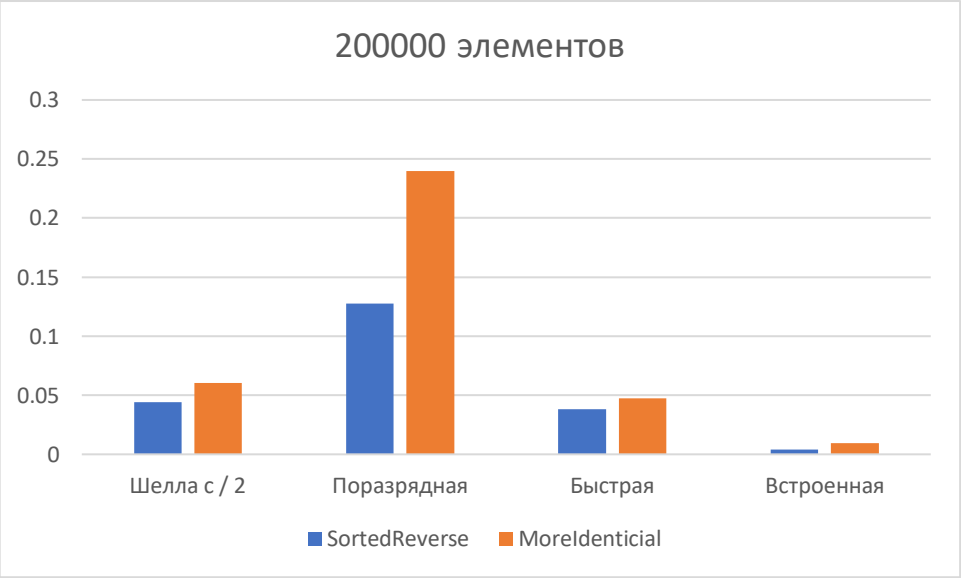

2000000 элементов
25
20
15
10
5
0
byte
int
string
DateTime
Шелла с / 2
Поразрядная
Быстрая
Встроенная

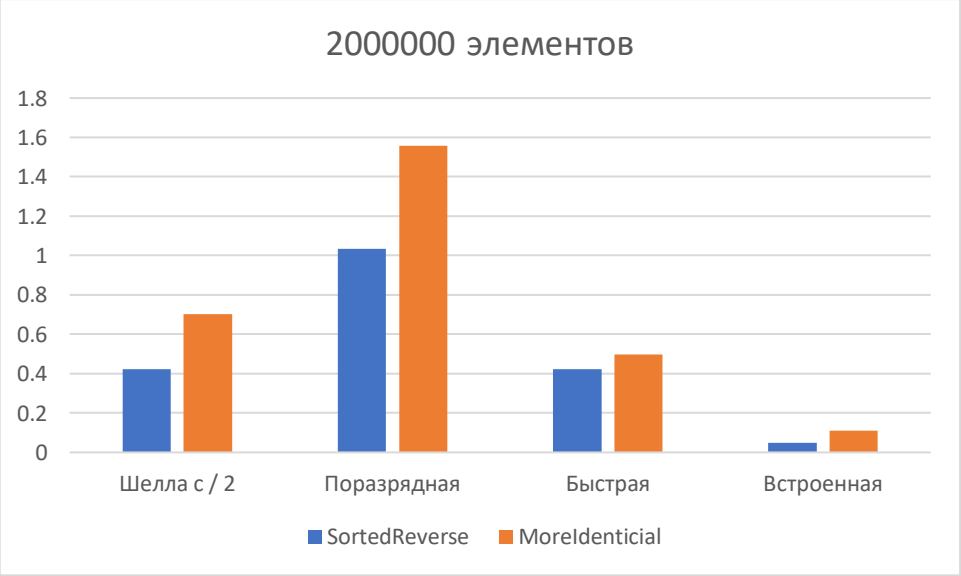
Результаты: уже при малых количествах элементов (2000) отчетливо видно, насколько неэффективна сортировка вставками. Поэтому, в последующих размерах массива, данная сортировка уже не рассматривалась. Также можно заметить неэффективность сортировки Шелла с / 2 для тип данных string, в сравнении с остальными. Продвинутые схемы сортировок, а именно Шелла с / 2 и поразрядная, вели себя примерно одинаково для всех типов, кроме string. Ну и самыми эффективными оказались быстрая и встроенная сортировки. Однако встроенная была немножко впереди.

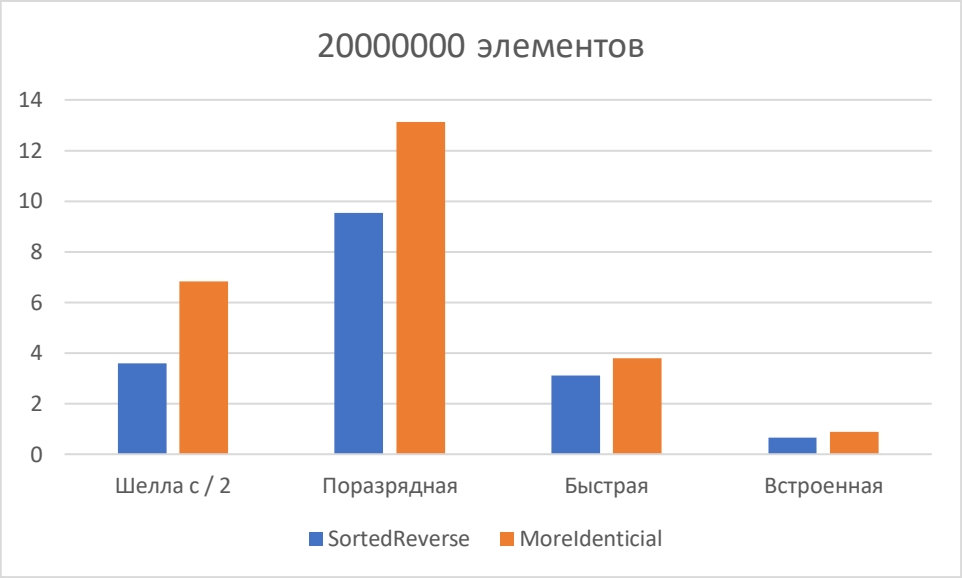
Отсортированные в обратном направлении и с большой долей одинаковых элементов массивы




20000 элементов
0.03
0.025
0.02
0.015
0.01
0.005
0
Шелла с / 2
Поразрядная
Быстрая
Встроенная
SortedReverse
MoreIdenticial


200000 элементов
0.3
0.25
0.2
0.15
0.1
0.05
0
Шелла с / 2
Поразрядная
Быстрая
Встроенная
SortedReverse
MoreIdenticial


2000000 элементов
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Шелла с / 2
Поразрядная
Быстрая
Встроенная
SortedReverse
MoreIdenticial



Результаты: при таких порожденных массивах также, как и при случайно сгенерированных, уже при малых количествах элементов (2000) отчетливо видно, насколько неэффективна сортировка вставками. Поэтому, в последующих размерах массива, данная сортировка уже не рассматривалась. Два вида продвинутых схем сортировок (Шелла с / 2 и поразрядная) на этот раз существенно отличались. Первый был существенно лучше, второго. Фаворитами также остаются встроенная и быстрая сортировки, но с значительным отрывом первого от второго.

Заключение

Таким образом, мы убедились в том, что существует большое количество алгоритмов сортировок (видов сортировок), каждый из которых имеет свои преимущества и недостатки в зависимости от типа данных и объема сортируемых элементов, и выбор конкретного алгоритма (вида) зависит от условий задач. Проверили совпадение теоретических данных с практическими.

Так, мы выяснили, что если необходимо отсортировать маленький массив, то можно, не заморачиваясь, использовать простую схему сортировок, например, сортировку вставками или же выбором. Если же нужно отсортировать большой массив, тогда подойдет встроенная или же быстрая схема такая, как быстрая сортировка или сортировка слиянием. Если массив порождён неслучайно, то также подойдет встроенная сортировка. Для таких типов данных, как string и struct, лучше использовать поразрядный алгоритм сортировки.

Приложение 1. Программный код

```
using System.Diagnostics;
using System.Numerics;
using System.Security.Cryptography;
using System.Text;
Random random = new Random();
string alpha = "aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ";

long freq = Stopwatch.Frequency;
Stopwatch stopwatch = new Stopwatch();

int N = Convert.ToInt32(Console.ReadLine()); // кол-во опытов

int cnt = 20000;
for (int i = 0; i < N; i++)
{
    int[] arr = GetArrayInt(cnt, TypesArrays.Random);

    stopwatch.Start();
    MergeSort(arr);
    stopwatch.Stop();

    double sec = (double)stopwatch.ElapsedTicks / freq;
    Console.WriteLine(sec);
    stopwatch.Reset();
}

int[] GetArrayInt(int count, TypesArrays type)
{
    int[] output = new int[count];

    switch (type)
    {
        case TypesArrays.Random:
            for (int i = 0; i < count; i++)
                output[i] = random.Next();
            break;
        case TypesArrays.MoreIdenticial:
            // Около 99% процентов одинаковых
            int c = random.Next();
            for (int i = 0; i < count; i++)
                output[i] = random.Next();

            for (int i = 0; i < count - count / 100; i++)
                output[random.Next(0, count - 1)] = c;
            break;
        case TypesArrays.SortedReverse:
            for (int i = 0; i < count; i++)
                output[i] = count - i;
            break;
    }

    return output;
}

byte[] GetArrayByte(int count, TypesArrays type)
{
    byte[] output = new byte[count];
```

```
    switch (type)
    {
        case TypesArrays.Random:
            for (int i = 0; i < count; i++)
                output[i] = (byte)random.Next();
            break;
    }

    return output;
}


#region Простые схемы


//Сортировка пузырьком
static void BubbleSort<T>(T[] arr) where T : IComparable<T>
{
    //Рассматриваем каждый элемент массива и "тяжелые" элементы меняем
    //с легкими, тем самым тяжелые будут в конце массива
    for (var i = 0; i < arr.Length; i++)
        for (var j = i + 1; j < arr.Length; j++)
            //CompareTo возвращает число в зависимости от того какой
аргумент больше.
            //Если левый больше, то будет положительное числа, если
меньше, то отрицательное.
            if (arr[j].CompareTo(arr[i]) < 0)
                Swap(ref arr[j], ref arr[i]);
}


//Сортировка вставками
static void InsertionSort<T>(T[] arr) where T : IComparable<T>
{
    //На каждом шаге считаем, что i элементов отсортированы между собой
    for (int i = 1; i < arr.Length; i++)
    {
        //Запоминаем вставляемый элемент
        T cur = arr[i];
        int j = i;

        //Так как i элементов отсортированы между собой, то нужно
        //сместить элементы вправо, освободив место для текущего
        while (j > 0 && arr[j - 1].CompareTo(cur) > 0)
        {
            Swap(ref arr[j], ref arr[j - 1]);
            j -= 1;
        }

        //Вставляем в нужное место
        arr[j] = cur;
    }
}


//Сортировка выбором
static void SelectionSort<T>(T[] arr) where T : IComparable<T>
{
    //Считаем, что на первых i элементах распаложены i минимальных
    //элементов массива, причем в отсортированном порядке
```

```
    for (var i = 0; i < arr.Length - 1; i++)
    {
        //Предполагаем, что минимальный элемент это i по счету
        var minIndex = i;

        //Из оставшихся частей исчем минимальный
        for (var j = i + 1; j < arr.Length; j++)
        {
            if (arr[j].CompareTo(arr[minIndex]) < 0)
            {
                minIndex = j;
            }
        }

        //Вставляем минимальный в нужную позицию
        Swap(ref arr[minIndex], ref arr[i]);
    }
}


//Сортировка перемешиванием (Шейкерная сортировка)
static void ShakerSort<T>(T[] arr) where T : IComparable<T>
{
    for (var i = 0; i < arr.Length / 2; i++)
    {
        //Идем слева направо
        for (var j = i; j < arr.Length - i - 1; j++)
        {
            if (arr[j].CompareTo(arr[j + 1]) > 0)
                Swap(ref arr[j], ref arr[j + 1]);
        }

        //Идем справа налево
        for (var j = arr.Length - 2 - i; j > i; j--)
        {
            if (arr[j - 1].CompareTo(arr[j]) > 0)
                Swap(ref arr[j - 1], ref arr[j]);
        }
    }
}

#endregion


#region Продвинутые схемы

//Сортировка Шелла (с делением на 2)
static void ShellSort<T>(T[] arr) where T : IComparable<T>
{
    for (int gap = arr.Length / 2; gap > 0; gap /= 2)
        for (int i = gap; i < arr.Length; i++)
        {
            //Сохраняем текущий элемент
            var cur = arr[i];
            var k = i;

            //Применяем сортировку вставками, но с шагом gap
            while (k >= gap && arr[k - gap].CompareTo(cur) > 0)
            {
                arr[k] = arr[k - gap];
```

```
                k -= gap;
            }

            //Обновляем текущий элемент
            arr[k] = cur;
        }
}


//Сортировка Шелла (с последовательностью Хиббарда)
static void ShellHibSort<T>(T[] arr) where T : IComparable<T>
{
    //Получение последнего числа последовательности Хиббарда для данного
массива
    int gap = (int)Math.Pow(2, (int)(Math.Log2(arr.Length + 1))) - 1;


    //Применяем сортировку вставками, но с шагом gap
    while (gap > 0)
    {
        for (int i = gap; i < arr.Length; i++)
        {
            //Сохраняем текущий элемент
            var cur = arr[i];
            var k = i;

            //Применяем сортировку вставками, но с шагом gap
            while (k >= gap && arr[k - gap].CompareTo(cur) > 0)
            {
                arr[k] = arr[k - gap];
                k -= gap;
            }

            //Обновляем текущий элемент
            arr[k] = cur;
        }

        //Обновляем шаг gap
        gap = gap / 2;
    }
}

//Поразрядная сортировка для строк
static void RadixSortS(string[] arr, int min = 0, int max = 255)
{
    //Выделение вспомогательного массива
    string[] output = new string[arr.Length];

    //Вычисление строки максимальной длины
    int Max = arr.Max(value =>  value.Length);

    var GetKey = (char c) =>
    {
        if (65 <= c && c <= 77)
            return (char)((c - 65) * 2 + 1);
        else if (77 < c && c <= 90)
            return (char)(26 + (c - 78) * 2 + 1);
        else if (97 <= c && c <= 109)
            return (char)((c - 97) * 2);
        else if (109 < c && c <= 122)
```

```
            return (char)(26 + (c - 110) * 2);
        return c;
    };

    //Поразрядная сортировка строк начиная с последнего символа
    for (int j = Max - 1;  j >= 0; j--)
    {
        //Массив для определения количества подмассивов
        int[] counts = Enumerable.Repeat(0, 52).ToArray();


        for (int i = 0; i < arr.Length; i++)
        {
            char val = GetKey(arr[i][j]);
            counts[val]++;
        }

        //Составление массива префиксных сумм
        for (int i = 1; i < counts.Length; i++)
            counts[i] += counts[i - 1];

        //Сортировка по разрядам
        for (int i = arr.Length - 1; i >= 0; i--)
        {
            output[counts[GetKey(arr[i][j])] - 1] = arr[i];
            counts[GetKey(arr[i][j])]--;
        }

        //Обратное копирование
        for (int i = 0; i < arr.Length; i++)
            arr[i] = output[i];
    }
}


//Поразрядная сортировка для целых чисел
static void RadixSort<T>(T[] arr) where T : IComparable<T>, IConvertible,
INumber<T>
{
    //Выделение вспомогательного массива
    T[] output = new T[arr.Length];

    //Вычисление максимального по модулю числа
    int MaxABS = Math.Max(Math.Abs(Convert.ToInt32(arr.Max())),
Math.Abs(Convert.ToInt32(arr.Min())));


    //Сортировка по модулю по разрядам начиная с первого
    for (int exp = 1; MaxABS / exp > 0; exp *= 10)
    {
        //Массив для определения количества подмассивов
        int[] counts = Enumerable.Repeat(0, 10).ToArray();

        for (int i = 0; i < arr.Length; i++)
            counts[(Math.Abs(Convert.ToInt32(arr[i])) / exp) % 10]++;

        //Составление массива префиксных сумм
        for (int i = 1; i < counts.Length; i++)
            counts[i] += counts[i - 1];
```

```
        //Сортировка по модулю для текущего рязряда
        for (int i = arr.Length - 1; i >= 0; i--)
        {
            output[counts[(Math.Abs(Convert.ToInt32(arr[i])) / exp) % 10]
- 1] = arr[i];
            counts[(Math.Abs(Convert.ToInt32(arr[i])) / exp) % 10]--;
        }

        //Обратное копирование
        for (int i = 0; i < arr.Length; i++)
            arr[i] = output[i];
    }

    //Так как мы сортировали по модулю, а не по значению, нам нужно
отрицательные
    //элементы вставить в начало в обратном порядке
    int minusIndex = 0;
    for (int i = output.Length - 1; i >= 0; i--)
        if (Convert.ToInt32(output[i]) < 0)
            arr[minusIndex++] = output[i];

    for (int i = 0; i < output.Length; i++)
        if (Convert.ToInt32(output[i]) >= 0)
            arr[minusIndex++] = output[i];
}


#endregion


#region Быстрые схемы

    //Сортировка слиянием
static void MergeSort<T>(T[] arr) where T : IComparable<T>
{
    //Выделение дополнительного массива
    T[] temps = new T[arr.Length];
    Array.Copy(arr, temps, arr.Length);

    //Дополнительный переменные
    int curLeft, curRight;  //Переменные для отслеживания границ текущего
подмассива
    int i, j, m;            //Вспомогательные переменные для прохода по
текущему подмассиву

    //Нерекурсивная сортировка слиянием снизу вверх (длина рассматриваемых
подмассивов будет увеличиваться вдвое)
    for (int k = 1; k < arr.Length; k *= 2)
    {
        //Проход по подмассивам длинной k * 2, где каждый из них
разбивается на подмассивы размером k
        for (int left = 0; left + k < arr.Length; left += k * 2)
        {
            curLeft = left + k;
            curRight = curLeft + k;

            //Если подмассив стал сравним по размерам с исходным массивом
            if (curRight > arr.Length)
                curRight = arr.Length;
```

```
            //Слияние
            //m - переменная для вставки в нужную позицию
            //i - переменная начала левого подмассива
            //j - переменная начала правого подмассива
            i = m = left; j = curLeft;
            while (i < curLeft && j < curRight)
            {
                if (arr[i].CompareTo(arr[j]) <= 0)
                    temps[m] = arr[i++];
                else
                    temps[m] = arr[j++];

                m++;//Инкеремент после вставки
            }

            //На случай того, если еще остались элементы, которые не были
вставлены
            while (i < curLeft)
                temps[m++] = arr[i++];

            while (j < curRight)
                temps[m++] = arr[j++];

            //Изменение исходного массива
            for (m = left; m < curRight; m++)
                arr[m] = temps[m];
        }
    }
}


//Сортировка кучей
static void HeapSort<T>(T[] arr) where T : IComparable<T>
{
    //Упорядочивание кучи
    for (int i = 1; i < arr.Length; i++)
        if (arr[i].CompareTo(arr[(i - 1) / 2]) > 0)
        {
            int j = i;

            //Если предок меньше потомка, то меняем их
            //Далее рассматриваем предка в качестве потомка
            while (arr[j].CompareTo(arr[(j - 1) / 2]) > 0)
            {
                Swap(ref arr[j], ref arr[(j - 1) / 2]);
                j = (j - 1) / 2;
            }
        }



    for (int i = arr.Length - 1; i > 0; i--)
    {
        //Корневой элемент самый большой, поэтому отправляем его в конец
        //Далее рассматриваем лишь оставшуюся часть массива
        Swap(ref arr[0], ref arr[i]);

        //Заново переупорядочиваем дерево начиная с корневого (0 элемент)
        int j = 0, index;
```

```
        do
        {
            //Индекс первого потомка
            index = (2 * j + 1);

            //Если второй потомок оказался больше первого, то index
изменим на индекс второго потомка
            if (index < (i - 1) && arr[index].CompareTo(arr[index + 1]) <
0)
                index++;

            //Меняем предка с большим потомком
            if (index < i && arr[j].CompareTo(arr[index]) < 0)
                Swap(ref arr[j], ref arr[index]);

            //Далее рассматриваем текущего потомка в качестве предка
            j = index;

        } while (index < i);
    }
}


//Быстрая сортировка
static void QuickSort<T>(T[] arr) where T : IComparable<T>
{
    //Левый и правый индексы массива
    int left = 0;
    int right = arr.Length - 1;

    //Имитация стека с помощью массива
    //Этот стек нам нужен будет для выполнения рекурсивной функции
итерационным способом
    int top = -1;
    int[] stack = new int[arr.Length + 1];

    //В стеке парами будут хранняться индексы подмассивов
    stack[++top] = left;
    stack[++top] = right;


    //Динамическое выполнение алгоритма быстрой сортировки
    while (top >= 0)
    {
        //Извлечение из стека индексов подмассива
        var j = right = stack[top--];
        var i = left = stack[top--];


        //Выбор медианы из трех в качестве опорного
        //При этом сами эти три элементы мы предварительно тоже сортируем
        int mid = (left + right) / 2;
        if (arr[mid].CompareTo(arr[left]) < 0)  Swap(ref arr[left], ref
arr[mid]);
        if (arr[right].CompareTo(arr[mid]) < 0) Swap(ref arr[mid], ref
arr[right]);
        if (arr[mid].CompareTo(arr[left]) < 0)  Swap(ref arr[right], ref
arr[mid]);

        var pivot = arr[mid];
```

```
            //Разбиение чисел относительно опорного
            while (i <= j)
            {
                while (arr[i].CompareTo(pivot) < 0) i++;

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

                if (i <= j) Swap(ref arr[i++], ref arr[j--]);
            }

            //Добавление индексов подмассива в стек
            if (j > left)
            {
                stack[++top] = left;
                stack[++top] = j;
            }

            if (i < right)
            {
                stack[++top] = i;
                stack[++top] = right;
            }
        }
    }


    #endregion


    #region Дополнительные функции

    static void Swap<T>(ref T first, ref T second)
    {
        //Выделение дополнительной переменной
        T temp = first;

        first = second;
        second = temp;
    }

    static void Print<T>(T[] arr)
    {
        for (int i = 0; i < arr.Length - 1; i++)
            Console.Write(arr[i] + " ");
        if (arr.Length != 0)
            Console.Write(arr[arr.Length - 1] + "\n");

    }

    bool IsSort<T>(T[] arr) where T : IComparable<T>
    {
        for (int i = 0; i < arr.Length - 1; i++)
            if (arr[i].CompareTo(arr[i + 1]) > 0)
                return false;
        return true;
    }


    string[] GetStrings(int count, int length)
```

```
{
    string[] strings = new string[count];

    for (int i = 0; i < strings.Length; i++)
    {
        StringBuilder sb = new StringBuilder();
        for (int j = 0; j < length; j++)
        {
            sb.Append(alpha[random.Next(0, alpha.Length - 1)]);
        }

        strings[i] = sb.ToString();
    }

    return strings;
}

DateTime[] GetArrayDateTime(int count)
{
    DateTime[] dateTimes = new DateTime[count];

    for (int i = 0; i < count; i++)
    {
        DateTime start = new DateTime(1995, 1, 1);
        dateTimes[i] = start.AddDays((DateTime.Today -
start).Days).AddHours(random.Next(23)).AddMinutes(random.Next(59)).AddSeco
nds(random.Next(59));
    }

    return dateTimes;
}
#endregion

enum TypesArrays
{
    Random,
    MoreIdenticial,
    Identicial,
    Sorted,
    SortedReverse
}
```