Asymmetric quicksort

Takeuchi Leorge（竹内 良治）
qmisort@gmail.com

Abstract

Quicksort, invented by Tony Hoare in 1959, is one of the fastest sorting algorithms. However, conventional implementations have some weak points, including the following: swaps to exchange two elements are redundant, deep recursive calls may encounter stack overflow, and the case of repeated many elements in input data is a well-known issue. This paper improves quicksort to make it more secure and faster using new or known ideas in C language.

1. Introduction

Quicksort[1] is a sorting algorithm that reorders an array in some logical order, such as numerical order or lexicographical order. Quicksort chooses an array element as a pivot, and then iterates the exchanging of elements such that greater elements than the pivot are moved to the right of a lesser element if existing, and lesser elements are moved to the left of a greater element if existing. Conventionally, a lesser element and greater element are swapped. The pivot is usually swapped with the first or last element before the iteration, and finally swapped again with an element at the boundary of lesser or equal elements and greater or equal elements (equal elements don't move). Then an array is divided into two sub-partitions with the pivot element between them. This partitioning operation is applied on two sub-partitions recursively until any partitions consist of one or no element. When all of the recursive partitioning is completed, the array is sorted.

Applying a pivot hole instead of swaps reduces the number of copies by about 1/3, and makes the simplest new quicksort faster. Conversion of the recursive calls for longer sub-partition to iteration prevents stack overflow. The issue of repeated elements is resolved by asymmetric loops and the expansion of a pivot element to continuous equal elements in partitioning. The random choice of pivot avoids other malicious input data, to make quicksort more secure. The speed of quicksort is increased by the choice of the median of several elements as the pivot.

Note in this paper: **`N`** refers to the number of elements. The output tabs are converted to spaces to adjust columns. Bold font is used for emphasis.

2. Pivot hole

A swap needs three copies to exchange two elements: t <-- a, a <-- b, b <-- t, where a and b are elements to be exchanged, and t is a temporary buffer. As this operation could be considered redundant, I suggest using a **pivot hole** instead of swaps. When an element moves away, its previous place is regarded as an empty hole like an electron hole in a current. An element is chosen as a pivot and saved in a temporary buffer; thus, the first hole is dug at a pivot element. Then, greater or lesser elements than the pivot move to a hole alternately, and the final hole is filled with the saved pivot.

The simplest pseudocode is presented below.

```
Quicksort(a[], lo, hi)
    IF lo < hi
        p = Partition(a, lo, hi) // the boundary position is returned
        Quicksort(a, lo, p – 1)  // sort lesser or equal elements recursively
        Quicksort(a, p + 1, hi)  // sort greater or equal elements recursively

Partition(a[], lo, hi)
    pivot = a[hi]        // dig a hole
    hole = hi--
    WHILE lo < hole
        IF a[lo] > pivot
            a[hole] = a[lo]      // move a greater element
            hole = lo            // move a hole
            WHILE hi > hole
                IF a[hi] < pivot // lesser element
                    a[hole] = a[hi]
                    hole = hi
                hi--
        lo++
    a[hole] = pivot      // restore the pivot
    RETURN hole
```

The following example demonstrates the behavior of this process.

```
(C, A, D, B)     Input array.
(C, A, D, -)  B  Save the last element B as a pivot. Then, '-' is a hole.
(-, A, D, C)  B  Search for a greater element from the first position,
                 and find C, then move C to the hole.
(A, -, D, C)  B  Search for a lesser element from the position before C,
                 and find A, then move A to the hole.
(A, B, D, C)     Restore the pivot B to the hole.
(A), B, (D, C)   Divide the partition.
A, B, (D, C)     (A) contains only one element.
A, B, (D, -)  C  Save the last element C as a pivot in (D, C).
A, B, (-, D)  C  Move D to the hole.
A, B, (C, D)     Restore the pivot C.
A, B, C, (D)     Divide the partition (C, D) to C and (D).
A, B, C, D       (D) contains only one element.
```

The number of calls, comparisons and copies, and the performance of algorithms are evaluated by executable programs[1] written in C language with Eclipse 3.8.1 on ubuntu 14.04 LTS. Eclipse generates two executables to debug and to release. The numbers are measured in a debug build program.

The following is an example of output from the program.

```
$ random.awk 15 | xargs echo  # Sample of random data sequence
07 01 10 14 14 05 12 01 09 00 06 13 11 08 09
$ N=100000; random.awk $N | Debug/Sort -N $N -fhV 1
arguments : -N 100000 -fhV 1
qsort(3)      usec = 55834 call = 0     compare = 1536216 copy = 0
qsort_first() usec = 58694 call = 64360 compare = 2099403 copy = 1163697
quick_hole()  usec = 50152 call = 69505 compare = 2071997 copy = 796515
$ /lib/x86_64-linux-gnu/libc.so.6 | head -1   # version of GNU library
GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.7) stable release version 2.19, by Roland
McGrath et al.
```

The following list explains commands used above.

random.awk : Awk script to generate a random data sequence in a range [0, N).
awk : An interpreter for the AWK Programming Language.
xargs : Linux command to build and execute command lines from standard input
echo : Linux command to display a line of text
Debug/Sort : A debug build program in the Debug sub-directory. A command option **-?** shows all command options.

1 Repository - https://github.com/leorge/qmisort
How to evaluate and build - https://sites.google.com/site/qmisort/qmisort

-N *xx* : Command option to set the number of elements in input array.
qsort(3) : A function in GNU C library to sort an array. The number of calls and copies are uncountable.
Section number 3 enclosed in parentheses refers to Library calls.
qsort_first() : Conventional quicksort called by the -f option.
quick_hole() : New quicksort called by the -h option.
-V 1 : Tracing level to debug.
head : Linux command to output the first part of files.

qsort_first() is the simplest conventional quicksort that chooses the first element as a pivot and exchanges elements with swaps. The template of qsort_first() is Quick.java written by Robert Sedgewick and Kevin Wayne. quick_hole() is the implemented pseudocode above. The number of calls and comparisons are not that different from qsort_first(), but the number of copies in quick_hole() is about 2/3 that in qsort_first(). The processing time in microseconds as shown in the fourth field of output is inaccurate. A field is an area within a line separated by blank spaces that stores a particular type of data.

The following chart shows the accurate relative consumed CPU time[2] of algorithms in various N measured by a **release build** program[3]. The Y axis is the normalized $CPU\_time/n\log(n)$, where the value of quick_hole() is 1 at N=2^20=1M. The function strcmp(3) is used here to compare two strings. The size of an element is 16 bytes.

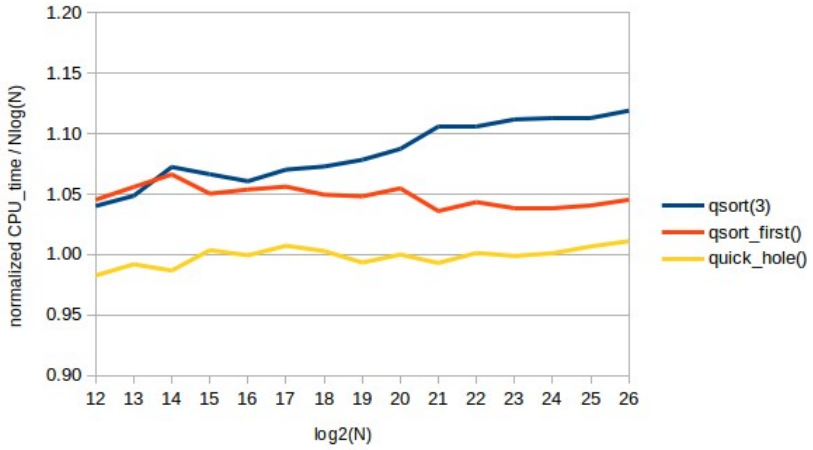

Fig. 1: Swap vs. Hole

Data for Fig.1 and Fig.2 - https://github.com/leorge/qmisort/wiki/data/Qsort/hole

quick_hole() is faster than qsort_first().

The series the chart is based on is the average of 20 random sequences to reduce data dependency, and each datapoint is the mean of 10 CPU times to reduce measurement error, as below. Thus, the value of a point is the average of 200 CPU times.

The following is a sample of output with the Release build program.

```
$ N=1000; random.awk $N | Release/Sort -N $N -f
qsort(3)      usec = 461 spread = 119 26 % [1467] (599) 595 593 590 593 476 354 353 353
352 352
qsort(3)      usec = 358 spread = 5    1 % [373] 365 364 362 (376) 362 357 354 354 354
353 352
qsort_first() usec = 221 spread = 0    0 % [238] (224) 222 222 221 221 221 222 221 221
221 221
```

The first field is a function name that represents an algorithm. The fourth field is the mean of 10 of the consumed CPU times listed in the last 12 fields. The first value enclosed in the square brackets is omitted from

2 Consumed CPU time of a process is measured by the function clock_gettime(2).
3 The release build program runs on Live ArchLinux which is very light-weight since it has no windows system..

calculation because it is sometimes a bit larger. Since it may take a short time to load a binary program onto a cache memory, the first value in qsort(3) 1467 is the largest. The value enclosed in the square brackets in the second line is not the largest because the program-cache is not changed, and the value in the third line is the largest because it may take a very short time to replace a small part of the program-cache for qsort(3) with qsort_first(). Excluding the first result, the largest value enclosed in parentheses is also omitted because it may be huge in probability. Thus, the mean is the average of 10 consumed CPU times. The seventh field is the estimated standard deviation (Stdev)[2], and the next field is the percentage of the mean: Stdev / mean * 100.

Each function repeats the test until rounded stdev becomes less than 3% in integer, that is, stdev/mean < 0.025.

The following are the hardware specifications of the personal computer used for evaluation.

```
CPU      : AMD FX(tm)-8300 Eight-Core Processor (64 bits)
L1 cache : 128KiB/core (64KiB for instruction + 64KiB for data)
L2 cache : 1MiB/core
L3 cache : 8MiB shared
Memory   : 16GiB
```

Another personal computer used for evaluation had the following specifications.

```
CPU      : AMD Phenom(tm) II X2 550 Two-Core processor (64 bits)
L1 cache : 128KiB/core (64KiB for instruction + 64KiB for data)
L2 cache : 512KiB/core
L3 cache : 6MiB shared
Memory   : 6GiB
```

The following chart shows the relative performance on the second personal computer with the copied executable program.

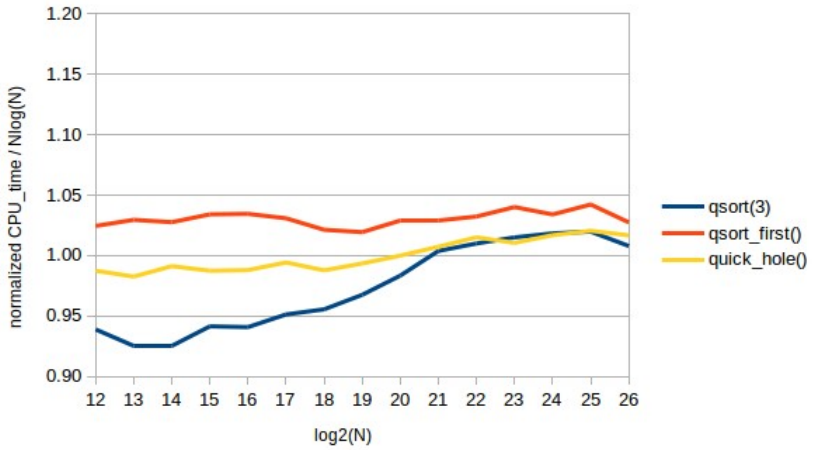

Fig. 2: Swap vs. Hole on another PC

Data for Fig.1 and Fig.2 - https://github.com/leorge/qmisort/wiki/data/Qsort/hole

The qsort_first() and quick_hole() series are not very different from the previous chart. Thus their relative performance does not depend on the hardware configuration. qsort(3) has a relatively quicker result compared with Figure 1, indicating that the relative performance of qsort(3) and new quicksort depends on the hardware configurations. Therefore, the series qsort(3) is a mere reference. All other charts in this paper were measured by the first computer as it has a larger main memory.

3. Secure quicksort

This section describes a known idea of R. Sedgewick, but this step is necessary to make quicksort secure.

If N is large, both qsort_first() and quick_hole() encounter stack overflow in the case of the sorted data presented below.

```
$ sorted.awk 12 | xargs echo   # sample of sorted data
00 01 02 03 04 05 06 07 08 09 10 11
$ sorted.awk 100000 > data     # make a sorted data in a file
$ for N in `seq 40000 10000 90000`; do Debug/Sort -N $N -fhV 1 data; done
arguments : -N 40000 -fhV 1 data
qsort(3)      usec = 8776     call = 0     compare = 298432     copy = 0
qsort_first() usec = 13581996 call = 39999 compare = 800059998  copy = 0
quick_hole()  usec = 11062354 call = 39999 compare = 799980000  copy = 79998
arguments : -N 50000 -fhV 1 data
qsort(3)      usec = 9799     call = 0     compare = 382512     copy = 0
qsort_first() usec = 21149308 call = 49999 compare = 1250074998 copy = 0
Segmentation fault (core dumped)
arguments : -N 60000 -fhV 1 data
qsort(3)      usec = 11746    call = 0     compare = 469008     copy = 0
qsort_first() usec = 30380979 call = 59999 compare = 1800089998 copy = 0
Segmentation fault (core dumped)
arguments : -N 70000 -fhV 1 data
qsort(3)      usec = 15002    call = 0     compare = 555200     copy = 0
qsort_first() usec = 41633101 call = 69999 compare = 2450104998 copy = 0
Segmentation fault (core dumped)
arguments : -N 80000 -fhV 1 data
qsort(3)      usec = 15597    call = 0     compare = 636864     copy = 0
qsort_first() usec = 53678405 call = 79999 compare = 3200119998 copy = 0
Segmentation fault (core dumped)
arguments : -N 90000 -fhV 1 data
qsort(3)      usec = 19712    call = 0     compare = 723680     copy = 0
Segmentation fault (core dumped)
```

sorted.awk : Awk script to generate a sequence of numbers.
seq : Linux command to print a sequence of numbers. The parameters are *`FIRST [INCREMENT] LAST`*.

qsort_hole() encounters stack overflow when N>=5000, and qsort_first() encounters it when N>=9000.

In this case, a partition is divided into 0 and N-1 elements because the largest/smallest element is chosen as the pivot. Thus, N in the longer sub-partition decreases by one such as N-1, N-2, N-3, … , 1; therefore, the number of calls is N-1. To call a function, parameters are passed via the stack area, and local variables in a function are also located in the stack area. Thus, the amount of working memory required is N-1 times the memory required for one function call. Therefore, recursive calls that are too deep exceed the limit of stack size and the function encounters stack overflow.

In the case of a random data sequence with depth of recursive calls **d**, the number of elements **n** in a partition is about $\mathbf{N/2^d}$. Thus, the average depth of recursive calls is about **log2(N)** because $\mathbf{n=1=N/2^d}$. Therefore, the depth of the shallowest recursion is less than log2(N), and so the shorter partition side of recursive calls has negligible impact; for example, log2(1073731824)=30. In contrast, in the case of the worst data sequence, the depth of the longer partition side is **N-1**, and the recursive call may encounter stack overflow. If the recursive call of the longer partition side is **converted** to iteration, no more stack area is required, avoiding stack overflow.

To convert recursion to iteration, the pseudocode is partially improved as below.

```
Quicksort(a[], lo, hi)
    WHILE lo < hi // IF → WHILE : change exit condition from recursive call to iteration
        p = Partition(a, lo, hi) // p is a position of the restored pivot
        IF p – lo < hi – p       // left partition is shorter than the right partition
            Quicksort(a, lo, p – 1)     // sort the left sub-partition recursively
            lo = p + 1   // instead of Quicksort(a, p+1, hi)
        ELSE      // left partition is not short than the right partition
            Quicksort(a, p + 1, hi)      // sort the right sub-partition recursively
            hi = p – 1   // instead of Quicksort(a, lo, p-1)
```

In the pseudocode, we replace the terminal condition of a recursive call "If lo < hi" with a condition of iteration "While lo < hi", and judge which is the shorter sub-partition after the partitioning p=Partition(a,lo,hi) where p is a boundary position between two sub-partitions. If the left sub-partition is shorter than the right sub-partition, we sort the left sub-partition recursively, and then move the starting position of the right sub-partition lo to p+1 to sort the right sub-partition in the next loop. Otherwise, if the left sub-partition is not shorter than the right sub-partition, we sort the right sub-partition recursively and then move the stopping position of the left sub-partition hi to p-1 to sort the left sub-partition in the next loop.

The following shows the effect of the new secured quicksort.

```
$ N=100000; sorted.awk $N | Debug/Sort -N $N -SV 1
arguments : -N 100000 -SV 1
qsort(3)       usec = 25520     call = 0     compare = 815024     copy = 0
quick_secure() usec = 68436980  call = 99999 compare = 4999950000 copy = 199998
```

quick_secure() : Secured quick_hole() called by the -S option, which is the implemented pseudocode above.

quick_secure() completes the sort despite the huge N. The number of calls is modified to be the added number of iterations and recursive calls.

The following chart shows the difference in relative CPU time between quick_secure() and quick_hole(). The Y axis is the normalized *CPU_time/n*log*(n)*, where the value of quick_hole() at N=2^20 is 1.

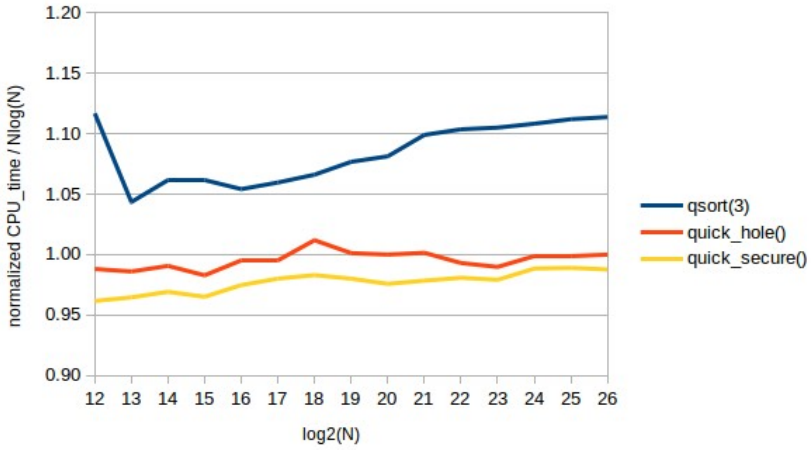

Fig. 3: Overhead of the secured quicksort

Data - https://github.com/leorge/qmisort/wiki/data/Qsort/secure

quick_ secure() is a bit faster than quick_hole(), and so the cost of iteration is slightly lower than for a recursive call.

quick_secure() avoids stack overflow, but the number of comparisons is still large: N(N-1)/2. This problem is resolved by the choice of pivot.

4. Asymmetric quicksort

The issue of many repeated elements in input data for quicksort is well known. Repeated equal elements is a type of this problem. As is well known, three-way partitioning resolves this issue by dividing a partition into lesser, equal and greater elements.

```
$ nnnn.awk 9 | xargs echo
9 9 9 9 9 9 9 9 9
$ N=100000; nnnn.awk $N | Debug/Sort -N $N -wSV 1
arguments : -N 100000 -wSV 1
qsort(3)       usec = 16622    call = 0     compare = 815024     copy = 0
qsort_3way()   usec = 1999     call = 1     compare = 99999      copy = 1
quick_secure() usec = 90008348 call = 99999 compare = 4999950000 copy = 199998
```

nnnn.awk : awk script to generate repeated equal data.
qsort_3way() : Three-way Partitioning Quicksort called by the -w option.

The numbers of comparisons, calls and copies in quick_secure() are equal to the case of sorted data above. qsort_3way() reduces these numbers; however, three-way partitioning is expensive in the case of random data sequence.

The following chart shows the relative CPU time. The Y axis is the normalized *CPU_time/n*log*(n)*, where the value of quick_hole() at N=2^20 is 1.

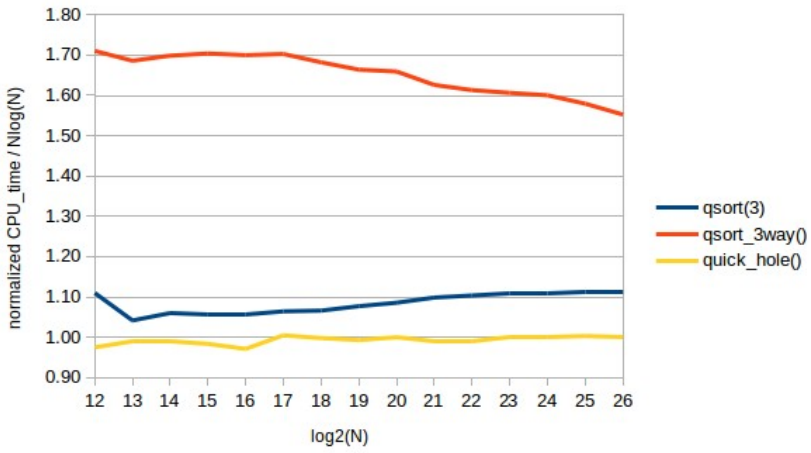

Fig. 4: 3-way partitioning is expensive

Data - https://github.com/leorge/qmisort/wiki/data/Qsort/3way

qsort_3way() is the slowest because the number of copies in qsort_3way() is the largest, as shown below.

```
$ N=100000; random.awk $N | Debug/Sort -N $N -hwV 1
arguments : -N 100000 -hwV 1
qsort(3)     usec = 65391 call = 0     compare = 1536226 copy = 0
qsort_3way() usec = 97441 call = 47820 compare = 1881709 copy = 5526483
quick_hole() usec = 59894 call = 69459 compare = 2027643 copy = 805984
```

Therefore, the object of three-way partitioning should be limited to the simple numeric array because copying an element by index, a[i] = a[j], is much faster than by memcpy(3).

I suggest another way to resolve this issue as follows. In partitioning, move the equal elements in the left partition to right, mark the start position at an equal element in the right partition, and clear the start position at a greater element. When the partitioning is complete, if the start position is marked, continuous equal elements can

be omitted from the right sub-partition. A partition is divided into 3 sub-partitions; lesser, equal, and greater **or equal** elements.

The following shows the improved pseudocode.

```
Quicksort(a[], lo, hi)
    WHILE lo < hi
        p, q = Partition(a, lo, hi)     // multiple return
        IF p – lo < hi – q
            Quicksort(a, lo, p – 1)
            lo = q + 1
        ELSE
            Quicksort(a, q + 1, hi)
            hi = p – 1

Partition(a[], lo, hi)
    hole = lo + (hi – lo) / 2    // choose the middle element as a pivot
    pivot = a[hole]
    a[hole] = a[hi]       // move the last element to the middle
    hole = hi--           // a hole is dug at the last position
    eq = -1               // initialize a variable to mark the start position
    WHILE lo < hole
        IF a[lo] >= pivot        // “>” is replaced with “>=”
            IF a[lo] > pivot         // a greater element
                eq = -1              //   clear the start position
            ELSE IF eq < 0           // first equal element
                eq = hole            //    mark a start position
            a[hole] = a[lo]
            hole = lo
            WHILE hi > hole
                IF a[hi] < pivot // a lesser element
                    a[hole] = a[hi]
                    hole = hi
                ELSE IF a[hi] > pivot    // a greater element
                    eq = -1              //   clear the start position
                ELSE IF eq < 0           // equal element again
                    eq = hi              //   mark a start position
                hi--
        lo++
    a[hole] = pivot
    RETURN hole, eq < 0 ? hole : eq
```

The outer loop and inner loop are asymmetric, and two sub-sub-partitions are sorted asymmetrically by iteration and recursion. For this reason, I have named this algorithm **Asymmetric Quicksort**.

In the case of repeated equal elements, the outer loop stops at the first element, and in the inner loop, eq stays at the initial hi, and hi reaches lo. Thus, Partition() returns lo and initial hi. Then, the number of elements in the sub-partitions are zero.

The following list shows the effect.

```
$ N=100000; nnnn.awk $N | Debug/Sort -N $N -wSaV 1
arguments : -N 100000 -wSaV 1
qsort(3)       usec = 18385    call = 0     compare = 815024     copy = 0
qsort_3way()   usec = 1890     call = 1     compare = 99999      copy = 1
quick_secure() usec = 88903076 call = 99999 compare = 4999950000 copy = 199998
quick_asymm()  usec = 1776     call = 1     compare = 99999      copy = 4
```

quick_asymm() : Imprelented pseudocode above called by the -a option.

In quick_asymm(), the number of comparisons is N-1; thus, this case is resolved. Other many repeated elements situations are also resolved, as shown below.

Various patterns of two data:

```
$ N=10000; for a in n111 n11n n1n1 n1nn nn11 nn1n nnn1; do
> echo ""; echo "$a.awk : `$a.awk 12`" | xargs echo
> $a.awk $N | Debug/Sort -N $N -haV 1; done

n111.awk : 12 01 01 01 01 01 01 01 01 01 01 01
arguments : -N 10000 -haV 1
qsort(3)      usec = 2888   call = 0    compare = 74594    copy = 0
quick_hole()  usec = 732612 call = 9999 compare = 49995000 copy = 19999
quick_asymm() usec = 199    call = 1    compare = 9999     copy = 4

n11n.awk : 12 01 01 01 01 01 01 01 01 01 01 12
arguments : -N 10000 -haV 1
qsort(3)      usec = 2041   call = 0    compare = 74594    copy = 0
quick_hole()  usec = 725638 call = 9999 compare = 49995000 copy = 19999
quick_asymm() usec = 427    call = 14   compare = 19990    copy = 56

n1n1.awk : 12 01 12 01 12 01 12 01 12 01 12 01
arguments : -N 10000 -haV 1
qsort(3)      usec = 4564   call = 0    compare = 96164    copy = 0
quick_hole()  usec = 717952 call = 7500 compare = 37502499 copy = 17500
quick_asymm() usec = 373    call = 2    compare = 14998    copy = 5007

n1nn.awk : 12 01 12 12 12 12 12 12 12 12 12 12
arguments : -N 10000 -haV 1
qsort(3)      usec = 2679   call = 0    compare = 64608    copy = 0
quick_hole()  usec = 750931 call = 9999 compare = 49995000 copy = 19999
quick_asymm() usec = 163    call = 1    compare = 9999     copy = 5

nn11.awk : 12 12 12 12 12 12 01 01 01 01 01 01
arguments : -N 10000 -haV 1
qsort(3)      usec = 5400   call = 0    compare = 64608    copy = 0
quick_hole()  usec = 713505 call = 9999 compare = 49995000 copy = 24998
quick_asymm() usec = 934    call = 5    compare = 49983    copy = 10014

nn1n.awk : 12 12 12 12 12 12 12 12 12 01 12
arguments : -N 10000 -haV 1
qsort(3)      usec = 2002   call = 0    compare = 64621    copy = 0
quick_hole()  usec = 709358 call = 9999 compare = 49995000 copy = 19999
quick_asymm() usec = 185    call = 1    compare = 9999    copy = 6

nnn1.awk : 12 12 12 12 12 12 12 12 12 12 12 01
arguments : -N 10000 -haV 1
qsort(3)      usec = 2177   call = 0    compare = 64621    copy = 0
quick_hole()  usec = 756220 call = 9999 compare = 49995000 copy = 19999
quick_asymm() usec = 191    call = 1    compare = 9999     copy = 6
```

Shuffled two data :

```
$ N=10000; nn11.awk $N | shuf | Debug/Sort -N $N -haV 1
arguments : -N 10000 -haV 1
qsort(3)      usec = 2970   call = 0    compare = 94677    copy = 0
quick_hole()  usec = 549488 call = 8761 compare = 37552309 copy = 20014
quick_asymm() usec = 400    call = 2    compare = 14998    copy = 5033
```

shuf : Linux command to generate random permutations.

Shuffled three data :

```
$ N=10000; for i in `seq 1 $N`; do echo 11111; echo 55555; echo 99999
> done | shuf | Debug/Sort -N $N -haV 1
arguments : -N 10000 -haV 1
qsort(3)      usec = 2911   call = 0    compare = 103721   copy = 0
quick_hole()  usec = 276857 call = 8900 compare = 18265579 copy = 23416
quick_asymm() usec = 632    call = 5    compare = 33005    copy = 33005
```

Shuffled 100 data 100 times :

```
$ N=10000; for i in `seq 0 99`; do for j in `seq -w 0 99`; do echo $j; done
> done | shuf | tee data | Debug/Sort -N $N -whaV 1
arguments : -N 10000 -whaV 1
qsort(3)      usec = 3960 call = 0    compare = 120217 copy = 0
qsort_3way()  usec = 2846 call = 100  compare = 70300  copy = 181228
quick_hole()  usec = 8384 call = 8799 compare = 493738 copy = 46887
quick_asymm() usec = 2393 call = 194  compare = 100086 copy = 31642
$ for i in `seq 1 15`; do shuf data | Debug/Sort -N $N -hV 1
> done | awk '/hole/{print $7}' | xargs echo  # the number of calls in quick_hole()
8750  8782  8761  8835  8783  8846  8794  8821  8804  8793  8809  8790  8751  8793  8813
$ for i in `seq 1 15`; do shuf data | Debug/Sort -N $N -aV 1
> done | awk '/asymm/{print $7}' | xargs echo # the number of calls in quick_asymm()
206 190 192 182 198 207 197 183 186 195 192 192 201 193 196
```

tee : Linux command to read from standard input and write to standard output and files.

The number of calls in qsort_3way() equals the kind of data value. Thus, 3-way partitioning is most efficient. The second and third commands output the number of calls in quick_hole() and quick_asymm(). quick_hole() has almost 8800 calls, which is much greater than for qsort_3way(). In contrast, quick_asymm() has about 200 calls, but this double the number for qsort_3way(), and efficient enough for this case.

5. Random choice

If a pivot is chosen from several fixed positions, it is possible to generate malicious data sequence, which makes the time complexity quadratic. As is well known, a random choice of pivot avoids this possibility.

The following list demonstrates the case[4] of median-of-five.

```
$ for N in 10000 20000 40000; do
> KillDualPivot.pl $N | Debug/Sort -N $N -jV 1; done
arguments : -N 10000 -jV1
qsort(3)     usec = 3051     call = 0    compare = 105743    copy = 0
dual_pivot() usec = 408774   call = 2453 compare = 12190378  copy = 18239305
arguments : -N 20000 -jV1
qsort(3)     usec = 13377    call = 0    compare = 227530    copy = 0
dual_pivot() usec = 1598094  call = 4953 compare = 49377878  copy = 73974305
arguments : -N 40000 -jV1
qsort(3)     usec = 17261    call = 0    compare = 486575    copy = 0
dual_pivot() usec = 6434579  call = 9953 compare = 198752878 copy = 297944305
```

KillDualPivot.pl[5] : A perl script to generate a worst data sequence for Dual-Pivot Quicksort in Java 7.
Perl : The Perl 5 language interpreter.
dual_pivot() : Dual-Pivot Quicksort called by the -j option.

The numbers of comparisons and copies increase quadratically; thus, the generated data sequence is the worst for Dual-Pivot Quicksort.

The template of dual_pivot() is DualPivotQuicksorot.java in the library of Java 7, which is a hybrid sorting of Dual-Pivot Quicksort, Three-way Partitioning Quicksort, Pair Insertion Sort and Linear Insertion Sort. **Dual-Pivot Quicksort**[3], invented by Vladimir Yaroslavskiy, picks up 5 elements at nearly 3N/14, 5N/14, 7N/14, 9N/14, 11N/14 in a partition, and chooses the **second** and **fourth** large elements from them as a small pivot and large pivot to divide a partition to three sub-partitions. The first sub-partition consists of elements smaller than the small pivot, the last sub-partition consists of elements greater than the large pivot, and the middle sub-partition consists of all other elements, i.e. elements greater than or equal to the small pivot and smaller than or equal to the large pivot. **KillDualPivot.pl** puts the **four** largest elements at the picking up positions recursively. Therefore, the fourth largest element will be chosen as a small pivot, and the second largest element will be chosen as a large pivot recursively. When a partitioning is completed, the middle sub-partition consists of only the third largest element, and the last sub-partition consists of only the largest element. Thus, the **four** largest elements are gathered at the last part of the partition. Other elements in the first sub-partition will be sorted recursively, and the number of

4 Other examples - https://github.com/leorge/qmisort/wiki/Malicious-data.
5 Source code is secret until the problem is resolved in Java.

elements passed to the recursive call decreases by **four**, so the depth of recursive call is about N/4 as above. Finally, dual_pivot() encounters stack overflow if N is large.

```
$ N=50000; KillDualPivot.pl $N | Debug/Sort -N $N -jV 1
arguments : -N 50000 -jV 1
qsort(3) usec = 26761 call = 0 compare = 622340 copy = 0
Segmentation fault (core dumped)
```

As discussed above, a random choice of pivot resolves this problem.

```
$ N=50000; KillDualPivot.pl $N | Debug/Sort -N $N -rV 1
arguments : -N 50000 -rV 1
qsort(3)       usec = 19443 call = 0     compare = 622340 copy = 0
quick_random() usec = 28669 call = 33247 compare = 970199 copy = 416577
```

quick_random() : Asymmetric Quicksort called by the -r option. The pivot is a random element entirely.

quick_random() avoids malicious data sequence as above. However, randomization is considered **expensive**.

The following chart shows the overhead of rand(3), which is a function to generate a pseudorandom number. quick_random() chooses a random element, and quick_asymm() chooses the middle element. The Y axis is the normalized $CPU\_time/n\log(n)$, where the value of quick_hole() at N=2^20 is 1.

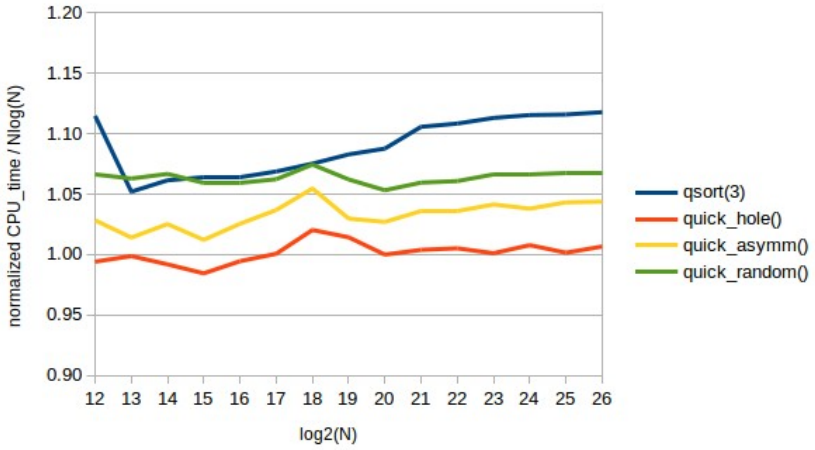

Fig. 5: Overhead of randomization

data - https://github.com/leorge/qmisort/wiki/data/Qsort/random

Since quick_random() is slower than quick_asymm(), the cost of rand(3) is expensive. However it is possible to reduce the cost as described below.

The following list shows the distribution of N in sub-partitions for various N. The index is log2(N), and the value is the number of calls.

```
$ for N in 1000 10000 100000; do random.awk $N |
> Debug/Sort -N $N -rV 2 | src/nmemb.awk; done
log2(1000)=9    1   2   3   4  5  6  7 8 9 sum
quick_random()  215 162 104 56 29 14 9 6 2 597
log2(10000)=13  1    2    3    4   5   6   7  8  9  10 11 12 13 sum
quick_random()  1997 1655 1011 565 286 150 73 52 21 10 3  2  2  5827
log2(100000)=16 1     2     3     4    5    6    7   8   9   10 11 12 13 14 15 16 sum
quick_random()  20717 16348 10264 5577 3087 1495 790 401 199 94 51 26 11 5  2  4  59071
```

nmemb.awk : awk script to show the distribution of N

The summations of calls is about **0.6N**, and rand(3) is called once in each call, thus the time complexity of randomization is *O*(n) in big O notation. *O*(*n*) is smaller than the time complexity of the sorting algorithm *O*(*n* log(*n*)). Therefore, the randomization is comparatively **inexpensive** when N is large.

The sum of first 3 values, N<16, is about 80% of calls.

```
(215+162+104)/597=0.81    (1997+1655+1011)/5827=0.80    (20717+16348+10264)/59071=0.80
```

Thus, rand(3) will be called about 0.1N (~ 0.6N * 0.2) times if rand(3) is not called when N<16.

The following chart shows the performances of the following three cases of choosing pivot: the pivot is the middle element, the pivot is an entirely random element, and the pivot is a random element while N>=16 or else the middle element. The Y axis is the normalized *CPU_time/n*log*(n)*, where the value of quick_hole() at N=2^20 is 1.

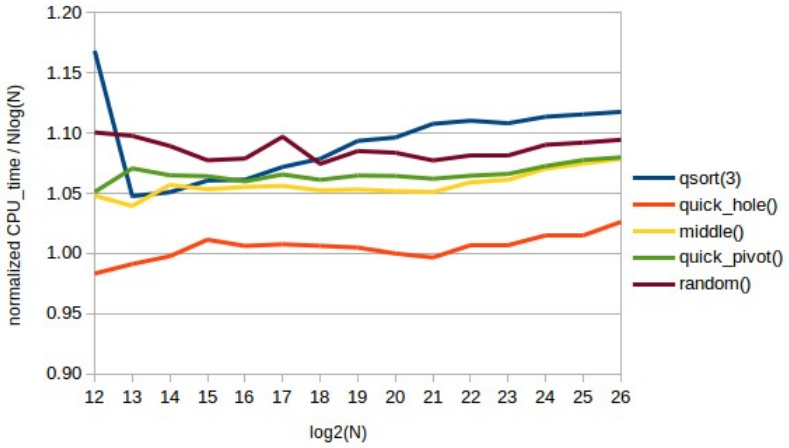

FIg. 6: Random and middle choice of pivot

quick_pivot() : Asymmetric quicksort to estimate performances of various choosing method of a pivot, called by the -t option. -P option set the threshold number to change the choice.
middle() : alias of quick_pivot() which chooses the middle element as a pivot.
random() : alias of quick_pivot() which entirely chooses a random element as a pivot.

data - https://github.com/leorge/qmisort/wiki/data/Qsort/pivot

quick_pivot() converged in middle(), so the cost of randomization is **negligible**.

6. Median

The choice of the median of several elements as a pivot is more efficient than the choice of a single element. In the case of median-of-three elements, I suggest choosing the first element at random in the range [0, N/2), with the distance between the other elements equal to N/4. Thus, the second element is in the range [N/4, 3N/4), and the last element is in [N/2, N). In the case of median-of-five elements, I suggest choosing the first element at random in [0, N/4), with the distance between other elements equal to 3N/16. As such, the ranges of the other elements are [3N/16, 7N/16), [6N/16, 10N/6), [9N/16, 13N/16), [12N/16, 16N/16).

I also suggest a median-of-logarithmic, where the logarithmic number L is defined as **`(int(log2(N))/2)|1`**. The first element is in the range [0, N/L), and the distance between the other elements is N/L. To get the median, make an array of pointers to the elements, sort it partially including the middle of array, and choose the middle pointer.

The following pseudocode demonstrates how to choose the middle pointer in the median-of-logarithmic.

```
Median_of_Logarithmic(idx[])
    lo = 0
    hi = idx.length - 1
    middle = hi / 2             // or idx.length/2
    WHILE lo < hi
        p = Partition(idx, lo, hi)  // pivot is the middle element
        IF p > middle     // the middle pointer is in the left partition
            hi = p - 1
        ELSE IF p < middle       // in the right partition
            lo = p + 1
        ELSE      // the middle pointer is found
            BREAK
    RETURN p
```

When N is decreased, a pivot element should be chosen at the middle of a partition against sorted data. Further, the threshold of N to change from median-of several elements to the middle should be determined by experiment.

The following chart shows the CPU time under various thresholds. The Y axis is the normalized *CPU_time/n*log*(n)*, where the average of quick_hole() is 1. The series in the chart are averages of 20 random sequences. N=2^24.

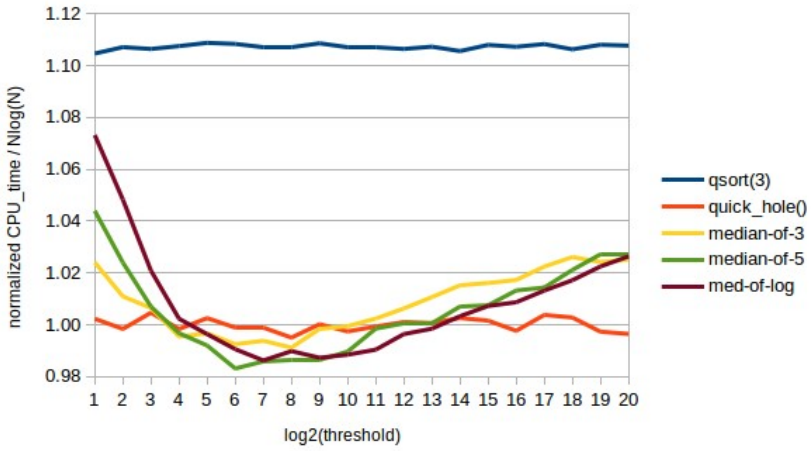

Fig. 7: Threshold number of several median-of (N=2^24)

Data for Fig.7 and Fig.8 - https://github.com/leorge/qmisort/wiki/data/Qsort/threshold

At the leftmost of the series are the choices of the median-of several elements entirely, which shows their relative computational complexity. The minimum of the series are the best thresholds, but they are difficult to find precisely.

The following chart shows the case of N=2^20. To smooth the results, the series in the chart are averag es of 255 random sequences.

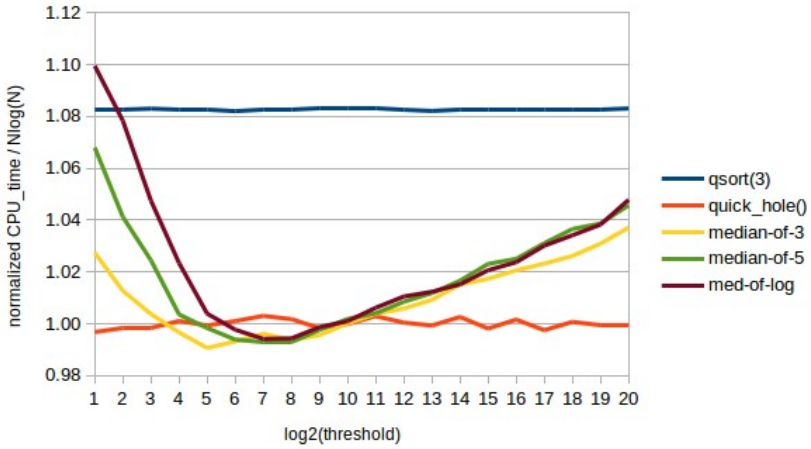

Fig. 8: Threshold number of several median-of (N=2^20)

Data for Fig.7 and Fig.8 - https://github.com/leorge/qmisort/wiki/data/Qsort/threshold

I determined the threshold as shown below.

type	threshold
median-of-3	2^5
median-of-5	2^6
median-of-L	2^7

The following chart shows the CPU time of several median-of in various N. The Y axis is the normalized *CPU_time/*`nlog`*(n)*, where the value of quick_hole() at N=2^20 is 1. The series in the chart are averages of 20 random sequences.

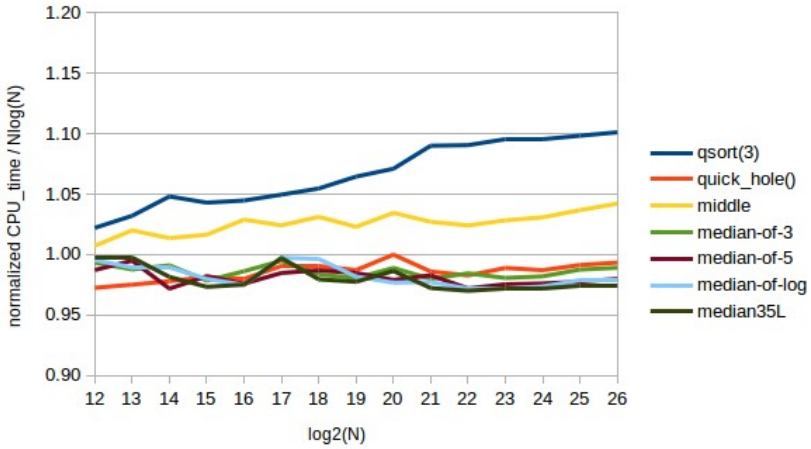

Fig. 9: Median-of several elements

middle : Pivot is the middle element.
median35L : Pivot is the middle element when N<=31, the median-of three when N<=127, the median-of five when N<=4095 else the median-of-logarithmic because `(int(log2(4095))/2|1=5` and `(int(log2(4096))/2|1=7.`

Data - https://github.com/leorge/qmisort/wiki/data/Qsort/medians

Median-of-3 is a bit slow whereas median-of-5, median-of-log and median35L are similar when N>2^21. In particular, median35L(),which is a multiple median-of, is the fastest by a small margin.

7. Conclusion

Asymmetric Quicksort prevents stack overflow, resolves the issue of repeated many elements, and avoids any other malicious data sequences. Therefore, Asymmetric Quicksort is secure.

Asymmetric Quicksort becomes faster through the use of multiple median-of. The following chart shows the CPU time of the final Asymmetric Quicksort for various N. The Y axis is the normalized $CPU\_time/n\log(n)$, where the value of quick_hole() at N=2^20 is 1. The series in the chart are averages of 20 random sequences.

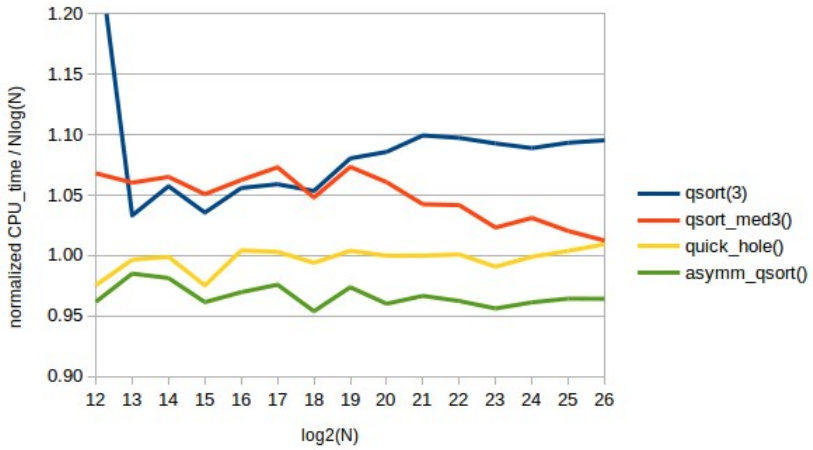

Fig. 10: Final Asymmetric Quicksort

qsort_med3() : Conventional median-of-3 quicksort called by the -3 option.
asymm_qsort() : Final asymmetric quicksort, called by the -q option, is like median35L above.
However, this uses the conventional median-of-three to reduce the use of rand(3).

Data - https://github.com/leorge/qmisort/wiki/data/Qsort/asymmetric

Asymmetric Quicksort is about 5% faster than the conventional quicksort with the median-of-3 and swaps.

This paper described entire quicksort, and the next theme for investigation is a hybrid sorting algorithm to make quicksort much faster.

8. Experimental results

Data and charts : https://www.dropbox.com/s/lp5xpxndesb5396/Qsort.ods

9. References

1. Quicksort: https://en.wikipedia.org/wiki/Quicksort, http://algs4.cs.princeton.edu/23quicksort
2. Stephanie Bell. A Beginner's Guide to Uncertainty of Measurement
3. Vladimir Yaroslavskiy. Dual-Pivot Quicksort algorithm

Appendix A. hole_qsort.c

The following list shows the source code of the simplest new Quicksort using a pivot hole.

```
#include <string.h>

#define copy(a, b)  memcpy((a), (b), length)

static int      (*comp)(const void *, const void *);
static size_t   length;

static void sort(void *base, size_t nmemb) {
    if (nmemb <= 1) return;
#define first (char *)base
    char *last = first + (nmemb - 1) * length;  // point the last element
    char pivot[length], *hole; copy(pivot, hole = last);    // save the last element as a pivot
    char *lo = first, *hi = last - length;  // search pointers
    for (; lo < hole; lo += length) {       // outer loop
        if (comp(lo, pivot) > 0) {
            copy(hole, lo); hole = lo;      // exchange a larger element with the hole
            for (; hi > hole; hi -= length) {   // inner loop, symmetric to the outer loop
                if (comp(hi, pivot) < 0) {      // symmetric comparison
                    copy(hole, hi); hole = hi;  // exchange a smaller element with the hole
                }
            }
        }
    }
    copy(hole, pivot);  // restore the pivot
    sort(first, (hole - first) / length);           // smaller elements
    sort(hole + length, (last - hole) / length);    // larger elements
}

void hole_qsort(void *base, size_t nmemb, size_t size, int (*comp)(const void *, const void *))
{
    length = size; comp = comp;
    sort(base, nmemb);
}
```

Appendix B. asymm_qsort.c

The following list shows the source code of the final Asymmetric Quicksort.

```
#include <math.h>
#include <stdlib.h>
#include <string.h>


#define copy(a, b)  memcpy((a), (b), length)
#define MIDDLE      63      // Choose the middle element as a pivot when N <= MIDDLE
#define MEDIAN3     127     // Median-of-3
#define MEDIAN5     4095    // Median-of-5

static int      (*comp)(const void *, const void *);
static size_t   length;

static void sort(void *base, size_t nmemb) {
    while (nmemb > 1) {
        char *hole, *first = (char *)base, *last = first + (nmemb - 1) * length;
        // choose a pivot element
        if (nmemb <= MIDDLE) {  // middle element
            hole = first + (nmemb >> 1) * length;
        } else if (nmemb <= MEDIAN3) {  // conventional median-of-3
            char *middle = first + (nmemb >> 1) * length;
            hole = (comp(first, last)   < 0 ?
                   (comp(middle, first) < 0 ? first: (comp(middle, last)  < 0 ? middle : last)) :
                   (comp(middle, last)  < 0 ? last: (comp(middle, first) < 0 ? middle : first)));
        } else if (nmemb <= MEDIAN5) {  // median-of-5
            char *p1, *p2, *p3, *p4, *p5, *tmp;
            p1 = (char *)base + (((nmemb >> 2) * rand()) / ((size_t)RAND_MAX + 1)) * length;
            size_t distance = ((nmemb >> 3) + (nmemb >> 4)) * length;   // N/8 + N/16 = 3N/16
            p5 = (p4 = (p3 = (p2 = p1 + distance) + distance) + distance) + distance;
            if (comp(p2, p4) > 0) {tmp = p2; p2 = p4; p4 = tmp;}        // *p2 <= *p4
            if (comp(p3, p2) < 0) {tmp = p2; p2 = p3; p3 = tmp;}
            else if (comp(p4, p3) < 0) {tmp = p4; p4 = p3; p3 = tmp;}   // *p2 <= *p3 <= *p4
            if (comp(p1, p5) > 0) {tmp = p1; p1 = p5; p5 = tmp;}        // *p1 <= *p5
            hole =  comp(p3, p1) < 0 ? (comp(p1, p4) < 0 ? p1 : p4)
                 : (comp(p5, p3) < 0 ? (comp(p5, p2) < 0 ? p2 : p5) : p3);
        } else {    // median-of log2(sqrt(N))|1 random elements
            size_t  pickup = ((size_t)log2(nmemb) >> 1) | 1;        // number of elements
            void *index[pickup];
            char *p = first + (nmemb * rand() / ((size_t)RAND_MAX + 1) / pickup) * length;
            size_t  distance = (size_t)(nmemb / pickup) * length;   // distance of elements
            for (size_t idx = 0; idx < pickup; p += distance) index[idx++] = p;
            void **left = index, **right = &index[pickup -1], **middle = &index[pickup >> 1];
            while (left < right) {  // search a pointer to the middle element
                void **phole = &left[(right - left) >> 1];  // hole in the index
                char *pivot = *phole;       // save the middle pointer
                *phole = *right;    // move the last pointer to the middle of index
                phole = right;      // dig a hole at the last of index
                void **plo = left, **phi = right - 1;
                for (int chk; plo < phole; plo++) {
                    if ((chk = comp(*plo, pivot)) >= 0) {
                        *phole = *plo; phole = plo;
                        for (; phi > phole; phi--) {
                            if ((chk = comp(*phi, pivot)) < 0) {
                                *phole = *phi; phole = phi;
                            }
                        }
                    }
                }
                *phole = pivot;     // restore
                if (middle < phole) right = phole - 1;
                else if (phole < middle) left = phole + 1;
                else break;         // phole == middle
            }
            hole = *middle;     // hole is in the middle of index[]
```

```
        }
        // partition
        char save[length];    // pivot element
        copy(save, hole); copy(hole, last);    // save <-- hole <-- last
        char *lo = first,  *hi = (hole = last) - length, *eq = NULL;
        for (int chk; lo < hole; lo += length) {
            if ((chk = comp(lo, save)) >= 0) {
                if (chk > 0) eq = NULL; // discontinued
                else if (eq == NULL) eq = hole;
                copy(hole, lo); hole = lo;
                for (; hi > hole; hi -= length) {
                    if ((chk = comp(hi, save)) < 0) {
                        copy(hole, hi); hole = hi;
                    }
                    else if (chk > 0) eq = NULL;
                    else if (eq == NULL) eq = hi;   // first equal element
                }
            }
        }
        if (eq == NULL) eq = hole;
        copy(hole, save);  // restore

        // sort sub-arrays recursively and iteratively.
        size_t  n_lo = (hole - first) / length; // number of element in the left sub-partition
        size_t  n_hi = (last - eq) / length;
        if (n_lo < n_hi) {
            sort(base, n_lo);  // sort the shorter sub-partition first
            nmemb = n_hi; base = eq + length;
        } else {
            sort(eq + length, n_hi);
            nmemb = n_lo;
        }
    }
}

void asymm_qsort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void
*))
{
    length = size; comp = compare;
    sort(base, nmemb);
}
```