**Алгоритмы сортировки**

1. ***Простейшая сортировка (Bubble Sort)***

Вот наша первая сортировка и отработала. Мы итерируемся во внешнем цикле (while) до тех пор, пока не решим, что итераций больше не нужно. По умолчанию перед каждой новой итерацией мы допускаем, что наш массив отсортирован, и больше итерироваться не хотим. Поэтому, мы проходим элементы последовательно и проверяем это допущение. Но если элементы не по порядку, мы выполняем swap элементов и понимаем, что нет уверенности, что теперь элементы в правильном порядке. Следовательно, хотим выполнить ещё одну итерацию. Например, [3, 5, 2]. 5 больше трёх, всё хорошо. Но 2 меньше 5. Однако [3, 2, 5] требует ещё одного прохода, т.к. 3 > 2 и их нужно поменять местами.

Сортировка пузырьком — одна из самых простых и неэффективных сортировок. Её ещё иногда называют "глупой сортировкой".

**Пример кода:**

**public** **static** **void** bubbleSort() {

**int**[] array = {10, 2, 10, 3, 1, 2, 5};

System.***out***.println(Arrays.*toString*(array));

**boolean** needIteration = **true**;

**while** (needIteration) {

needIteration = **false**;

**for** (**int** i = 1; i < array.length; i++) {

**if** (array[i] < array[i - 1]) {

*swap*(array, i, i-1);

needIteration = **true**;

}

}

}

System.***out***.println(Arrays.*toString*(array));

}

**private** **static** **void** swap(**int**[] array, **int** ind1, **int** ind2) {

**int** tmp = array[ind1];

array[ind1] = array[ind2];

array[ind2] = tmp;

}

1. ***Сортировка выбором (Selection Sort)***

Идея простая. Каждый проход выбирать самый минимальный элемент и смещать его в начало. При этом каждый новый проход начинать сдвигаясь вправо, то есть первый проход — с первого элемента, второй проход — со второго.

Данная сортировка неустойчива, т.к. одинаковые элементы (с точки зрения той характеристики, по которой мы сортируем элементы) могут изменить своё положение.

**Пример кода:**

**public** **static** **void** selectionSort() {

**int**[] array = { 10, 2, 10, 3, 1, 2, 5 };

System.***out***.println(Arrays.*toString*(array));

**for** (**int** left = 0; left < array.length; left++) {

**int** minInd = left;

**for** (**int** i = left; i < array.length; i++) {

**if** (array[i] < array[minInd]) {

minInd = i;

}

}

**if** (left != minInd) {

*swap*(array, left, minInd);

}

}

System.***out***.println(Arrays.*toString*(array));

}

**private** **static** **void** swap(**int**[] array, **int** ind1, **int** ind2) {

**int** tmp = array[ind1];

array[ind1] = array[ind2];

array[ind2] = tmp;

}

1. ***Сортировка вставками (Insertion Sort)***

Сортировка вставками тоже имеет квадратичную сложность, так как у нас опять цикл в цикле. В чём отличие от сортировки выбором? Данная сортировка является "устойчивой". Это значит, что одинаковые элементы не изменят свой порядок. Одинаковые с точки зрения характеристики, по которой мы сортируем.

**Пример кода:**

**public** **static** **void** insertionSort() {

**int**[] array = { 10, 2, 10, 3, 1, 2, 5 };

System.***out***.println(Arrays.*toString*(array));

**for** (**int** left = 0; left < array.length; left++) {

// Вытаскиваем значение элемента

**int** value = array[left];

// Перемещаемся по элементам, которые перед вытащенным элементом

**int** i = left - 1;

**for** (; i >= 0; i--) {

// Если вытащили значение меньшее — передвигаем больший элемент дальше

**if** (value < array[i]) {

array[i + 1] = array[i];

} **else** {

// Если вытащенный элемент больше — останавливаемся

**break**;

}

}

// В освободившееся место вставляем вытащенное значение

array[i + 1] = value;

}

System.***out***.println(Arrays.*toString*(array));

}

1. ***Челночная сортировка (Shuttle Sort)***

Среди простых сортировок есть ещё одна — челночная сортировка. Но мне больше нравится шаттл сорт. Мне кажется, мы редко говорим про космические челноки, а челночный у нас скорее бег. Поэтому проще представить, как в космос запускаются шаттлы. Вот вам ассоциация с этим алгоритмом.

В чём суть алгоритма? Суть алгоритма в том, что мы итерируемся слева направо, при этом при выполнении swap элементов мы выполняем проверку всех остальных элементов, которые остались позади, не нужно ли повторить swap.

**Пример кода:**

**public** **static** **void** shuttleSort() {

**int**[] array = { 10, 2, 10, 3, 1, 2, 5 };

System.***out***.println(Arrays.*toString*(array));

**for** (**int** i = 1; i < array.length; i++) {

**if** (array[i] < array[i - 1]) {

*swap*(array, i, i - 1);

**for** (**int** z = i - 1; (z - 1) >= 0; z--) {

**if** (array[z] < array[z - 1]) {

*swap*(array, z, z - 1);

} **else** {

**break**;

}

}

}

}

System.***out***.println(Arrays.*toString*(array));

}

**private** **static** **void** swap(**int**[] array, **int** ind1, **int** ind2) {

**int** tmp = array[ind1];

array[ind1] = array[ind2];

array[ind2] = tmp;

}

1. ***Сортировка Шелла***

Ещё одной простой сортировкой является сортировка Шелла.

Суть её похожа на сортировку пузырьком, но каждую итерацию мы имеем разный промежуток между сравниваемыми элементами. Каждую итерацию он уменьшается вдвое.

**Пример кода:**

**public** **static** **void** shellaSort() {

**int**[] array = {10, 2, 10, 3, 1, 2, 5};

System.***out***.println(Arrays.*toString*(array));

// Высчитываем промежуток между проверяемыми элементами

**int** gap = array.length / 2;

// Пока разница между элементами есть

**while** (gap >= 1) {

**for** (**int** right = 0; right < array.length; right++) {

// Смещаем правый указатель, пока не сможем найти такой, что

// между ним и элементом до него не будет нужного промежутка

**for** (**int** c = right - gap; c >= 0; c -= gap) {

**if** (array[c] > array[c + gap]) {

*swap*(array, c, c + gap);

}

}

}

// Пересчитываем разрыв

gap = gap / 2;

}

System.***out***.println(Arrays.*toString*(array));

}

**private** **static** **void** swap(**int**[] array, **int** ind1, **int** ind2) {

**int** tmp = array[ind1];

array[ind1] = array[ind2];

array[ind2] = tmp;

}

1. ***Cортировка слиянием (merge sort)***

Помимо указанных простых сортировок, есть сортировки и посложнее. Например, сортировка слиянием.

Во-первых, нам на помощь придёт [рекурсия](https://habr.com/post/275813/).

Во-вторых, сложность у нас будет уже не квадратичная, как мы с вами привыкли. Сложность данного алгоритма — логарифмическая. Записывается как O(n log n).

Сначала, напишем рекурсивный вызов метода сортировки:

**public** **static** **void** mergeSort(**int**[] source, **int** left, **int** right) {

// Выберем разделитель, т.е. разделим пополам входной массив

**int** delimiter = left + ((right - left) / 2) + 1;

// Выполним рекурсивно данную функцию для двух половинок (если сможем разбить(

**if** (delimiter > 0 && right > (left + 1)) {

*mergeSort*(source, left, delimiter - 1);

*mergeSort*(source, delimiter, right);

}

}

Теперь, давайте к нему добавим основное действие. Вот пример нашего суперметода с реализацией:

**public** **static** **void** mergeSort(**int**[] source, **int** left, **int** right) {

// Выберем разделитель, т.е. разделим пополам входной массив

**int** delimiter = left + ((right - left) / 2) + 1;

// Выполним рекурсивно данную функцию для двух половинок (если сможем разбить(

**if** (delimiter > 0 && right > (left + 1)) {

*mergeSort*(source, left, delimiter - 1);

*mergeSort*(source, delimiter, right);

}

// Создаём временный массив с нужным размером

**int**[] buffer = **new** **int**[right - left + 1];

// Начиная от указанной левой границы идём по каждому элементу

**int** cursor = left;

**for** (**int** i = 0; i < buffer.length; i++) {

// Мы используем delimeter чтобы указывать на элемент из правой части

// Если delimeter > right, значит в правой части не осталось недобавленных

// элементов

**if** (delimiter > right || source[cursor] > source[delimiter]) {

buffer[i] = source[cursor];

cursor++;

} **else** {

buffer[i] = source[delimiter];

delimiter++;

}

}

System.*arraycopy*(buffer, 0, source, left, buffer.length);

}

Запустим пример при помощи вызова метода mergeSort(array, 0, array.length-1). Как видно, суть сводится к тому, что мы принимаем на вход массив с указанием начала и конца участка для сортировки. При начале сортировки — это начало и конец массива. Далее мы вычисляем delimiter — положение делителя. Если делитель может разделить на 2 части, значит вызываем рекурсивно сортировку для участков, на которые делитель разбил массив. Подготавливаем дополнительный буферный массив, в котором выделяем отсортированный участок. Далее устанавливаем курсор в начало сортируемого участка и начинаем идти по каждому элементу пустого массива, который мы подготовили, и заполняем его наименьшими элементами. Если элемент, на который указывает курсор меньше, чем элемент, на который указывает делитель — помещаем в буферный массив этот элемент и сдвигаем курсор. В противном случае помещаем в буферный массив элемент, на который указывает разделитель и сдвигаем разделитель. Как только разделитель уйдёт за границы сортируемого участка или мы заполним весь массив, указанный диапазон считается отсортированным.

1. ***Сортировка подсчётом (Counting Sort) и Поразрядная сортировка (Radix Sort)***

Другим интересным алгоритмом сортировки является сортировка подсчётом (Counting Sort). Алгоритмическая сложность в этом случае будет O(n+k), где n — количество элементов, а k — максимальное значение элемента. Есть с алгоритмом одна незадача: нам нужно знать минимальное и максимальное значение в массиве.

Вот пример реализации сортировки подсчётом:

**public** **static** **int**[] countingSort(**int**[] theArray, **int** maxValue) {

// Массив со "счётчиками" размером от 0 до максимального значения

**int** numCounts[] = **new** **int**[maxValue + 1];

// В соответствующей ячейке (индекс = значение) увеличиваем счётчик

**for** (**int** num : theArray) {

numCounts[num]++;

}

// Подготавливаем массив для отсортированного результата

**int**[] sortedArray = **new** **int**[theArray.length];

**int** currentSortedIndex = 0;

// идём по массиву со "счётчиками"

**for** (**int** n = 0; n < numCounts.length; n++) {

**int** count = numCounts[n];

// идём по количеству значений

**for** (**int** k = 0; k < count; k++) {

sortedArray[currentSortedIndex] = n;

currentSortedIndex++;

}

}

**return** sortedArray;

}

Как мы понимаем, очень неудобно, когда мы должны знать заранее минимальное и максимальное значение. И тут есть другой алгоритм — Radix Sort.

1. **Быстрая сортировка (Quick Sort)**

Ну и на сладкое — один из самых известных алгоритмов: быстрая сортировка. Она имеет алгоритмическую сложность, то есть мы имеем O(n log n). Данную сортировку ещё называют сортировкой Хоара. Интересно, что алгоритм был придуман Хоаром во время его пребывания в Советском Союзе, где он обучался в Московском университете компьютерному переводу и занимался разработкой русско-английского разговорника. А ещё данный алгоритм в более сложной реализации используется в Arrays.sort в Java. А что с Collections.sort? Предлагаю посмотреть самостоятельно, как сортируются они "под капотом".

**Итак, код:**

**public** **static** **void** quickSort(**int**[] source, **int** leftBorder, **int** rightBorder) {

**int** leftMarker = leftBorder;

**int** rightMarker = rightBorder;

**int** pivot = source[(leftMarker + rightMarker) / 2];

**do** {

// Двигаем левый маркер слева направо пока элемент меньше, чем pivot

**while** (source[leftMarker] < pivot) {

leftMarker++;

}

// Двигаем правый маркер, пока элемент больше, чем pivot

**while** (source[rightMarker] > pivot) {

rightMarker--;

}

// Проверим, не нужно обменять местами элементы, на которые указывают маркеры

**if** (leftMarker <= rightMarker) {

// Левый маркер будет меньше правого только если мы должны выполнить swap

**if** (leftMarker < rightMarker) {

**int** tmp = source[leftMarker];

source[leftMarker] = source[rightMarker];

source[rightMarker] = tmp;

}

// Сдвигаем маркеры, чтобы получить новые границы

leftMarker++;

rightMarker--;

}

} **while** (leftMarker <= rightMarker);

// Выполняем рекурсивно для частей

**if** (leftMarker < rightBorder) {

*quickSort*(source, leftMarker, rightBorder);

}

**if** (leftBorder < rightMarker) {

*quickSort*(source, leftBorder, rightMarker);

}

}

Тут всё очень страшно, так что будем разбираться.

Для входного массива **int[]** source выставляем два маркера, левый (L) и правый (R). При первом вызове они соответствуют началу и концу массива.

Далее определяется опорный элемент, он же **pivot**. После этого наша задача — переместить значения, меньшие чем **pivot**, в левую от **pivot** часть, а большие — в правую.

Для этого сначала двигаем указатель **L**, пока не найдём значение, большее чем **pivot**. Если меньше значения не нашли, то L совпадёт с **pivot**.

Потом двигаем указатель **R** пока не найдём меньшее, чем **pivot** значение. Если меньшее значение не нашли, то **R** совпадёт с **pivot**.

Далее, если указатель **L** находится до указателя **R** или совпадает с ним, то пытаемся выполнить обмен элементов, если элемент **L** меньше, чем **R**.

Далее **L** сдвигаем вправо на 1 позицию, **R** сдвигаем влево на одну позицию.

Когда левый маркер **L** окажется за правым маркером **R** это будет означать, что обмен закончен, слева от **pivot** меньшие значения, справа от **pivot** — большие значения.

После этого рекурсивно вызываем такую же сортировку для участков массива от начала сортируемого участка до правого маркера и от левого маркера до конца сортируемого участка.

Почему от начала до правого? Потому что в конце итерации так и получится, что правый маркер сдвинется настолько, что станет границей части слева.

Этот алгоритм более сложный, чем простая сортировка, поэтому его лучше зарисовать. Возьмём белый лист бумаги, запишем: 4 2 6 7 3, а **Pivot** по центру, т.е. число 6. Обведём его в круг.

Под 4 напишем **L**, под 3 напишем **R**. 4 меньше чем 6, 2 меньше чем 6. Итого, **L** переместился на положение **pivot**, т.к. по условию **L** не может уйти дальше, чем **pivot**.

Напишем снова 4 2 6 7 3, обведём 6 вкруг (**pivot**) и поставим под ним **L**. Теперь двигаем указатель **R**.

3 меньше чем 6, поэтому ставим маркер **R** на цифру 3. Так как 3 меньше, чем **pivot 6** выполняем **swap**, т.е. обмен.

Запишем результат: 4 2 3 7 6, обводим 6 вкруг, т.к. он по прежнему **pivot**.

Указатель **L** на цифре 3, указатель **R** на цифре 6. Мы помним, что двигаем указатели до тех пор, пока **L** не зайдём за **R**. **L** двигаем на следующую цифру.

Тут хочется разобрать два варианта: если бы предпоследняя цифра была 7 и если бы она была не 7, а 1.

**Предпоследня цифра 1:** Сдвинули указатель **L** на цифру 1, т.к. мы можем двигать **L** до тех пор, пока указатель **L** указывает на цифру, меньшую чем **pivot**. А вот **R** мы не можем сдвинуть с 6, т.к. **R** не мы можем двигать только если указатель **R** указывает на цифру, которая больше чем **pivot**. **swap** не делаем, т.к. 1 меньше 6. Записываем положение: 4 2 3 1 6, обводим **pivot** 6. **L** сдвигается на **pivot** и больше не двигается. **R** тоже не двигается. Обмен не производим. Сдвигаем **L** и **R** на одну позицию и подписываем цифру 1 маркером **R**, а **L** получается вне числа. Т.к. **L** вне числа — ничего не делаем, а вот часть 4 2 3 1 выписываем снова, т.к. это наша левая часть, меньшая, чем **pivot** 6. Выделяем новый **pivot** и начинаем всё снова).

**Предпоследняя цифра 7:** Сдвинули указать **L** на цифру 7, правый указатель не можем двигать, т.к. он уже указывает на **pivot**. т.к. 7 больше, чем **pivot**, то делаем **swap**. Запишем результат: 4 2 3 6 7, обводим 6 кружком, т.к. он **pivot**. Указатель **L** теперь сдвигается на цифру 7, а указатель **R** сдвигается на цифру 3. Часть от **L** до конца нет смысла сортировать, т.к. там всего 1 элемент, а вот часть от 4 до указателя **R** отправляем на сортировку. Выбираем **pivot** и начинаем всё снова).

Может на первый взгляд показаться, что если расставить много одинаковых с **pivot** значений, это сломает алгоритм, но это не так. Можно напридумывать каверзных вариантов и на бумажке убедиться, что всё правильно и подивиться, как такие простые действия предоставляют такой надёжный механизм. Единственный минус — такая сортировка не является стабильной. Т.к. при выполнении обмена одинаковые элементы могут поменять свой порядок, если один из них встретился до **pivot** до того, как другой элемент попал в часть до **pivot** при помощи обмена.