Dual-Pivot Quicksort

Vladimir Yaroslavskiy
iaroslavski@mail.ru

First revision: February 16, 2009
Last updated: September 22, 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*n*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.

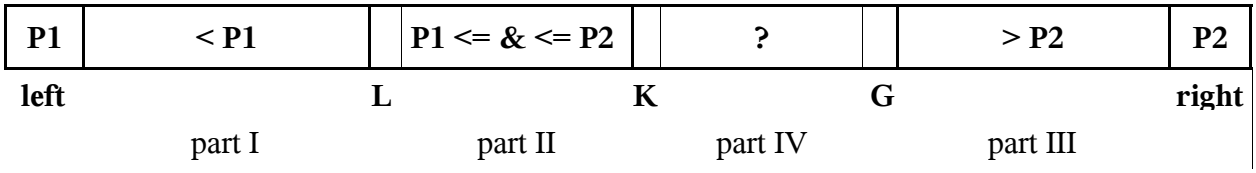
Hoare, Helman, Knuth, Sedgewick, Bentley and other scientists worked mostly on the effectiveness of the concrete "divide and conquer" algorithm implementations, or tried to increase performance due to the specific choice of the pivot element, but all of them used the classical partitioning scheme with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



The algorithm provides the following steps:

1. For small arrays (**length < 17**), 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.

At first consider the classic Quicksort scheme and find the average number of comparisons and swaps for it. We assume that input data is random permutation of n numbers from the range [1..n].

Classic Quicksort
==========
1. Choose a pivot element (take random),
2. Compare each (n-1) elements with the pivot and swap it, if necessary, to have the partitions: [<= pivot | >= pivot]
4. Sort recursively left and right parts.

From the algorithm above, the average number of comparisons C_n as a function of the number of elements may be represented by the equation:

(1) C_n = (n-1) + 1/n * sum_{k=0}^{n-1} {C_k + C_{n-k-1}}

and last sum can be rewritten:

(2) C_n = (n-1) + 2/n * sum_{k=0}^{n-1} C_k

Write formula (2) for n+1:

(3) C_{n+1} = n + 2/(n+1) * sum_{k=0}^{n} C_k

Multiply (2) by n and (3) by (n+1) and subtract one from other, we have:

(4) (n+1)*C_{n+1} - n*C_n = 2*n + 2*C_n

Sorting an array of n elements may be considered as selecting one permutation of the n elements among all possible permutations. The number of possible permutations of n elements is n!, so the task for any sorting algorithm is to determine the one permutation out of n! possibilities. The minimum number of operations (swap and comparisons) for sorting n elements is const*ln(n!). From the Stirling's formula the approximation of the number of operations is A*n*ln(n) + B*n + C, where A, B and C are constant coefficients. The coefficients B and C are not important for large n. Therefore, the function C_n may be approximated by the equation:

(5) C_n = A*n*ln(n)

The function C_n is substituted from equation (5) into equation (4), which yields the following equation:

(6) (n+1)*A*(n+1)*ln(n+1) - n*A*n*ln(n) = 2*n + 2*A*n*ln(n)

Using the properties of logarithms, equation (6) can then be reduced to:

(7) n*ln(1+1/n) + 2*ln(1+1/n) + (1/n) * ln(n+1) = 2/A

Using a property of logarithm: ln(1 + x) -> x, if x -> 0, and other property: ln(n) / n -> 0, when n -> +oo, equation (7) will be approximated by:

(8) 1 + 0 + 0 = 2/A

So, the coefficient A is equal to 2 and the average number of comparisons in sorting of n size arrays is

(9) C_n = 2*n*ln(n).

To find the approximation of the average number of swaps, we use the similar approach as in the case of comparisons. The average number of swaps S_n as a function of the number of elements may be represented by the equation:

(10) S_n = 1/2*(n-1) + 2/n * sum_{k=0}^{n-1} S_k

We assume that average number of swaps during one iteration is 1/2*(n-1). It means that in average one half of elements is swapped only. Using the same approach, we find that the coefficient A equals to 1. Therefore, the function S_n may be approximated by the equation:

(11) S_n = n*ln(n)

Now consider the Dual-Pivot Quicksort scheme and find the average number of comparisons and swaps for it. We assume that input data is random permutation of n numbers from the range [1..n].

Dual-Pivot Quicksort
==========
1. Choose 2 pivot elements pivot1 and pivot2 (take random),
2. pivot1 must be less or equal than pivot2, otherwise they are swapped
3. Compare each (n-2) elements with the pivots and swap it, if necessary, to have the partitions: [<= p1 | p1 <= & <= p2 | >= p2]
5. Sort recursively left, center and right parts.

From the algorithm above, the average number of comparisons C_n as a function of the number of elements may be represented by the equation:

(1) C_n = 1 + 2/(n*(n-1)) * sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {C_i + 1*i + C_{j-i-1} + 2*(j-i-1) + C_{n-j-1} + 2*(n-j-1)}

Equation (1) means that total number is the sum of the comparison numbers of all cases of partitions into 3 parts plus number of comparisons for elements from left part (one comparison), center and right parts (2 comparisons).

We will show that 2/(n*(n-1)) * sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {i + 2*(j-i- 1) + 2*(n-j-1)} equals to 5/3 * (n-2). Let's consider the double sum:

sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {i + 2*(j-i- 1) + 2*(n-j-1)} = sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {2*n - 4} - sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {i} = 2*(n-2)*sum_{i=0}^{n-2} {n-1-i} - sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {i} = 2*(n-2)*((n-1)^2 - (n-1)*(n-2)/2) - (n-1)*sum_{i=0}^{n-2} {i} + sum_{i=0}^{n-2} {i*i} =

---
here we use the property: sum_{k=1}{n} {k^2} = n^3/3 + n^2/2 + n/6

= 2*(n-2)*((n-1)^2 - (n-1)*(n-2)/2) - (n-1)*(n-1)*(n-2)/2 + (n-2)^3/3 + (n-2)^2/2 + (n-2)/6 = 1/6 * (12*(n-1)^2*(n-2) - 6*(n-1)*(n-2)^2 - 3*(n-1)^2*(n-2) + 2*(n-2)^3 + 3*(n-2)^2 + (n-2)) = 1/6 * (3*(n-1)*(n-2)*(3*(n-1) - 2*(n-2)) + (n-2)*(2*(n-2)^2 + 3*(n-2) + 1))) = 1/6 * (3*(n-1)*(n-2)*(n+1) + (n-2)*(2*n^2 - 5*n + 3)) =1/6 * (n-2)*(5*n^2 - 5*n) = 5/6 * n*(n-1)*(n-2)

Substitute the result into equation (1):

C_n = 1 + 2/(n*(n-1)) * 5/6 * n*(n-1)*(n-2) + 2/(n*(n-1)) * sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {C_i + C_{j-i-1} + C_{n-j-1}}

or

(2) C_n = 1 + 5/3*(n-2) + 2/(n*(n-1)) * sum_{i=0}^{n-2} sum_{j=i+1}^{n-1} {C_i + C_{j-i-1} + C_{n-j-1}}

The double sum in equation (2) can be reduced:

(3) C_n = 1 + 5/3*(n-2) + 2/(n*(n-1)) * sum_{k=0}^{n-2} {3 * (n-k-1) * C_k}

Denote 1 + 5/3*(n-2) by f_n and multiply to n*(n-1):

(4) n*(n-1)*C_n = n*(n-1)*f_n + 6 * sum_{k=0}^{n-2} {(n-k-1) * C_k}

Write formula (4) for n+1:

(5) n*(n+1)*C_{n+1} = n*(n+1)*f_n + 6 * sum_{k=0}^{n-1} {(n-k) * C_k}

Subtract (4) from (5), we have:

(6) n*(n+1)*C_{n+1} - n*(n-1)*C_n = n*(n+1)*f_n - n*(n-1)*f_n + 6 * sum_{k=0}^{n-2} C_k + 6*C_{n-1}

Denote n*(n+1)*C_{n+1} - n*(n-1)*C_n by X_n and n*(n+1)*f_n - n*(n-1)*f_n by F_n:

(7) X_n = F_n + 6 * sum_{k=0}^{n-2} C_k + 6*C_{n-1}

Write formula (7) for n+1:

(8) X_{n+1} = F_{n+1} + 6 * sum_{k=0}^{n-1} C_k + 6*C_n

Subtract (7) from (8), we have:

(9) X_{n+1} - X_n = F_{n+1} - F_n + 6*C_n

Resolving of F_{n+1} - F_n gives:

(10) X_{n+1} - X_n = 2 + 10*n + 6*C_n

The function X_n is substituted into equation (10), which yields the following equation:

(11) (n+1)*(n+2)C_{n+2} -2*n(n+1)*C_{n+1} + (n*(n-1) - 6)*C_n = 2 + 10*n

We will find the function C_n approximated by the equation:

(12) C_n = A*n*ln(n)

The function C_n is substituted from equation (12) into equation (11), which yields the following equation:

(13) (n^3+5*n^2+8*n+4)*ln(n+2) - (2*n^3+4*n^2+2*n)*ln(n+1) + (n^3-n^2-6*n)*ln(n) = (10*n+2) / A

Using the properties of logarithms, equation (13) can then be reduced to:

(14) n^3*(ln(n+2)-2*ln(n+1)+ln(n)) + n^2*(5*ln(n+2)-4*ln(n+1)-ln(n)) + n*(8*ln(n+2)-2*ln(n+1)-6*ln(n)) + 4*ln(n+2) = (10*n+2) / A

Using a property of logarithm: ln(1 + x) -> x, if x -> 0, and other property: ln(n) / n -> 0, when n -> +oo, equation (15) will be approximated by:

(15) -1 + 4 + 2 + 0 + 0 = 10 / A

So, the coefficient A is equal to 2 and the average number of comparisons in sorting of n size arrays is

(9) C_n = 2*n*ln(n).

To find the approximation of the average number of swaps, we use the similar approach as in the case of comparisons. The average number of swaps S_n as a function of the number of elements may be represented by the equation:

(10) S_n = 4 + 2/3*(n-2) + 2/(n*(n-1)) * sum_{k=0}^{n-2} {(n-k-1)*S_k}

We assume that average number of swaps during one iteration is 2/3*(n-2). It means that in average one third of elements is swapped only. Using the same approach, we find that

the coefficient A equals to 0.8. Therefore, the function S_n may be approximated by the equation:

(11) $S_n = 0.8*n*\ln(n)$

And as summary: the value of the coefficient A:

	dual-pivot	classic
comparison:	2.0	2.0
swap:	0.8	1.0

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 easy 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 from 5 middle elements. The described modification does not make worse the properties of the Dual-Pivot Quicksort algorithm in the case of random source data.

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

```
/**
 * @author Vladimir Yaroslavskiy
 * @version 2009.09.17 m765.817
 */
public class DualPivotQuicksort817 {

    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);
    }

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

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

        if (len < TINY_SIZE) { // insertion sort on tiny array
            for (int i = left + 1; i <= right; i++) {
                for (int j = i; j > left && a[j] < a[j - 1]; j--) {
                    x = a[j - 1];
                    a[j - 1] = a[j];
                    a[j] = x;
                }
            }
            return;
        }
        // median indexes
        int sixth = len / 6;
        int m1 = left + sixth;
        int m2 = m1 + sixth;
        int m3 = m2 + sixth;
        int m4 = m3 + sixth;
        int m5 = m4 + sixth;

        // 5-element sorting network
        if (a[m1] > a[m2]) { x = a[m1]; a[m1] = a[m2]; a[m2] = x; }
        if (a[m4] > a[m5]) { x = a[m4]; a[m4] = a[m5]; a[m5] = x; }
        if (a[m1] > a[m3]) { x = a[m1]; a[m1] = a[m3]; a[m3] = x; }
        if (a[m2] > a[m3]) { x = a[m2]; a[m2] = a[m3]; a[m3] = x; }
        if (a[m1] > a[m4]) { x = a[m1]; a[m1] = a[m4]; a[m4] = x; }
        if (a[m3] > a[m4]) { x = a[m3]; a[m3] = a[m4]; a[m4] = x; }
        if (a[m2] > a[m5]) { x = a[m2]; a[m2] = a[m5]; a[m5] = x; }
        if (a[m2] > a[m3]) { x = a[m2]; a[m2] = a[m3]; a[m3] = x; }
```

```
if (a[m4] > a[m5]) { x = a[m4]; a[m4] = a[m5]; a[m5] = x; }

// pivots: [ < pivot1 | pivot1 <= && <= pivot2 | > pivot2 ]
int pivot1 = a[m2];
int pivot2 = a[m4];

boolean diffPivots = pivot1 != pivot2;

a[m2] = a[left];
a[m4] = a[right];

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

// sorting
if (diffPivots) {
    for (int k = less; k <= great; k++) {
        x = a[k];

        if (x < pivot1) {
            a[k] = a[less];
            a[less++] = x;
        }
        else if (x > pivot2) {
            while (a[great] > pivot2 && k < great) {
                great--;
            }
            a[k] = a[great];
            a[great--] = x;
            x = a[k];

            if (x < pivot1) {
                a[k] = a[less];
                a[less++] = x;
            }
        }
    }
}
else {
    for (int k = less; k <= great; k++) {
        x = a[k];

        if (x == pivot1) {
          continue;
        }
        if (x < pivot1) {
            a[k] = a[less];
            a[less++] = x;
        }
        else {
            while (a[great] > pivot2 && k < great) {
                great--;
            }
            a[k] = a[great];
            a[great--] = x;
            x = a[k];

            if (x < pivot1) {
                a[k] = a[less];
                a[less++] = x;
            }
```

```
                }
            }
        }
        // swap
        a[left] = a[less - 1];
        a[less - 1] = pivot1;

        a[right] = a[great + 1];
        a[great + 1] = pivot2;

        // left and right parts
        dualPivotQuicksort(a, left,   less - 2);
        dualPivotQuicksort(a, great + 2, right);

        // equal elements
        if (great - less > len - DIST_SIZE && diffPivots) {
            for (int k = less; k <= great; k++) {
                x = a[k];

                if (x == pivot1) {
                    a[k] = a[less];
                    a[less++] = x;
                }
                else if (x == pivot2) {
                    a[k] = a[great];
                    a[great--] = x;
                    x = a[k];

                    if (x == pivot1) {
                        a[k] = a[less];
                        a[less++] = x;
                    }
                }
            }
        }
        // center part
        if (diffPivots) {
            dualPivotQuicksort(a, less, great);
        }
    }

    private static final int DIST_SIZE = 13;
    private static final int TINY_SIZE = 17;
}
```

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_EAWUkQ4mD3BIbOv8Fa-AQ&output=html

Client VM:
http://spreadsheets.google.com/pub?key=tdiMo8xleTxd23nKUObcz0Q&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