

Dual-Pivot Quicksort algorithm

Vladimir Yaroslavskiy

iaroslavski@mail.ru

First revision: February 16, 2009

Last updated: September 11, 2009

Introduction

Sorting data is one of the most fundamental problems in Computer Science, especially if the arranging objects are primitive ones, such as integers, bytes, floats, etc. Since sorting methods play an important role in the operation of computers and other data processing systems, there has been an interest in seeking new algorithms better than the existing ones. We compare sorting methods by the number of the most "expensive" operations, which influence on effectiveness of the sorting techniques, — comparisons and swaps. Quicksort algorithm is an effective and wide-spread sorting procedure with $C \cdot n \cdot \ln(n)$ operations, where n is the size of the arranged array. The problem is to find an algorithm with the least coefficient C . There were many attempts to improve the classical variant of the Quicksort algorithm:

1. Pick an element, called a *pivot*, from the array.
2. Reorder the array so that all elements, which are less than the pivot, come before the pivot and all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot element is in its final position.
3. Recursively sort the sub-array of lesser elements and the sub-array of greater elements.

Hoare, Helman, Knuth, Sedgewick, and other scientists worked mostly on the effectiveness of the concrete "divide and conquer" algorithm implementations, or tried to increase performance due to the specific choice of the pivot element, but all of them used the classical partitioning scheme with **two** parts.

We can show that using **two** pivot elements (or partitioning to **three** parts) is more effective, especially on large arrays. We suggest the new *Dual-Pivot Quicksort* scheme, faster than the known implementations, which improves this situation. The implementation of the Dual-Pivot Quicksort algorithm has been investigated on different inputs and primitive data types. Its advantages in comparison with one of the most effective known implementations of the classical Quicksort algorithm [1], [2], and implementation of Quicksort in JDK™ 6.0 platform [3] have been shown.

New Dual-Pivot Quicksort algorithm

The new Quicksort algorithm uses partitioning a source array $T[] a$, where T is primitive type (such as int, float, byte, char, double, long and short), to three parts defined by two pivot elements **P1** and **P2** (and therefore, there are three pointers **L**, **K**, **G**, and **left** and **right** — indices of the first and last elements respectively) shown in Figure 1:

Figure 1.

P1	< P1	P1 <= & <= P2	?	> P2	P2
left	L	K	G	right	
part I		part II		part III	
		part IV			

The algorithm provides the following steps:

1. For small arrays (**length < 27**), use the Insertion sort algorithm.
2. Choose two pivot elements **P1** and **P2**. We can get, for example, the first element **a[left]** as **P1** and the last element **a[right]** as **P2**.
3. **P1** must be less than **P2**, otherwise they are swapped. So, there are the following parts:
 - **part I** with indices from **left+1** to **L-1** with elements, which are less than **P1**,
 - **part II** with indices from **L** to **K-1** with elements, which are greater or equal to **P1** and less or equal to **P2**,
 - **part III** with indices from **G+1** to **right-1** with elements greater than **P2**,
 - **part IV** contains the rest of the elements to be examined with indices from **K** to **G**.
4. The next element **a[K]** from the **part IV** is compared with two pivots **P1** and **P2**, and placed to the corresponding **part I**, **II**, or **III**.
5. The pointers **L**, **K**, and **G** are changed in the corresponding directions.
6. The steps 4 - 5 are repeated while **K ≤ G**.
7. The pivot element **P1** is swapped with the last element from **part I**, the pivot element **P2** is swapped with the first element from **part III**.
8. The steps 1 - 7 are repeated recursively for every **part I**, **part II**, and **part III**.

Mathematical proof

It is proved that for the Dual-Pivot Quicksort the average number of comparisons is $2*n*\ln(n)$, the average number of swaps is $0.8*n*\ln(n)$, whereas classical Quicksort algorithm has $2*n*\ln(n)$ and $1*n*\ln(n)$ respectively. Full mathematical investigation has been done and will be copied here (TBD).

Comparison and summary

The new Dual-Pivot Quicksort algorithm provides the following advantages:

- While sorting primitive objects, it is more efficient to use partitioning of unsorted array to 3 parts instead of the usage of the classical approach. The more the size of the array to be sorted, the more efficiently the new algorithm works in comparison with the classical Quicksort [2] and the Quicksort implemented in JDK 6.0 platform [3]. For example, we provided the following numerical experiment: 2 000 000 random integer elements were sorted 50 times using the new Dual-Pivot Quicksort, algorithms [2] and [3], and analyzed the calculation time. It took 16.5, 18.9, and 20.3 seconds respectively. The implementation of the new Dual-Pivot Quicksort algorithm for integers can be easily adjusted for another numeric, string and comparable types.

- The suggested Dual-Pivot Quicksort algorithm also works quicker than the classical schemes on the arranged arrays or the arrays with repeated elements. In these cases of nonrandom inputs the time metric for the Dual-Pivot Quicksort algorithm is 55 against 100 for Quicksort implemented in JDK 6.0 platform, where all tests have been run on.
- The supposed algorithm has additional improvement of the special choice procedure for pivot elements **P1** and **P2**. We take not the first **a[left]** and last **a[right]** elements of the array, but choose two middle elements. The described modification does not make worse the properties of the Dual-Pivot Quicksort algorithm in the case of random source data:

```

    int third = arrayLength / 3;
    P1 = a[left + third];
    P2 = a[right - third];

```

We can summarize the features of the suggested algorithm as follows:

- Time savings.
- The "divide and conquer" strategy.
- Usage of two pivot elements instead of one.
- More effective sorting procedure that can be used in numerical analysis.
- The Dual-Pivot Quicksort algorithm could be recommended in the next JDK releases.

Implementation in Java™ programming language

```

public static void sort(int[] a) {
    sort(a, 0, a.length);
}

public static void sort(int[] a, int fromIndex, int toIndex) {
    rangeCheck(a.length, fromIndex, toIndex);
    dualPivotQuicksort(a, fromIndex, toIndex - 1, 3);
}

private static void rangeCheck(int length, int fromIndex, int toIndex) {
    if (fromIndex > toIndex) {
        throw new IllegalArgumentException("fromIndex > toIndex");
    }
    if (fromIndex < 0) {
        throw new ArrayIndexOutOfBoundsException(fromIndex);
    }
    if (toIndex > length) {
        throw new ArrayIndexOutOfBoundsException(toIndex);
    }
}

private static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

private static void dualPivotQuicksort(int[] a, int left, int right, int div) {
    int len = right - left;

    if (len < 27) { // insertion sort for tiny array
        for (int i = left + 1; i <= right; i++) {

```

```

        for (int j = i; j > left && a[j] < a[j - 1]; j--) {
            swap(a, j, j - 1);
        }
    }
    return;
}
int third = len / div;

// "medians"
int m1 = left + third;
int m2 = right - third;

if (m1 <= left) {
    m1 = left + 1;
}
if (m2 >= right) {
    m2 = right - 1;
}
if (a[m1] < a[m2]) {
    swap(a, m1, left);
    swap(a, m2, right);
}
else {
    swap(a, m1, right);
    swap(a, m2, left);
}
// pivots
int pivot1 = a[left];
int pivot2 = a[right];

// pointers
int less = left + 1;
int great = right - 1;

// sorting
for (int k = less; k <= great; k++) {
    if (a[k] < pivot1) {
        swap(a, k, less++);
    }
    else if (a[k] > pivot2) {
        while (k < great && a[great] > pivot2) {
            great--;
        }
        swap(a, k, great--);

        if (a[k] < pivot1) {
            swap(a, k, less++);
        }
    }
}
// swaps
int dist = great - less;

if (dist < 13) {
    div++;
}
swap(a, less - 1, left);
swap(a, great + 1, right);

// subarrays
dualPivotQuicksort(a, left, less - 2, div);
dualPivotQuicksort(a, great + 2, right, div);

```

```

// equal elements
if (dist > len - 13 && pivot1 != pivot2) {
    for (int k = less; k <= great; k++) {
        if (a[k] == pivot1) {
            swap(a, k, less++);
        }
        else if (a[k] == pivot2) {
            swap(a, k, great--);
        }
        if (a[k] == pivot1) {
            swap(a, k, less++);
        }
    }
}
// subarray
if (pivot1 < pivot2) {
    dualPivotQuicksort(a, less, great, div);
}
}

```

Experimental results

The a lot of tests have been run, here is the time of executions:

Server VM:

http://spreadsheets.google.com/pub?key=t_EAWUkO4mD3BIbOv8Fa-A0&output=html

Client VM:

<http://spreadsheets.google.com/pub?key=tdiMo8xleTxd23nKU0bcz00&single=true&gid=0&output=html>

References

1. Classical Quicksort: <http://en.wikipedia.org/wiki/Quicksort>
2. Sample: <http://www.roseindia.net/java/beginners/arrayexamples/QuickSort.shtml>
3. Jon L. Bentley, M. Douglas McIlroy. Engineering a Sort Function. Software–Practice and Experience, Vol.23 (11), P.1249–1265, November 1993.
4. Big O notation: http://en.wikipedia.org/wiki/Big_O_notation
5. Stirling's formula: http://en.wikipedia.org/wiki/Stirling%27s_approximation