

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Уральский федеральный университет имени первого Президента России Б.Н.Ельцина»  
Институт радиоэлектроники и информационных технологий – РИИТ

## Анализ сложности алгоритмов сортировки строк

Отчёт по лабораторной работе

по дисциплине «Алгоритмы, структуры данных и анализ сложности»

Вариант 6

Выполнил:  
студент группы

Преподаватель:  
доцент, к.ф.-м.н.  
Трофимов С.П.

**Оглавление**

Задание.....	3
Теоретическая часть.....	4
Инструкция пользователя .....	6
Инструкция программиста.....	8
Тестирование .....	9
Выводы .....	10
Литература.....	12
Приложение .....	13

## Задание

Реализовать один из алгоритмов сортировки строк:

### 6. Пирамидальная сортировка HeapTree

Выбор алгоритма выбирается по согласованию с преподавателем.

Для алгоритма определить сложность относительно наиболее характерной операции (сравнение, перестановка и др.). Вид функции сложности  $F(n)$  подобрать в соответствии с теорией. Например, для оптимальных алгоритмов  $F(n) = C \cdot n \cdot \log_2(n)$ . Найти также коэффициент пропорциональности  $C$ . Для аппроксимации можно использовать метод наименьших квадратов и сервис «Поиск решения».

План проведения эксперимента с алгоритмом называется массовой задачей. Представьте план в виде xml-файла.

Результаты решения массовой задачи записать в текстовый файл в 2 столбика: длина массива, количество операций. Файл импортировать в Excel. В «шапке» листа указать параметры тренда, вычислить квадратичные невязки и минимизировать их сумму с помощью «Данные-Поиск решения»

## Теоретическая часть

Пирамидальная сортировка (англ. Heapsort, «Сортировка кучей») была предложена Дж. Уильямсом в 1964 году.

Сортировка пирамидой использует бинарное сортирующее дерево. Сортирующее дерево — это такое дерево, у которого выполнены условия:

- 1) Каждый лист имеет глубину либо  $d-1$ , либо  $d$ , где  $d$  — максимальная глубина дерева.
- 2) Значение в любой вершине не больше значения её двух потомков.

Удобная структура данных для сортирующего дерева — такой массив `Array`, что `Array[0]` — элемент в корне, а потомки элемента `Array[i]` являются `Array[2i+1]` и `Array[2i+2]`, где  $i = 0, \dots, n/2-1$ .

Алгоритм сортировки будет состоять из двух основных шагов:

- 1) Выстраиваем элементы массива в виде сортирующего дерева:

$\text{Array}[i] \leq \text{Array}[2i+1],$

$\text{Array}[i] \leq \text{Array}[2i+2]$

при  $i = n/2-1, \dots, 0$ .

Для этого сравниваем `Array[i]` с  $\min \{ \text{Array}[2i+1], \text{Array}[2i+2] \}$  и при необходимости переставляем.

Этот шаг требует  $O(n)$  операций.

- 2) Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем `Array[0]` и `Array[n-1]`, преобразовываем `Array[0], Array[1], \dots, Array[n-2]` в сортирующее дерево. Затем переставляем `Array[0]` и `Array[n-2]`, преобразовываем `Array[0], Array[1], \dots, Array[n-3]` в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда `Array[0], Array[1], \dots, Array[n-1]` — упорядоченная последовательность.

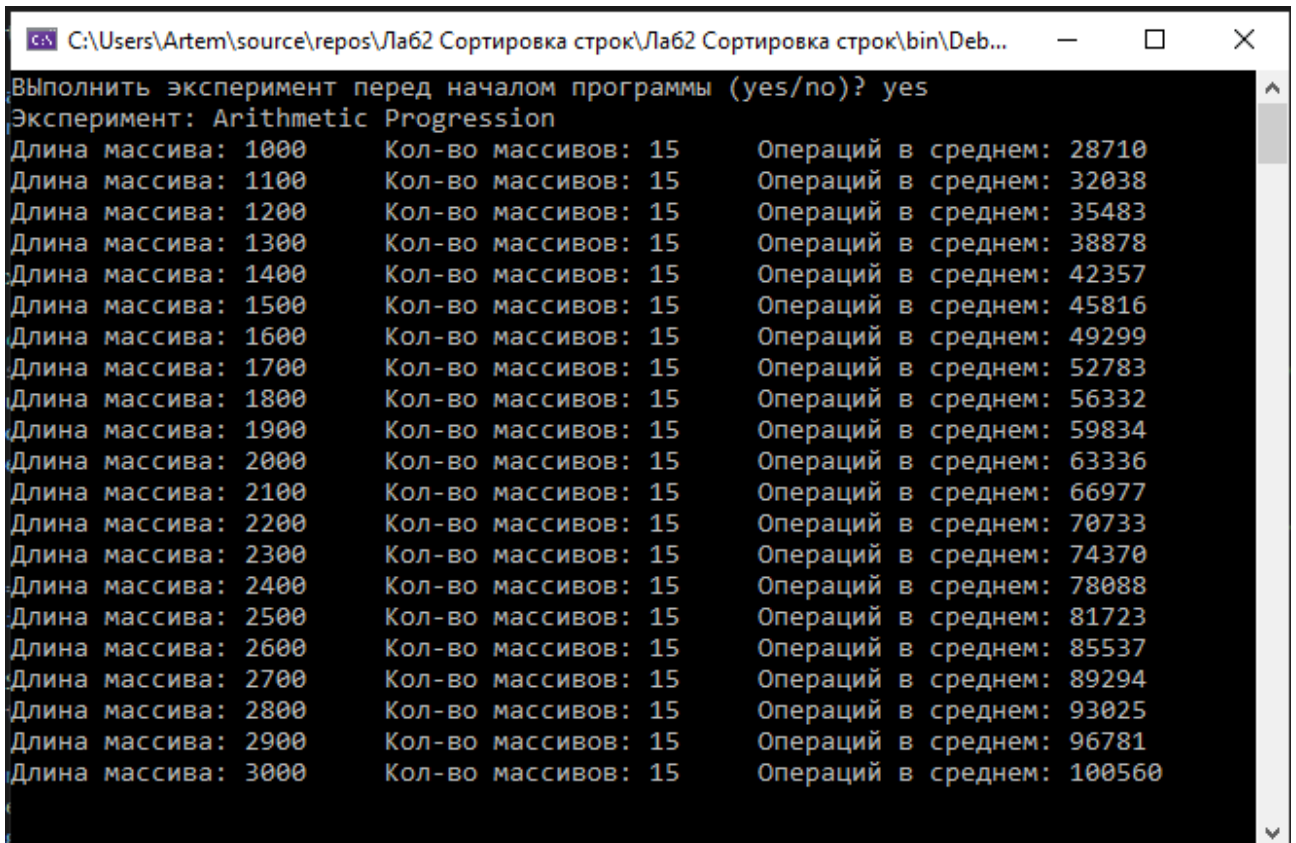
Этот шаг требует  $O(n \cdot \log n)$  операций.

Особенности алгоритма:

- 1) Работает в худшем, в среднем и в лучшем случае (то есть гарантированно) за  $\Theta(n \log n)$  операций при сортировке  $n$  элементов.
- 2) Количество применяемой служебной памяти не зависит от размера массива (то есть,  $O(1)$ ).
- 3) Из-за сложности алгоритма выигрыш получается только на больших  $n$
- 4) Неустойчив

## Инструкция пользователя

При запуске программы открывается консоль, где пользователю предлагается перед сортировкой собственного массива провести эксперимент и оценить производительность алгоритма. Если он согласен, то на консоль будут выведены результаты эксперимента.



```
C:\Users\Artem\source\repos\Ла62 Сортировка строк\Ла62 Сортировка строк\bin\Deb...
Выполнить эксперимент перед началом программы (yes/no)? yes
Эксперимент: Arithmetic Progression
Длина массива: 1000    Кол-во массивов: 15    Операций в среднем: 28710
Длина массива: 1100    Кол-во массивов: 15    Операций в среднем: 32038
Длина массива: 1200    Кол-во массивов: 15    Операций в среднем: 35483
Длина массива: 1300    Кол-во массивов: 15    Операций в среднем: 38878
Длина массива: 1400    Кол-во массивов: 15    Операций в среднем: 42357
Длина массива: 1500    Кол-во массивов: 15    Операций в среднем: 45816
Длина массива: 1600    Кол-во массивов: 15    Операций в среднем: 49299
Длина массива: 1700    Кол-во массивов: 15    Операций в среднем: 52783
Длина массива: 1800    Кол-во массивов: 15    Операций в среднем: 56332
Длина массива: 1900    Кол-во массивов: 15    Операций в среднем: 59834
Длина массива: 2000    Кол-во массивов: 15    Операций в среднем: 63336
Длина массива: 2100    Кол-во массивов: 15    Операций в среднем: 66977
Длина массива: 2200    Кол-во массивов: 15    Операций в среднем: 70733
Длина массива: 2300    Кол-во массивов: 15    Операций в среднем: 74370
Длина массива: 2400    Кол-во массивов: 15    Операций в среднем: 78088
Длина массива: 2500    Кол-во массивов: 15    Операций в среднем: 81723
Длина массива: 2600    Кол-во массивов: 15    Операций в среднем: 85537
Длина массива: 2700    Кол-во массивов: 15    Операций в среднем: 89294
Длина массива: 2800    Кол-во массивов: 15    Операций в среднем: 93025
Длина массива: 2900    Кол-во массивов: 15    Операций в среднем: 96781
Длина массива: 3000    Кол-во массивов: 15    Операций в среднем: 100560
```

Рисунок 1 – Проведение эксперимента

Далее, независимо от того, производился опыт или нет, пользователю предлагается ввести собственный массив строк, разделяя его элементы пробелом. После ввода на консоль будет выведен уже отсортированный массив.

```
C:\Users\Artem\source\repos\Ла62 Сортировка строк\Ла62 Сортировка строк\bin\Deb...
Эксперимент: Geometric Progression
Длина массива: 10      Кол-во массивов: 15      Операций в среднем: 95
Длина массива: 20      Кол-во массивов: 15      Операций в среднем: 247
Длина массива: 40      Кол-во массивов: 15      Операций в среднем: 588
Длина массива: 80      Кол-во массивов: 15      Операций в среднем: 1396
Длина массива: 160     Кол-во массивов: 15      Операций в среднем: 3255
Длина массива: 320     Кол-во массивов: 15      Операций в среднем: 7408
Длина массива: 640     Кол-во массивов: 15      Операций в среднем: 16608
Длина массива: 1280    Кол-во массивов: 15      Операций в среднем: 36816
Длина массива: 2560    Кол-во массивов: 15      Операций в среднем: 80779
Длина массива: 5120    Кол-во массивов: 15      Операций в среднем: 176150

Эксперимент: Geometric Progression
Длина массива: 10      Кол-во массивов: 15      Операций в среднем: 100
Длина массива: 50      Кол-во массивов: 15      Операций в среднем: 807
Длина массива: 250     Кол-во массивов: 15      Операций в среднем: 5684
Длина массива: 1250    Кол-во массивов: 15      Операций в среднем: 37208
Длина массива: 6250    Кол-во массивов: 15      Операций в среднем: 229602

Введите строки для сортировки, разделяя их пробелом: fde abc aan liv kjh evc
Отсортированный массив: aan abc evc fde kjh liv _
```

Рисунок 2 – Сортировка пользовательского массива строк

## Инструкция программиста

В программе, написанной на языке C#, алгоритм пирамидальной сортировки представлен в классе Program. Для его реализации была написана функция HeapSort:

```
public static int HeapSort(string[] array)
```

принимая массив строк и сортирующая его алгоритмом HeapSort. Для возможности оценивать производительность сортировки функция возвращает значение счётчика той операций, которая вызывалась чаще всего.

Для удобства самая частая операция в HeapSort вынесена в отдельную функцию Heapify:

```
private static void Heapify(string[] array, int heapSize, int i, ref int swapsCount, ref int branchesCount)
```

принимая массив строк, размер формируемой двоичной кучи, индекс корневого узла и ссылки на счетчики операций. Данная функция используется для формирования из массива строк бинарного дерева и вызывается как в HeapSort, так и рекурсивно.

Помимо этих основных функций, в программе реализованы вспомогательные. Это функция PrintArray:

```
private static void PrintArray(string[] array)
```

принимая массив строк и выводящая его на консоль, а также функции GetExperimentResults:

```
private static void GetExperimentResults()
```

выполняющая экспериментальные вычисления и выводящая их в консоль, и GetOperationsCount:

```
private static int GetOperationsCount(int minElement, int maxElement, int repeatsCount, int arrayLength)
```

принимая параметры для создания массивов строк (их длину, диапазон значений элементов, количество массивов); она формирует массивы, запускает функцию сортировки для них и возвращает счётчик самой часто вызываемой операции.



## Тестирование

Кроме основных и вспомогательных, программа также содержит функцию Test. Она содержит тесты, проверяющий корректность работы алгоритма сортировки и выполняется в момент запуска программы. Если один или несколько тестов завершились неудачно, на консоль будут выведены соответствующие сообщения, и программа завершит работу. Часть кода тестов представлена ниже, с полным кодом можно ознакомиться в Приложении.

```
ссылка.1
private static bool Test() // Тестовая функция
{
    var isAllTestsCompleted = true;
    {
        var startArray = new[] { "bdg", "avc", "hdf", "aa", "dg", "chd" };
        var expectedArray = new[] { "aa", "avc", "bdg", "chd", "dg", "hdf" };
        HeapSort(startArray);
        if (!Enumerable.SequenceEqual(startArray, expectedArray))
        {
            Console.WriteLine("Test 1 wasn't passed");
            isAllTestsCompleted = false;
        }
    }
    {
        var startArray = new[] { "enfgh", "sgasf", "asd", "ksd", "asa", "bhj", "ksgd", "assa", "bhdfj" };
        var expectedArray = new[] { "asa", "asd", "assa", "bhdfj", "bhj", "enfgh", "ksd", "ksgd", "sgasf" };
        HeapSort(startArray);
        if (!Enumerable.SequenceEqual(startArray, expectedArray))
        {
            Console.WriteLine("Test 2 wasn't passed");
            isAllTestsCompleted = false;
        }
    }
    {
        var startArray = new[] { "mbdsdfg", "avghssdc", "hjsdf", "abfga", "dhjsg", "lnvchd" };
        var expectedArray = new[] { "abfga", "avghssdc", "dhjsg", "hjsdf", "lnvchd", "mbdsdfg" };
        HeapSort(startArray);
        if (!Enumerable.SequenceEqual(startArray, expectedArray))
        {
            Console.WriteLine("Test 3 wasn't passed");
            isAllTestsCompleted = false;
        }
    }
}
```

Рисунок 3 – Часть кода тестирующей функции Test

Помимо тестирования, программа предоставляет возможность провести эксперимент с пирамидальной сортировкой на больших массивах данных. Для этого в xml-файле Experiment был создан план эксперимента. Он включает в себя несколько элементов nodes, каждый из которых описывает свою часть эксперимента: какое количество массивов будет сгенерировано, какой длины, с какими значениями и по какому принципу эти массивы будут изменяться (здесь

реализовано изменения длины массива в арифметической и геометрической прогрессиях).

Данный документ обрабатывается в функции `GetExperimentResults`: значения атрибутов каждого элемента эксперимента будут получены и использованы для генерации массивов (все значения их элементов выбираются случайным образом из заданного диапазона) и последующей сортировки.

Полученные результаты (длины массивов и кол-во операций для их сортировки) заносятся в таблицу Excel. В добавление к ним рассчитываем также значения тренда для каждого случая и строим графики функции сложности. Скорее всего, они будут отстоять далеко друг от друга, поэтому для их сближения применим метод наименьших квадратов: рассчитываем для каждого случая квадратичные невязки, а затем получим коэффициент для функции сложности с помощью сервиса Excel «Поиск решения».

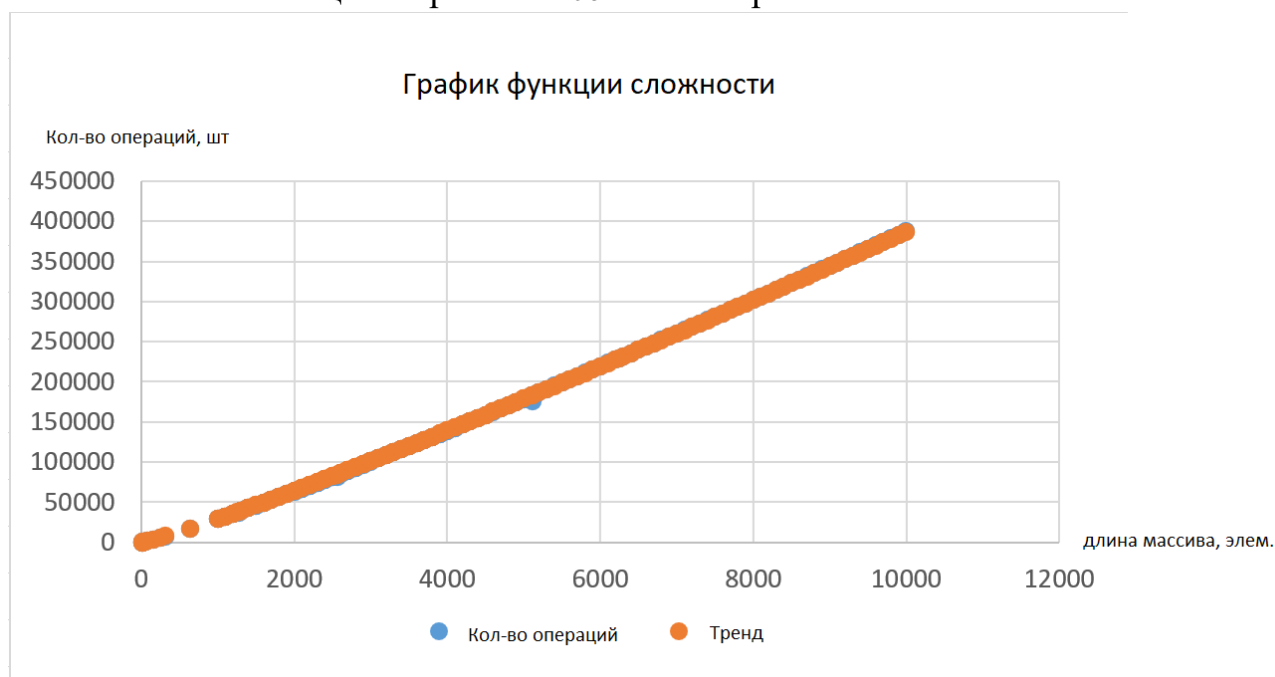


Рисунок 4 – Графики функции сложности после применения сервиса «Поиск решения»

В нашем случае для пирамидальной сортировки строк мы получаем коэффициент пропорциональности  $C = 2,91049832015149 \approx 2,91$ , а функция сложности для данного алгоритма принимает вид  $F(n) = 2,91 * n * \log(n)$ .

## Выводы

В данной работе мы познакомились с одним из алгоритмов сортировки – пирамидальной сортировкой HeapTree, написали программу для сортировки массивов строк с его использованием, а также провели эксперимент для оценки сложности данного алгоритма. Полученная программа работает исправно и позволяет достаточно быстро сортировать массивы из тысяч строк. Это же подтверждается и результатами эксперимента: алгоритм имеет сложность вида  $O(n * \log(n))$  и, к тому же, требует константное значение дополнительной памяти, т. е. не зависит от размера входных данных.

Всё это говорит о том, что пирамидальная сортировка является одной из самых эффективных для сортировки большого объёма данных, однако алгоритм также имеет и недостатки, например, неустойчивость и выигрыш в производительности только на больших значениях  $n$ . Таким образом, в определённых ситуациях целесообразнее использовать другие алгоритмы сортировки, более эффективные для выбранной задачи.

## Литература

- 1) Левитин А. В. Глава 6. Метод преобразования: Пирамиды и пирамидальная сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 275—284. — 576 с.
- 2) Федоряева Т. И. Комбинаторные алгоритмы: Учебное пособие /Новосиб. гос. ун-т. Новосибирск, 2011. 118 с.
- 3) [https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%80%D0%B0%D0%B%D0%B8%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F\\_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%80%D0%B0%D0%B%D0%B8%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) Пирамидальная сортировка
- 4) <https://habr.com/ru/company/otus/blog/460087/> Пирамидальная сортировка (HeapSort)

## Приложение

### Код класса Program:

```
using System;
using System.Linq;
using System.Xml;

namespace Лаб2_Сортировка_строк
{
    class Program
    {
        static void Main(string[] args)
        {
            if (!Test())
                return;
            Console.WriteLine("Выполнить эксперимент перед началом программы (yes/no)? ");
            var answer = Console.ReadLine();
            if (answer == "yes")
                GetExperimentResults();
            Console.WriteLine("Введите строки для сортировки, разделяя их пробелом: ");
            var input = Console.ReadLine().Split();
            HeapSort(input);
            Console.WriteLine("Отсортированный массив: ");
            PrintArray(input);
        }

        public static int HeapSort(string[] array) // Сортировка массива
        {
            var heapSize = array.Length;
            var swapsCount = 0; // Счётчик swap-ов
            var branchesCount = 0; // Счётчик ветвлений

            for (var i = heapSize / 2 - 1; i >= 0; i--) // Построение кучи (перегруппируем
массив)
                Heapify(array, heapSize, i, ref swapsCount, ref branchesCount);

            for (var i = heapSize - 1; i >= 0; i--) // По очереди извлекаем элементы из
кучи
            {
                var swap = array[0]; // Перемещаем текущий корень в конец
                array[0] = array[i];
                array[i] = swap;
                swapsCount++;
                Heapify(array, i, 0, ref swapsCount, ref branchesCount); // Вызываем
процедуру heapify на уменьшенной куче
            }
            return swapsCount >= branchesCount ? swapsCount : branchesCount; // Возвращаем
счётчик самой частой операции
        }

        private static void Heapify(string[] array, int heapSize, int i, ref int swapsCount,
ref int branchesCount) // Преобразования в двоичную кучу размера heapSize поддерева с
корневым узлом i (индекс в array)
        {
            var largestElem = i; // Инициализируем наибольший элемент как корень
            var leftChild = 2 * i + 1; // left = 2 * i + 1
            var rightChild = 2 * i + 2; // right = 2 * i + 2

            if (leftChild < heapSize && String.Compare(array[leftChild], array[largestElem],
StringComparison.Ordinal) > 0) // Если левый дочерний элемент больше корня
                largestElem = leftChild;
        }
    }
}
```

```

        if (rightChild < heapSize && String.Compare(array[rightChild],
array[largestElem], StringComparison.Ordinal) > 0) // Если правый дочерний элемент больше,
чем самый большой элемент на данный момент
            largestElem = rightChild;

        if (largestElem != i) // Если самый большой элемент не корень
        {
            var swap = array[i];
            array[i] = array[largestElem];
            array[largestElem] = swap;
            swapsCount++;
            Heapify(array, heapSize, largestElem, ref swapsCount, ref branchesCount);
// Рекурсивно преобразуем в двоичную кучу затронутое поддерево
        }
        branchesCount += 3;
    }

    private static void PrintArray(string[] array) // Вывод массива строк на экран
    {
        for (var i = 0; i < array.Length; i++)
            Console.Write(array[i] + " ");
        Console.Read();
    }

    private static void GetExperimentResults() // Выполнение эксперимента
    {
        var xDoc = new XmlDocument();
        xDoc.Load(@"C:\Users\Artem\source\repos\Лаб2 Сортировка строк\Лаб2 Сортировка
строк\Experiment.xml"); // Получаем xml-файл эксперимента
        var xRoot = xDoc.DocumentElement;
        var experiment = xRoot.SelectSingleNode("experiment");
        foreach (XmlNode node in experiment.ChildNodes)
        {
            var minElement = int.Parse(node.SelectSingleNode("@minElement").Value);
// Получение значений атрибутов каждого node
            var maxElement = int.Parse(node.SelectSingleNode("@maxElement").Value);
            var startLength = int.Parse(node.SelectSingleNode("@startLength").Value);
            var maxLength = int.Parse(node.SelectSingleNode("@maxLength").Value);
            var repeat = int.Parse(node.SelectSingleNode("@repeat").Value);
            var name = node.SelectSingleNode("@name").Value;
            if (name == "Arithmetic Progression")
            {
                var diff = int.Parse(node.SelectSingleNode("@diff").Value);
                Console.WriteLine("Эксперимент: " + name);
                for (var length = startLength; length <= maxLength; length += diff)
                    Console.WriteLine("Длина массива: " + length + "\t" + "Кол-во
массивов: " + repeat + "\t" + "Операций в среднем: " + GetOperationsCount(minElement,
maxElement, repeat, length) / repeat);
            }
            if (name == "Geometric Progression")
            {
                var znamen = double.Parse(node.SelectSingleNode("@Znamen").Value);
                Console.WriteLine("Эксперимент: " + name);
                for (var length = startLength; length <= maxLength; length =
(int)Math.Round(length * znamen))
                    Console.WriteLine("Длина массива: " + length + "\t" + "Кол-во
массивов: " + repeat + "\t" + "Операций в среднем: " + GetOperationsCount(minElement,
maxElement, repeat, length) / repeat);
            }
            Console.WriteLine("\n");
        }
    }
}

```

```

private static int GetOperationsCount(int minElement, int maxElement, int
repeatsCount, int arrayLength) // Получаем количество операций для массивов заданной длины
{
    var operationsCount = 0;
    for (var i = 0; i < repeatsCount; i++)
    {
        var array = new string[arrayLength];
        var random = new Random();
        for (var j = 0; j < array.Length; j++)
            array[j] = random.Next(minElement, maxElement).ToString();
        operationsCount += HeapSort(array);
    }
    return operationsCount;
}

private static bool Test() // Тестовая функция
{
    var isAllTestsCompleted = true;
    {
        var startArray = new[] { "bdg", "avc", "hdf", "aa", "dg", "chd" };
        var expectedArray = new[] { "aa", "avc", "bdg", "chd", "dg", "hdf" };
        HeapSort(startArray);
        if (!Enumerable.SequenceEqual(startArray, expectedArray))
        {
            Console.WriteLine("Test 1 wasn't passed");
            isAllTestsCompleted = false;
        }
    }
    {
        var startArray = new[] { "enfgh", "sgasf", "asdg", "ksd", "asa", "bhj",
"ksgd", "assa", "bhdfj" };
        var expectedArray = new[] { "asa", "asdg", "assa", "bhdfj", "bhj", "enfgh",
"ksd", "ksgd", "sgasf" };
        HeapSort(startArray);
        if (!Enumerable.SequenceEqual(startArray, expectedArray))
        {
            Console.WriteLine("Test 2 wasn't passed");
            isAllTestsCompleted = false;
        }
    }
    {
        var startArray = new[] { "mbdsdfg", "avghssdc", "hjsdf", "abfga", "dhjsg",
"lnvchd" };
        var expectedArray = new[] { "abfga", "avghssdc", "dhjsg", "hjsdf", "lnvchd",
"mbdsdfg" };
        HeapSort(startArray);
        if (!Enumerable.SequenceEqual(startArray, expectedArray))
        {
            Console.WriteLine("Test 3 wasn't passed");
            isAllTestsCompleted = false;
        }
    }
    {
        var startArray = new[] { "ajb", "aja", "agah", "agga", "aswgs", "aswgc",
"aba", "aab" };
        var expectedArray = new[] { "aab", "aba", "agah", "agga", "aja", "ajb",
"aswgc", "aswgs" };
        HeapSort(startArray);
        if (!Enumerable.SequenceEqual(startArray, expectedArray))
        {
            Console.WriteLine("Test 4 wasn't passed");
            isAllTestsCompleted = false;
        }
    }
}

```

```

        {
            var startArray = new[] { "baggage", "auto", "holiday", "bug", "dog", "child"
};
            var expectedArray = new[] { "auto", "baggage", "bug", "child", "dog",
            "holiday" };
            HeapSort(startArray);
            if (!Enumerable.SequenceEqual(startArray, expectedArray))
            {
                Console.WriteLine("Test 5 wasn't passed");
                isAllTestsCompleted = false;
            }
        }
        return isAllTestsCompleted;
    }
}
}

```

### Содержимое xml-файла Experiment:

```

<?xml version="1.0" encoding="utf-8" ?>
<experiments>
    <experiment name= "HeapTree">
        <nodes name = "Arithmetic Progression" minElement = "0" maxElement = "300"
startLength="1000" diff = "100" maxLength = "3000" repeat = "15">
            </nodes>
        <nodes name = "Arithmetic Progression" minElement = "0" maxElement = "1000"
startLength="2000" diff = "100" maxLength = "5000" repeat = "15">
            </nodes>
        <nodes name = "Arithmetic Progression" minElement = "0" maxElement = "10000"
startLength="1000" diff = "100" maxLength = "10000" repeat = "15">
            </nodes>
        <nodes name = "Geometric Progression" minElement = "0" maxElement = "20"
startLength="10" Znamen = "2" maxLength = "10000" repeat = "15">
            </nodes>
        <nodes name = "Geometric Progression" minElement = "0" maxElement = "1000"
startLength="10" Znamen = "5" maxLength = "10000" repeat = "15">
            </nodes>
        </experiment>
    </experiments>

```