SORT Quick Merge Insert

<https://dev64.wordpress.com/2013/07/24/quick-sort/>

# Быстрая сортировка (Quicksort)

### [Галерея](https://dev64.wordpress.com/type/gallery/)

Posted on [Июль 24, 2013](https://dev64.wordpress.com/2013/07/24/quick-sort/)

Продолжаю просмотр курса «Алгоритмы и структуры данных» от MIT: <http://videolectures.net/mit6046jf05_leiserson_lec04/> Сегодня quicksort.

**Несколько дополнительных ссылок**

<http://www.algolist.net/Algorithms/Sorting/Quicksort>  
<http://algolist.manual.ru/sort/quick_sort.php>  
[http://ru.wikipedia.org/wiki/Быстрая\_сортировка](http://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)

Быстрая сортировка заключается в рекурсивном делении сортируемого массива данных на 2 части: большую либо равную значению некоторого элемента, называемого pivot, и меньшую либо равную, чем pivot.

[quick-sort](https://dev64.files.wordpress.com/2013/07/quick-sort.png)

Ниже исходный код на java, демонстрирующий реализацию быстрой сортировки.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // @see <http://www.algolist.net/Algorithms/Sorting/Quicksort>  void quickSort(int arr[], int left, int right) {      int index = partition(arr, left, right);      if (left < index - 1)          quickSort(arr, left, index - 1);      if (index < right)          quickSort(arr, index, right);  } |

Основная работа выполняется в функции partition. Эта функция выполняет в один проход по массиву разбиение массива на 2 части (как показано на картинке выше).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // @see <http://www.algolist.net/Algorithms/Sorting/Quicksort>  int partition(int arr[], int left, int right)  {      int i = left, j = right;      int tmp;      int pivot = arr[(left + right) / 2];        while (i <= j) {          while (arr[i] < pivot)              i++;          while (arr[j] > pivot)              j--;            if (i <= j) {              tmp = arr[i];              arr[i] = arr[j];              arr[j] = tmp;              i++;              j--;          }      }        return i;  } |

В данном случае в качестве pivot берется элемент из середины массива. Скорость работы быстрой сортировки зависит от значения pivot, выбираемого на каждом шаге. Чтобы глубина рекурсии была наименьшей, нужно чтобы pivot попадал в медианное значение. Тогда на каждом шаге сортировки массив будет делиться на 2 примерно равные части. Это приводит к сложности алгоритма O(n \* lg n). В худшем случае, если на каждом шаге в качестве pivot выберется наибольшее или наименьшее значение, тогда глубина рекурсии будет равна n, соответственно будет n рекурсивных проходов по всему массиву размером n. Отсюда сложность в худшем случае достигает O(n^2). В среднем сложность алгоритма O(n \* lg n).

Для данного алгоритма при «плохих» случаях глубина рекурсии достигает n, что может вызвать переполнение стека при огромных данных, поэтому для огромных массивов полезно комбинировать Quick Sort с [Merge Sort (Сортировка слиянием)](https://dev64.wordpress.com/2013/07/02/merge-sort/), а для небольших массивов с [Insertion Sort (сортировка вставкой)](https://dev64.wordpress.com/2013/07/01/insertion-sort/).

Т.е. огромный массив может разбиваться на части с помощью Merge Sort, части некоторого разумного размера могут сортироваться с помощью Quick Sort, а внутри алгоритма QuickSort, в свою очередь, если pivot неудачно разбил массив на большую и маленькую часть, сортировать меньшую часть не рекурсивным Quick Sort, а обычным Insertion Sort. Это позволит получить более оптимизированный алгоритм. Достоинства и недостатки алгоритма хорошо перечислены здесь: [http://ru.wikipedia.org/wiki/Быстрая\_сортировка](http://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0). В качестве одного из вариантов улучшения алгоритма, предлагается разбиение не на две, а на три части для уменьшения глубины рекурсии: <http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf>

Наилучшая иллюстрация алгоритма деления (функции partition()), здесь: <http://www.algolist.net/Algorithms/Sorting/Quicksort>. Также полезно посмотреть как это работает под отладчиком:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Test  public void testQuickSort() {      int a[] = {1, 12, 5, 26, 7, 14, 3, 7, 2, 3};        quickSort(a, 0, a.length-1);        System.out.println(Arrays.toString(a));  } |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://www.algolist.net/Algorithms/Sorting/Quicksort>

# Quicksort

Quicksort is a fast sorting algorithm, which is used not only for educational purposes, but widely applied in practice. On the average, it has O(n log n) complexity, making quicksort suitable for sorting big data volumes. The idea of the algorithm is quite simple and once you realize it, you can write quicksort as fast as [bubble sort](http://www.algolist.net/Algorithms/Sorting/Bubble_sort).

## Algorithm

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

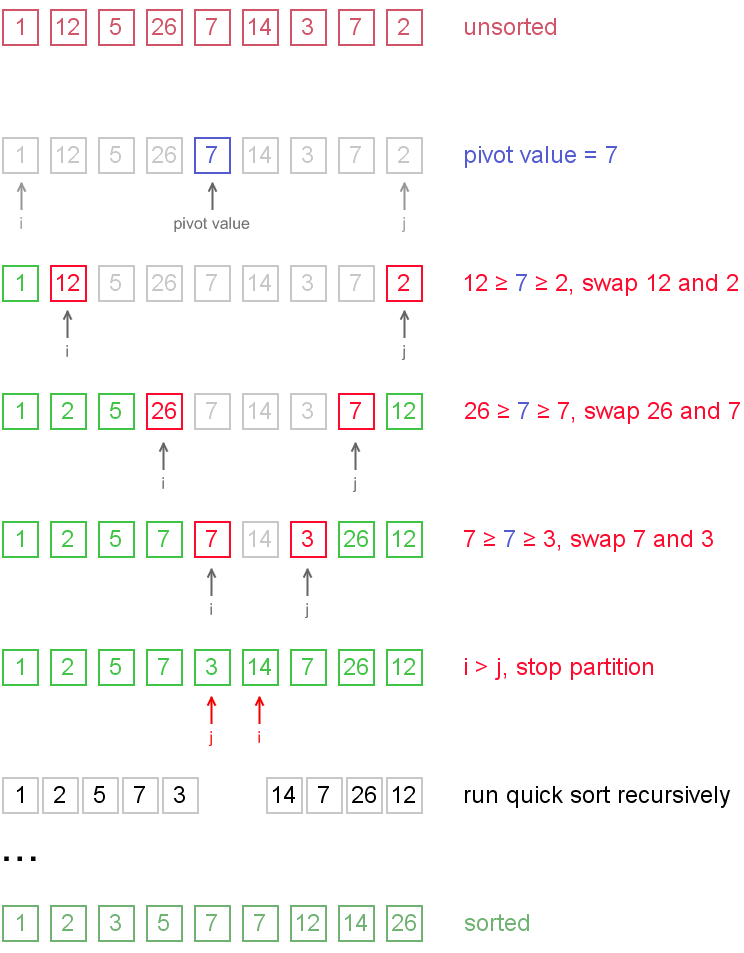
1. **Choose a pivot value.** We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. **Partition.** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. **Sort both parts.** Apply quicksort algorithm recursively to the left and the right parts.

### Partition algorithm in detail

There are two indices **i** and **j** and at the very beginning of the partition algorithm **i** points to the first element in the array and **j** points to the last one. Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found. Index **j** is moved backward, until an element with value lesser or equal to the pivot is found. If **i ≤ j** then they are swapped and i steps to the next position (**i + 1**), j steps to the previous one **(j - 1)**. Algorithm stops, when **i** becomes greater than **j**.

After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

Example. Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.



Notice, that we show here only the first recursion step, in order not to make example too long. But, in fact, {1, 2, 5, 7, 3} and {14, 7, 26, 12} are sorted then recursively.

## Why does it work?

On the partition step algorithm divides the array into two parts and every element **a** from the left part is less or equal than every element **b** from the right part. Also **a** and **b** satisfy **a ≤ pivot ≤ b** inequality. After completion of the recursion calls both of the parts become sorted and, taking into account arguments stated above, the whole array is sorted.

## Complexity analysis

On the average quicksort has O(n log n) complexity, but strong proof of this fact is not trivial and not presented here. Still, you can find the proof in [[1]](http://www.algolist.net/Algorithms/Sorting/Quicksort#cormen_book). In worst case, quicksort runs O(n2) time, but on the most "practical" data it works just fine and outperforms other O(n log n) sorting algorithms.

## Code snippets

Partition algorithm is important per se, therefore it may be carried out as a separate function. The code for C++ contains solid function for quicksort, but Java code contains two separate functions for partition and sort, accordingly.

### Java

**int** partition(**int** arr[], **int** left, **int** right)

{

**int** i = left, j = right;

**int** tmp;

**int** pivot = arr[(left + right) / 2];

**while** (i <= j) {

**while** (arr[i] < pivot)

                  i++;

**while** (arr[j] > pivot)

                  j--;

**if** (i <= j) {

                  tmp = arr[i];

                  arr[i] = arr[j];

                  arr[j] = tmp;

                  i++;

                  j--;

            }

      };

**return** i;

}

**void** quickSort(**int** arr[], **int** left, **int** right) {

**int** index = partition(arr, left, right);

**if** (left < index - 1)

            quickSort(arr, left, index - 1);

**if** (index < right)

            quickSort(arr, index, right);

}

### C++

void quickSort(int arr[], int left, int right) {

      int i = left, j = right;

      int tmp;

      int pivot = arr[(left + right) / 2];

      /\* partition \*/

      while (i <= j) {

            while (arr[i] < pivot)

                  i++;

            while (arr[j] > pivot)

                  j--;

            if (i <= j) {

                  tmp = arr[i];

                  arr[i] = arr[j];

                  arr[j] = tmp;

                  i++;

                  j--;

            }

      };

      /\* recursion \*/

      if (left < j)

            quickSort(arr, left, j);

      if (i < right)

            quickSort(arr, i, right);

}

## Full quicksort package

Full quicksort package includes:

* Ready-to-print PDF version of quicksort tutorial.
* Full, thoroughly commented quicksort source code (Java & C++).
* Generic quicksort source code in Java **(Advanced).**
* Generic quicksort source code using templates in C++ **(Advanced).**

Download link: [full quicksort package](http://www.algolist.net/Algorithms/Sorting/Quicksort/Full).

## Recommended books

1. [Cormen, Leiserson, Rivest. **Introduction to algorithms.**](http://www.algolist.net/Books/0262531968) **(Theory)**
2. [Aho, Ullman, Hopcroft. **Data Structures and Algorithms.**](http://www.algolist.net/Books/0201000237) **(Theory)**
3. [Robert Lafore. **Data Structures and Algorithms in Java.**](http://www.algolist.net/Books/0672324539) **(Practice)**
4. [Mark Allen Weiss. **Data Structures and Problem Solving Using C++.**](http://www.algolist.net/Books/020161250X) **(Practice)**

## Visualizers

1. [Quicksort Animation (with source code line by line visualization)](http://www.cs.oswego.edu/%7Emohammad/classes/csc241/samples/sort/Sort2-E.html)
2. [Quicksort in Java Applets Centre](http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/QSort.html)
3. [Animated Sorting Algorithms: Quicksort](http://www.sorting-algorithms.com/quick-sort)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<https://dev64.wordpress.com/2013/07/02/merge-sort/>

# Merge Sort (Сортировка слиянием)

### [Галерея](https://dev64.wordpress.com/type/gallery/)

Posted on [Июль 2, 2013](https://dev64.wordpress.com/2013/07/02/merge-sort/)

Продолжаю на досуге просмотр лекций курса «Introduction to Algorithms» MIT: <http://videolectures.net/mit6046jf05_leiserson_lec01/>. В результате получился небольшой тестовый код, реализующий сортировку слиянием.

Сортировка слиянием заключается в последовательном делении сортируемого массива пополам, сортировки половинок по отдельности, а затем слияния двух отсортированных массивов в один результирующий. Деление пополам происходит рекурсивно, до тех пор, пока размер сортируемого массива не станет 1 или 2.  
Если размер массива 1, то возвращаем новый массив из одного элемента (ничего сортировать не надо), если размер 2, тогда сравниваем 2 элемента и возвращаем массив из 2 элементов либо в том же порядке, либо в переставленном. В зависимости от того, правильно ли идут элементы в исходном массиве.

Ниже исходный код. Функция merge() осуществляет слияние двух отсортированных массивов в один (Используется внутри mergeSoft()). Функция mergeSort() — рекурсивно сортирует и мержит.

Юнит-тесты:

* testMerge() — показывает работу merge()
* testMergeSoft() — тестирует mergeSoft()

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | public class MergeSort {        int[] merge(int a[], int startA, int endA, int b[], int startB, int endB) {          int aLength=endA-startA+1;          int bLength=endB-startB+1;            int[] c = new int[aLength+bLength];            int ai = 0;          int bi = 0;            for(int i=0; i<c.length; i++) {              if(ai>=aLength) {                  c[i] = b[bi++];                  continue;              }              if(bi>=bLength) {                  c[i] = a[ai++];                  continue;              }              if(a[ai]<b[bi]) {                  c[i] = a[ai++];              } else {                  c[i]= b[bi++];              }          }          return c;      }        int [] mergeSort(int [] a, int start, int end) {            int length = end-start+1;            if(length == 1) {              return new int[] { a[start] };          }            if(length == 2) {              if(a[start]<a[end]) {                  return new int[] { a[start], a[end] };              } {                  return new int[] { a[end], a[start] };              }          }            int half = length/2;            int [] b = mergeSort(a, start, start+half-1);          int [] c = mergeSort(a, start+half, end);            return merge(b, 0, b.length-1, c, 0, c.length-1);      }          @Test      public void testMerge() {          int[] a = { 1, 5, 8 };          int[] b = { 3, 4 };            int[] c = merge(a, 0, 2, b, 0, 1);            System.out.println(Arrays.toString(c));      }        @Test      public void testMergeSoft() {          int[] a = { 3, 10, 8, 6, 3, 1, 7, 2, 5, 3};            System.out.println(Arrays.toString(mergeSort(a, 0, a.length-1)));      }    } |

Асимптотическая сложность такого алгоритма сортировки O(n \* ln n). Еще важно знать что более медленные алгоритмы могут работать быстрее при некотором диапозоне от (0 до n>0). Поэтому не всегда более быстрый алгоритм выигрывает у более медленного. Например, если сравнивать [сортировку вставкой](https://dev64.wordpress.com/2013/07/01/insertion-sort/) с сортировкой слиянием, то получится, что Merge Sort выигрывает у простой сортировки вставкой только при достаточно больших n. [Первоисточник](http://videolectures.net/mit6046jf05_leiserson_lec01/) гласит, что сортировка слиянием выигрывает у сортировки вставкой при n > 30. Мне думается, что это число может быть и больше из-за необходимости выделения дополнительной памяти.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<https://dev64.wordpress.com/2013/07/01/insertion-sort/>

# Сортировка вставкой

### [Галерея](https://dev64.wordpress.com/type/gallery/)

Posted on [Июль 1, 2013](https://dev64.wordpress.com/2013/07/01/insertion-sort/)

Данный пост написан в процессе обновления знаний по базовым алгоритмам и английскому языку по материалам лекций курса «Introduction to Algorithms» MIT: <http://videolectures.net/mit6046jf05_leiserson_lec01/>.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | @Test  public void test() {      int[] a = { 3, 10, 8 , 6, 3 ,1 , 7, 2, 5, 3};        for(int j = 1; j<=a.length-1; j++) {          int key = a[j];            for(int i=j-1; i>=0 && a[i]<key; i--) {              a[i+1] = a[i];              a[i]=key;          }        }          for(int i=0; i<a.length; i++) {          System.out.println(String.valueOf(a[i])+",");      }  } |

Код выше на Java для наглядности. Конечно, на Java (да и других языках тоже), нужно пользоваться готовыми функциями сортировки коллекций. Однако, иногда стоит вспомнить базовые алгоритмы. Мало-ли. Вдруг придется писать что-то на низком уровне в условиях отсутствия библиотек.

На java сортировку можно сделать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Test  public void testJavaSort() {      Integer[] a = { 3, 10, 8  6, 3, 1, 7, 2, 5, 3};        Arrays.sort(a, new Comparator<Integer>() {          public int compare(Integer o1, Integer o2) {              return o1-o2;          }      });        System.out.println(Arrays.toString(a));  } |

В данном случае, правда, массив объектов уже не базового типа int. Вместо этого используется обертка Integer, чтобы можно было описать класс Comparator. (Кстати говоря, способа автоматически создать из массива базовых типов массив оберток в Java нет [(proof)](http://stackoverflow.com/questions/880581/how-to-convert-int-to-integer-in-java).) Сортировка осуществляется одним вызовом функции Arrays.sort(). Первым параметром передается ссылка на массив, который нужно сортировать, вторым передается анонимный объект, созданный на основе интерфейса Comparator.

Здесь же применен самый простейший способ распечатать массив в Java (способ конвертировать массив в строку):

System.out.println(Arrays.toString(a));

Функция Arrays.toString(), превращает массив в строку.

Если же придется сортировать массив базовых int, то, чтобы использовать стандартные средства сортировки, сначала придется сделать копию массива в виде Integer[]… А для этого все равно стандартной функции нет и придется накидывать eё вручную ([(proof)](http://stackoverflow.com/questions/880581/how-to-convert-int-to-integer-in-java)).

Копировать стандартными способами можно опять таки только массивы оберточных типов. Например пример ниже скомпилируется, но бросит исключение:

java.lang.ArrayStoreException  
at java.lang.System.arraycopy(Native Method)  
at InsertionSortTest.testJavaWithCopy(InsertionSortTest.java:44)  
...

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Test   public void testJavaWithCopy() {       int[] a = { 3, 10, 8 , 6, 3 ,1 , 7, 2, 5, 3};         Integer[] wrapped = new Integer[a.length];           System.arraycopy(a, 0, wrapped, 0, a.length);         Arrays.sort(wrapped, new Comparator<Integer>() {           public int compare(Integer o1, Integer o2) {               return o1-o2;           }       });         System.out.println(Arrays.toString(wrapped));   } |

А если заменить в нем int[] на Integer[], то все будет работать без проблем:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Test   public void testJavaWithCopy() {       Integer[] a = { 3, 10, 8 , 6, 3 ,1 , 7, 2, 5, 3};         Integer[] wrapped = new Integer[a.length];           System.arraycopy(a, 0, wrapped, 0, a.length);         Arrays.sort(wrapped, new Comparator<Integer>() {           public int compare(Integer o1, Integer o2) {               return o1-o2;           }       });         System.out.println(Arrays.toString(wrapped));   } |

Собственно и копирование то как было показано выше тут не нужно. Оно только для демонстрации…

Выводы… Вывод интересный. Примитивные типы использовать вроде бы выгодно в плане быстроты работы. Но вся скорость, полученная за счет использования примитивных типов может потеряться из-за дополнительных операций конвертации или копирования, которые могут потребоваться в некоторых случаях.

P.S. Несколько слов про сложность сортировки вставкой. В худшем и усредненном случае сортировка вставкой имеет сложность O(n^2). Однако в случае если б’ольшая часть массива уже отсортирована правильным образом, тогда сложность такой сортировки примерно равна O(n), что существенно лучше. Таким образом, если вы знаете что начальная часть данных заранее отсортирована (сортировка выполняется после добавления небольшого количества данных в большой отсортированный массив), то сортировка вставкой то, что нужно.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*