

Модуль 4. Алгоритмы и структуры данных

Тема 4.6. Алгоритмы сортировок

2 часа

Оглавление

4.6. Алгоритмы сортировок	2
4.6.1 Сортировки.....	2
Определения.....	2
Оценка эффективности алгоритма сортировки	3
4.6.2. Сортировка вставками (Insertion sort)	5
4.6.3. Быстрая сортировка (Quick sort).....	6
4.6.4 Реализация сортировок в Java.....	9
Сортировка методами классов Arrays и Collections.....	9
Ненатуральный абсолютный порядок.....	12
Упражнение 4.6.1	12
Задание 4.6.1.....	16
Задание 4.6.2.....	16
Благодарности	16

4.6. Алгоритмы сортировок

4.6.1 Сортировки

Пусть требуется собрать матрешку.

В реальном мире для решения этой задачи матрешка должна лежать на контрастном столе в хорошо освещенной комнате, а у человека, ее собирающего, должны быть глаза и минимум одна рука. Тогда решение задачи тривиально: глазами определить самую маленькую часть и поставить на основание, повторить с оставшимися верхними половинами. Важнейшей предпосылкой является возможность оценить размер всех половинок за пренебрежимо малое время.



Однако именно эта предпосылка мешает построению универсального алгоритма: оценить размер всех половинок одновременно не представляется возможным при достаточно большом их количестве. Необходимо сравнивать размер основания и верхней части для каждой куклы в отдельности. Тогда идеальное расположение верхних частей выглядит так:



Для сборки матрешки в этом случае требуется без какого-либо принятия решений помещать на основание самую правую половинку. Расположить половинки таким образом позволяют алгоритмы сортировок.

Сортировка необходима для **эффективной реализации поиска**, прежде всего речь идет о бинарном поиске, который работает только на отсортированной последовательности и значительно быстрее последовательного поиска.

Отсортированная последовательность может быть получена не только путем сортировки изначально неупорядоченной последовательности, но и путем реализации алгоритмов:

- **Вставки с сохранением порядка:** каждый новый элемент вставляется так, чтобы последовательность оставалась отсортированной.
- **Слияния отсортированных последовательностей:** получение нового отсортированного массива, состоящего из элементов двух отсортированных массивов, с минимальными затратами по времени.

Определения

В программировании часто требуется отсортировать набор каких-нибудь объектов, хранящихся в виде списка (List) или массива (int[]). Элементы этого списка или массива могут быть, в том числе, объектами. В этом случае, у каждого элемента нужно определить ключ сортировки - поле или набор полей, по которым набор должен быть отсортирован. Например, если у вас есть список объектов, где каждый объект описывает одного ученика (его класс, возраст, пол, фамилию и имя),

то вы можете отсортировать его разными способами. Если вам нужно отсортировать список по фамилии, значит ключом сортировки является поле “фамилия”, а если вы хотите отсортировать учеников по классам, в которых они учатся, то ключом сортировки должно быть поле “класс”. У вас может быть составной ключ сортировки, например, если вам нужно отсортировать учеников по классам, а внутри каждого класса они должны быть упорядочены по фамилии, то ключом сортировки будет пара полей “класс, фамилия”.

Введем формальные определения для понятий, участвующих в процессе сортировки.

Алгоритм сортировки — это алгоритм для упорядочивания некоторого набора элементов (в списке, массиве и пр.) в соответствии с заданным порядком.

Абсолютный порядок — правило получения результата сравнения двух объектов.

Натуральный порядок — абсолютный порядок, основанный на традиционном отношении «больше-меньше» между элементами счетных множеств таких, как множество целых чисел.

Ключ сортировки — поле (либо набор полей) элемента, служащее критерием порядка при сортировке элементов.

«О» большое — математическое обозначение, которое в применении к алгоритмам используется для выражения его трудоемкости.

В частности в применении к алгоритмам сортировки фраза «трудоемкость алгоритма есть $O(f(n))$ » означает, что с увеличением числа сортируемых элементов n , время работы алгоритма сортировки будет возрастать не быстрее, чем некоторая константа, умноженная на выражение, заключенное в скобках - $f(n)$. Так как умножение на некоторую константу уже подразумевается, то, если $f(n)$ имеет вид $2n$, то, как правило, умножение на 2 опускают и вместо $O(2n)$ пишут $O(n)$.

Оценка эффективности алгоритма сортировки

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

- **Время** — основной параметр, характеризующий быстроедействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в зависимости от числа элементов n в сортируемом множестве n . Для типичного алгоритма хорошее поведение — это $O(n * \log n)$ и плохое поведение — это $O(n^2)$. Идеальное поведение для упорядочения — $O(n)$. Время работы алгоритма зачастую измеряют в количестве операций сравнения, которые необходимы для его реализации.
- **Память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют $O(\log n)$ памяти. При оценке не учитывается место, которое занимает исходный массив и, независимые от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет $O(1)$).



Алгоритмы сортировки характеризуют следующими свойствами:

Устойчивость (англ. *stability*) — устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами.

Естественность поведения — эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Использование операции сравнения. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость худшего случая для этих алгоритмов составляет $O(n * \log n)$, но они отличаются гибкостью применения. Для специальных случаев

(типов данных) существуют более эффективные алгоритмы.

Использование дополнительной памяти. Многие быстрые сортировки используют дополнительную память для хранения промежуточных результатов. Объем дополнительной памяти зависит от размера входных данных.

Использование дополнительных знаний. В особых случаях присутствует дополнительная информация об элементах массива или об их расположении во входных данных. Применение специализированных алгоритмов сортировки может значительно повысить быстродействие, вплоть до $O(n)$. Кусочно-сортированный массив является примером такого случая.

Сортировки подразделяются на две группы по расположению исходных данных:

- **Внутренняя сортировка** оперирует массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте без дополнительных затрат. В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.
- **Внешняя сортировка** оперирует запоминающими устройствами большого объема, но не с произвольным доступом, а последовательным (упорядочение файлов), то есть в данный момент «виден» только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным во внешней памяти производится намного медленнее, чем операции с оперативной памятью. Доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим. Объем данных не позволяет им разместиться в ОЗУ.



Отличная визуализация скорости работы алгоритмов сортировки приведена на сайте известного проекта Algo-rythmics:

<http://algo-rythmics.ms.sapientia.ro/efficiency/selection>.

Можно увидеть сравнение работы 5-ти наиболее известных алгоритмов сортировки:

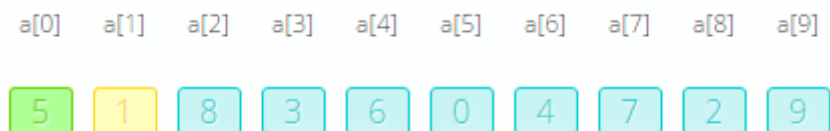
- Insertion sort - сортировка вставками
- Selection sort - сортировка выбором
- Bubble sort - пузырьковая сортировка
- Quick sort - быстрая сортировка
- Shell sort - сортировка Шелла

4.6.2. Сортировка вставками (Insertion sort)

Существует несколько модификаций алгоритма сортировки вставками. Разберем наиболее распространенный вариант, его идея в следующем:

1. Последовательно выбираем все элементы последовательности, начиная со второго. Этот элемент будем называть текущим. Переходим к шагу 2. Если элементы закончились - конец алгоритма.
2. Все элементы до текущего (слева) уже образуют отсортированную последовательность. Берем текущий элемент и последовательно сравниваем с элементами слева. Если элемент больше, то меняемся с ним местами. И так до тех пор, пока не найдем место, где слева от него будет меньший элемент, а справа - больший. В итоге последовательность слева становится больше на 1 и остается отсортированной.
3. Переходим на шаг 1.

Для лучшего понимания работы алгоритма перейдите на сайт проекта Algo-rythmics по адресу <http://algo-rythmics.ms.sapientia.ro/sort/insertion>, В разделе **Insertion sort** можно запустить иллюстрацию в виде танца, выбрав **Dance**, или анимации **Your turn** ⇒ **Start animation** (желтый элемент - это и есть тот самый текущий элемент из описания):



Приведенный ниже код реализует описанный выше алгоритм.

```
public static void insertionSort(int[] arr) {  
    for (int i = 1; i < arr.length; i++) {  
        int currElem = arr[i];  
        int prevKey = i - 1;  
        while (prevKey >= 0 && arr[prevKey] > currElem) {  
            arr[prevKey + 1] = arr[prevKey];  
            arr[prevKey] = currElem;  
            prevKey--;  
        }  
    }  
}
```

Время выполнения алгоритма зависит от входных данных: чем большее множество нужно отсортировать, тем большее время выполняется сортировка. Также на время выполнения влияет исходная упорядоченность массива. Так, лучшим случаем является отсортированный массив, а худшим — массив, отсортированный в порядке, обратном нужному. Временная сложность алгоритма при худшем варианте входных данных — $O(n^2)$. Данный алгоритм можно ускорить при помощи использования бинарного поиска для нахождения места текущему элементу в отсортированной части.

К тому же в массиве скорость теряется из-за проблема с необходимостью сдвига части элементов массива вправо на 1 позицию каждый раз, как происходит вставка текущего элемента в отсортированную часть. Если хранить последовательность не в массиве, а в списке, то этот момент решается путем изменения ссылок (см. рис. ниже).

Иллюстрация преимущества сортировки вставкой в списке на примере одной итерации. На данном шаге - текущий элемент 3. Нашли место вставки - между 1 и 4.



Список после вставки текущего элемента в нужное место:



Мы изменили только три ссылки - выделены красным цветом.

4.6.3. Быстрая сортировка (Quick sort)

Сортировка полностью оправдывает свое название. Она действительно одна из самых эффективных. Сортировка использует стратегию «разделяй и властвуй». Рассмотрим его на примере массива. Этапы алгоритма таковы:

1. **Выбор опорного элемента.** Выбор опорного элемента не влияет на его корректность. А без дополнительных сведений о сортируемых данных невозможно оценить какой выбор лучше. Поэтому применяют самые разнообразные стратегии: выбирать постоянно один и тот же элемент, например, средний или первый по положению; выбирать элемент со случайно выбранным индексом, выбирать среднее арифметическое между минимальным и максимальным элементами массива и т.п.
2. **Разделение массива:** реорганизуем последовательность таким образом, чтобы все элементы со значением меньшим или равным опорному элементу, оказались слева от него, а все элементы, превышающие по значению опорный — справа от него.
3. Для каждого полученного куска последовательности **рекурсивно повторяем шаги 1-2**, до тех пор, пока не получим последовательности, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов. В итоге получаем упорядоченную в процессе разделения общую последовательность.

Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается, по меньшей мере, на единицу обработка гарантированно завершится.

Для лучшего понимания работы алгоритма перейдите на уже знакомый нам сайт проекта Algorithmics по адресу <http://algo-rythmics.ms.sapientia.ro/dance/quick>. В разделе **Insertion Quick sort** запустите танец. Разберем на его примере работу алгоритма:

Этап 1. В качестве опорного элемента берем первый (3).

Этап 2. Разделение. Сравниваем опорный элемент (3) с последним (6).



Поочередно сравниваем опорный элемент с элементами начиная с конца до тех пор, пока элементы больше опорного. Как только нашли элемент меньше опорного (2), меняем его местами с опорным (3). В итоге 3 встает между 7 и 5, а 2 - встает на первое место.



Далее начинаем сравнение опорного элемента (3) с элементами слева. Как только встречаем элемент, больше опорного (а слева, как мы знаем, все элементы должны быть меньше опорного) - происходит обмен (3 меняем с 8):



И снова начинаем сравнение с элементами справа от опорного, которые лежат между переставленными 3 и 8. Между ними остался только элемент 7. А так как, он больше опорного, то обмена не происходит и этап разделения заканчивается.

Этап 3. Для каждого отрезка справа и слева от опорного рекурсивно выполняем этапы 1-2. И так до тех пор, пока все отрезки не сузятся до 2 или 1 элемента.



Как уже было сказано, существует множество различных вариантов реализации быстрой сортировки. Здесь можно посмотреть некоторые из них на различных языках.

https://ru.wikibooks.org/wiki/Реализации_алгоритмов/Сортировка/Быстрая

Вот один из вариантов:

```
private static int partition(int[] array, int start, int end) {
    int marker = start;
    for (int i = start; i <= end; i++) {
        if (array[i] <= array[end]) {
            int temp = array[marker]; // перестановка
            array[marker] = array[i];
            array[i] = temp;
            marker += 1;
        }
    }
    return marker - 1;
}

public static void quickSort(int[] array, int start, int end) {
    if (start >= end)
        return;
    int pivot = partition(array, start, end);
    quickSort(array, start, pivot - 1);
    quickSort(array, pivot + 1, end);
}
```

Ясно, что операция разделения массива на две части относительно опорного элемента занимает время $O(n)$. Поскольку все операции разделения, выполняемые на одной глубине рекурсии, обрабатывают разные части исходного массива, размер которого постоянен, суммарно на каждом уровне рекурсии потребуется также $O(n)$ операций. Следовательно, общая сложность алгоритма определяется лишь количеством разделений, то есть глубиной рекурсии. Глубина рекурсии, в свою очередь, зависит от сочетания входных данных и способа определения опорного элемента. Не будем здесь углубляться в вычисления, приведем итоговые оценки:

- В наилучшем случае, когда последовательность хорошо перемешана и каждое деление дает примерно одинаковые по размеру отрезки средняя сложность составит $O(n \cdot \log_2 n)$.
- В худшем случае при самом несбалансированном варианте общее время работы составит $O(n^2)$. Для больших значений n худший случай может привести к исчерпанию памяти во время работы программы (переполнению стека вызовов функций из-за рекурсии).

Помимо разобранных нами двух алгоритмов сортировки известно множество других, причем у каждого алгоритма еще могут быть свои модификации.

4.6.4 Реализация сортировок в Java

На практике в большинстве задач на Java нет никакой необходимости разрабатывать свои алгоритмы сортировки, достаточно использовать стандартные методы, которые предлагает богатая библиотека Java.

Сортировка методами классов `Arrays` и `Collections`

Методы сортировки класса `Arrays`:

<code>static void sort(int[] a)</code>	Метод, сортирующий массив <code>a</code> в натуральном порядке.
<code>static void sort(int[] a, int fromIndex, int toIndex)</code>	Метод, сортирующий часть массива <code>a</code> от <code>fromIndex</code> до <code>toIndex</code> в натуральном порядке.
<code>static <T> void sort(T[] a, Comparator<? super T> c)</code>	Метод, сортирующий массив <code>a</code> в порядке, заданном объектом с интерфейсом <code>Comparator</code> .
<code>static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)</code>	Метод, сортирующий часть массива <code>a</code> от <code>fromIndex</code> до <code>toIndex</code> в порядке, заданном объектом с интерфейсом <code>Comparator</code> .

Первые два метода перегружены для всех простых типов и класса `Object`. Важно помнить, что требование реализации интерфейса `Comparable` не указано в явном виде в сигнатуре метода `static void sort(Object[] a)`.

Методы сортировки класса `Collections`:

<code>static <T extends Comparable<? super T>> void sort(List<T> list)</code>	Метод, сортирующий список <code>list</code> в натуральном порядке.
<code>static <T> void sort(List<T> list, Comparator<? super T> c)</code>	Метод, сортирующий список <code>list</code> в порядке, заданном объектом с интерфейсом <code>Comparator</code> .

Все сортировки устойчивы, но их алгоритмы не закреплены в стандарте языка Java. Реализация сортировок производится разработчиками виртуальной машины Java и может отличаться на разных платформах. Например, в JVM для Java 8 метод `sort` класса `Arrays` реализован на основе Dual-Pivot Quicksort алгоритма. Это модификация классического алгоритма Quicksort, которая на многих данных показывает результат в $O(n \cdot \log_2 n)$, где обычный Quicksort выдает $O(n^2)$.



В Java 8 в классе `Arrays` появился метод `parallelSort` для более быстрой сортировки больших объемов данных. Стоит отметить, что если входной массив будет недостаточно большой, то сортировка будет производиться в одном потоке. Это связано с накладными расходами для создания новых потоков.

Ниже приведена таблица продолжительности работы алгоритмов обычной и параллельной сортировки в зависимости от количества данных.

Number of element	Serial(sec)	Parallel(sec)
602	0.001	0.004
1400	0.004	0.004
2492	0.007	0.007
4984	0.025	0.008
9968	0.027	0.011
19936	0.064	0.015
39872	0.101	0.024
79744	0.135	0.04
159488	0.253	0.077
318976	0.419	0.152
637952	0.501	0.253
1275904	0.588	0.248
2551808	0.821	0.344
5103616	1.578	0.686

Сортировка простых типов с помощью методов класса `Arrays` и `Collections`

Для простых типов вполне понятно, что означает натуральный порядок, поэтому в Java методы `sort` полностью готовы и их можно применять к контейнерам таких типов без дополнительной подготовки.

Пример сортировки массива простых типов через `Arrays.sort`:

```
import java.util.Arrays;

public class ArraySortTest {

    public static void main(String[] args) {
        int[] array = { 15, 42, 4, 23, 16, 8 };
        System.out.println(Arrays.toString(array));
        Arrays.sort(array);
        System.out.println(Arrays.toString(array));
    }
}
```

Пример сортировки списка простых типов через `Collections.sort`:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ListSortTest {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        for (int c : new int[] { 15, 42, 4, 23, 16, 8 })
            list.add(c);
    }
}
```

```
        System.out.println(list.toString());
        Collections.sort(list);
        System.out.println(list.toString());
    }
}
```

Для хранения простых типов в списках необходимо использовать их классы-обертки.

Сортировка объектов

Заранее определить, что понимать под натуральным порядком для любых объектов невозможно, поэтому в Java ответственность за его определение ложится на разработчика класса в форме требования “реализовать интерфейс `Comparable`”.

Интерфейс `Comparable` включает в себя всего один метод:

<code>int compareTo(T o)</code>	Метод, возвращающий результат сравнения текущего объекта с объектом <code>o</code> . Возвращаемое значение: <ul style="list-style-type: none">• Отрицательное, если <code>this < o</code>• Ноль, если <code>this == o</code>• Положительное, если <code>this > o</code>
---------------------------------	---

Например, определение натурального порядка для объектов класса `Rectangle` может выглядеть так:

```
class Rectangle implements Comparable<Rectangle> {
    double w, h;

    public Rectangle(double w, double h) {
        super();
        this.w = w;
        this.h = h;
    }

    public double area() {
        return w * h;
    }

    @Override
    public int compareTo(Rectangle arg0) {
        return (int) (this.area() - arg0.area());
    }

    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Rectangle) {
            Rectangle that = (Rectangle) other;
            result = (this.w == that.w && this.h == that.h);
        }
        return result;
    }
}
```

Таким образом, прямоугольники сравниваются по площади с точностью до 1 квадратной единицы.

С точки зрения применения методов `sort`, контейнеры объектов не имеют отличий от контейнеров простых типов.

Обратите внимание, хоть и метод `equals` не входит в `Comparable`, его реализация является рекомендуемой. Т.к. если методы `equals` и `compareTo` будут вести себя не согласованно, то могут появиться неожиданные неприятности с хранением данных в коллекциях (`SortedSet`, `SortedMap` и др.).

Ненатуральный абсолютный порядок

Для сортировки некоторых объектов может существовать больше, чем один способ для их упорядочивания. Кроме того, не всегда легко определить, какой порядок является натуральным. В этом случае, можно реализовать несколько различных компараторов (“сравнителей”), каждый из которых определял бы нужный порядок, и использовать в каждом случае свой компаратор. Основной метод интерфейса `Comparator` определен следующим образом:

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Он служит для сравнения двух объектов `o1` и `o2` и, аналогично методу `compareTo` интерфейса `Comparable`, должен возвращать отрицательное число, если `o1` меньше, чем `o2`, ноль, если `o1` равен `o2` и положительное число, если `o1` больше `o2`.

Упражнение 4.6.1

Рассмотрим простой класс, который хранит данные о человеке:

```
/**  
 * Класс, описывающий конкретного человека. Не позволяет менять содержимое полей  
 * после создания.  
 */  
public class Person {  
    private final String name;  
    private final String surname;  
    private final int age;  
  
    /**  
     * @return возраст этого человека  
     */  
    public int getAge() {  
        return age;  
    }  
  
    /**  
     * @return имя этого человека  
     */  
    public String getName() {  
        return name;  
    }  
  
    /**  
     * @return фамилия этого человека  
     */  
    public String getSurname() {  
        return surname;  
    }  
}
```

```

    }

    @Override
    public String toString() {
        return "Person{" +
            "age=" + age +
            ", name='" + name + '\'' +
            ", surname='" + surname + '\'' +
            '}';
    }

    /**
     * Позволяет создать нового человека
     * @param age возраст
     * @param name имя
     * @param surname фамилия
     */
    public Person(int age, String name, String surname) {
        this.age = age;
        this.name = name;
        this.surname = surname;
    }
}

```

Предположим, в программе, в зависимости от выбора пользователя, нужно упорядочивать людей в зависимости от выбора пользователя или по именам, или по фамилиям и именам (иначе говоря, все люди упорядочены по фамилиям, а однофамильцы между собой - по именам), или по возрасту и фамилиям и именам (иначе говоря, все упорядочены по возрасту, те, у кого возраст одинаков, упорядочены между собой по фамилиям, а однофамильцы, у которых возраст одинаков, упорядочены между собой по именам).

Это может быть реализовано следующим образом (описание полей, геттеров, конструктора и toString опущено, так как они были приведены выше):

```

public class Person {

    /**
     * Экземпляр компаратора, который сравнивает двух людей по именам
     */
    public static final Comparator<Person> NAME_COMPARATOR =
        new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                // возвращаем результат сравнения имен
                return o1.getName().compareTo(o2.getName());
            }
        };

    /**
     * Экземпляр компаратора, который сравнивает двух людей по фамилиям,
     * а в случае равенства фамилий - по именам
     */
    public static final Comparator<Person> SURNAME_NAME_COMPARATOR =
        new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {

```



```

        // сравним фамилии
        int result = o1.getSurname().compareTo(o2.getSurname());
        // если фамилии не одинаковы - вернем результат сравнения
        if (result != 0)
            return result;
        // для одинаковых фамилий, результат сравнения - сравнение имен
        return o1.getName().compareTo(o2.getName());
    }
};

/**
 * Экземпляр компаратора, который сравнивает двух людей по возрасту,
 * в случае равенства возрастов - по фамилиям, а в случае равенства
 * фамилий - по именам
 */
public static final Comparator<Person> AGE_SURNAME_NAME_COMPARATOR =
    new Comparator<Person>() {
        @Override
        public int compare(Person o1, Person o2) {
            // сравним возраст
            int result = Integer.compare(o1.getAge(), o2.getAge());
            // если возраст не одинаков - вернем результат сравнения
            if (result != 0)
                return result;

            // сравним фамилии
            result = o1.getSurname().compareTo(o2.getSurname());
            // если фамилии не одинаковы - вернем результат сравнения
            if (result != 0)
                return result;

            // для одинаковых возрастов и фамилий, результат
            // сравнения - сравнение имен
            return o1.getName().compareTo(o2.getName());
        }
    };
}

```

В этом коде определены три компаратора, которые доступны через статические поля класса Person. Они позволяют упорядочить список людей в том виде, в котором он нам необходим. Приведем пример использования этих компараторов:

```

public static void main(String[] args) {
    List<Person> people = new ArrayList<Person>();
    people.add(new Person(22, "Eikichi", "Onizuka"));
    people.add(new Person(72, "Edsger", "Dijkstra"));
    people.add(new Person(41, "Alan", "Turing"));
    people.add(new Person(41, "Sergey", "Brin"));
    people.add(new Person(41, "Impersonator", "Brin"));
    people.add(new Person(28, "Dmitriy", "Karamazov"));
    people.add(new Person(23, "Ivan", "Karamazov"));
    people.add(new Person(16, "Alex", "Karamazov"));

    // отсортируем людей по именам
    Collections.sort(people, Person.NAME_COMPARATOR);
    // выведем их
}

```

```
System.out.println("By name:");
for (Person p: people)
    System.out.println(p);
System.out.println();

// отсортируем людей по фамилиям, а затем по именам
Collections.sort(people, Person.SURNAME_NAME_COMPARATOR);
// выведем их
System.out.println("By surname and then by name:");
for (Person p: people)
    System.out.println(p);
System.out.println();

// отсортируем людей по возрасту, а затем по фамилиям, а затем по именам
Collections.sort(people, Person.AGE_SURNAME_NAME_COMPARATOR);
// выведем их
System.out.println("By age and then by surname and then by name:");
for (Person p: people)
    System.out.println(p);
}
```

Результат работы этого примера следующий:

```
By name:
Person{age=41, name='Alan', surname='Turing'}
Person{age=16, name='Alex', surname='Karamazov'}
Person{age=28, name='Dmitriy', surname='Karamazov'}
Person{age=72, name='Edsger', surname='Dijkstra'}
Person{age=22, name='Eikichi', surname='Onizuka'}
Person{age=41, name='Impersonator', surname='Brin'}
Person{age=23, name='Ivan', surname='Karamazov'}
Person{age=41, name='Sergey', surname='Brin'}

By surname and then by name:
Person{age=41, name='Impersonator', surname='Brin'}
Person{age=41, name='Sergey', surname='Brin'}
Person{age=72, name='Edsger', surname='Dijkstra'}
Person{age=16, name='Alex', surname='Karamazov'}
Person{age=28, name='Dmitriy', surname='Karamazov'}
Person{age=23, name='Ivan', surname='Karamazov'}
Person{age=22, name='Eikichi', surname='Onizuka'}
Person{age=41, name='Alan', surname='Turing'}

By age and then by surname and then by name:
Person{age=16, name='Alex', surname='Karamazov'}
Person{age=22, name='Eikichi', surname='Onizuka'}
Person{age=23, name='Ivan', surname='Karamazov'}
Person{age=28, name='Dmitriy', surname='Karamazov'}
Person{age=41, name='Impersonator', surname='Brin'}
Person{age=41, name='Sergey', surname='Brin'}
Person{age=41, name='Alan', surname='Turing'}
Person{age=72, name='Edsger', surname='Dijkstra'}
```

Практически никогда нет необходимости создавать больше одного экземпляра каждого компаратора, поэтому классы-компараторы объявлены как приватные статические классы внутри класса Person и их единственные экземпляры доступны через статические переменные.

Стоит отметить, что возможность дать переменной-компаратору имя существенно улучшает понятность вашего кода как для вас самих, так и для других людей, которые будут его читать. Предположим, что вы встретили в чужом коде строку:

```
Collections.sort(people);
```

У вас нет никакой информации о том, как именно будет отсортирован список в данном случае. Хорошо, если вы можете заглянуть в исходные коды класса `Person` и узнать, как был реализован метод `compareTo`, и выяснить, какой порядок сравнения был принят за натуральный, но это требует от вас дополнительных действий каждый раз, когда вы встречаете подобную строку. Если же вы встречаете строку

```
Collections.sort(people, Person.NAME_COMPARATOR);
```

то вам не нужно производить дополнительных действий, вы сразу видите, что здесь автор подразумевал упорядочивание людей по их именам. Когда вы имеете дело не с объектами типа строк или чисел, для которых есть очевидный всем естественный порядок, часто имеет смысл создать компаратор просто для того, чтобы дать ему имя и повысить понятность кода.

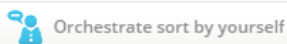
Задание 4.6.1


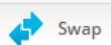
Реализовать функцию, которая проверяет, является ли массив отсортированным.

Задание 4.6.2

На сайте проекта Algo-rythmics (<http://algo-rythmics.ms.sapientia.ro/>) в разделах **Your turn** для сортировки вставками и быстрой сортировки выполните предлагаемое интерактивное

упражнение, нажав на кнопку:



В задании необходимо вручную выполнить шаги сортировки, поочередно выделяя элементы и задавая операцию: сравнение  или перестановку 

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателям ИТ ШКОЛЫ SAMSUNG Лянгузову Федору Андреевичу, Козловскому Павлу Александровичу, Мансурову Руслану Маратовичу.