Indirect quicksort and mergesort

Takeuchi Leorge (竹内 良治)
qmisort@gmail.com

Abstract

This paper estimates indirect quicksort and mergesort for various sizes of array element, and suggests a data structure to guarantee time complexity of O(n log(n)) in C language.

1. Introduction

Sorting algorithms reorder an array in some logical order, such as numerical order or lexicographical order. Quicksort[1], invented by Tony Hoare in 1959, and mergesort[2], invented by John von Neumann in 1945, are the fastest representative sorting algorithms. A sorting algorithm in C language reorders continuous array elements, so the cost to move elements increases with the size of the elements. To reduce the cost, an indirect sorting algorithm is conventionally used. However, its time complexity is actually greater than $O(n\log(n))$ in big O notation because of cache memories.

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

2. Quicksort vs. mergesort

This section describes the algorithms of quicksort and mergesort and mentions their benefits and disadvantages.

Quicksort chooses and saves an array element as a pivot. Then, it iterates exchanging an element not smaller than the pivot in the lower position for an element not larger in the higher position, and restores the pivot at the boundary of these two elements. Thus, this operation divides the array into two sub-arrays[1] omitting the pivot: to the left of the pivot are elements not larger than the pivot, and to the right are elements not smaller. The destination of equal elements and the choice of pivot both depend on the implementation. This operation is applied to two sub-arrays recursively until every sub-array contain one or no elements. When all of the operations are completed, the array is sorted.

The simplest pseudocode of Asymmetric Quicksort[3], invented in 2016, is as follows:

```
Quicksort(a[], lo, hi)        // sort a[lo..hi]
    IF lo < hi       // termination condition
        p = Partition(a, lo, hi)       // divide an array to two sub-arrays
        Quicksort(a, lo, p – 1)        // sort smaller elements recursively
        Quicksort(a, p + 1, hi)        // sort larger or equal elements recursively
```

1 One of the sub-arrays may be empty if the pivot is the smallest or largest element.

```
Partition(a[], lo, hi)
    pivot = a[hi]             // save the last array element as a pivot
    hole = hi--               // dig a hole
    WHILE lo < hole
        IF a[lo] >= pivot
            a[hole] = a[lo]   // move a larger or equal element to a higher position
            hole = lo         // move the hole
            WHILE hi > hole
                IF a[hi] < pivot
                    a[hole] = a[hi]    // move a smaller element to a lower position
                    hole = hi
                hi--
        lo++
    a[hole] = pivot  // restore the pivot
    RETURN hole      // return the boundary position of the two sub-arrays
```

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 the pivot. Then, - denotes a hole.
(-, A, D, C)  B       Search for an element larger than or equal to B from the first
                      position. We find C and move it to the hole.
(A, -, D, C)  B       Search for a smaller element from the position before C.
                      We find A and move it 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.
```

Top-down mergesort[2] halves the input array recursively until each sub-array contains one element, and then repeatedly merges two sub-arrays to a sorted array to obtain the sorted input array. Conventional mergesort initially duplicates the input array and alternately uses the original and the copy for the merging array. Since the duplication requires memory in the same size as the input array, the heap memory overflows if the input data is huge, which limits the size of the input array to about half of that for most sorting algorithms. This is a **disadvantage** of mergesort.

The simplest pseudocode is presented below.

```
Mergesort(a[], lo, hi)        // sort an array a[lo..hi]
    Sort(a, a.duplicate(), lo, hi)   // duplicate the array before sorting

Sort(dst[], src[], lo, hi)    // merge src[lo..hi] to dst[lo..hi]
    IF lo < hi
        mid = lo + (hi – lo) / 2               // middle position
        Sort(src, dst, left = lo, mid)         // Sort the left sub-array a[lo..mid]
        Sort(src, dst, right = mid + 1, hi)    // Sort the right sub-array a[(mid+1)..hi]
        LOOP      // merge src[lo..mid] and src[(mid+1)..hi] to dst[lo..hi]
           IF src[left] <= src[right]
              dst[lo++] = src[left++]          // append a not-larger element to the merging array
              IF left > mid                    // the left sub-array is empty
                 src.copy(dst, lo, right, hi)  // append src[right..hi] to dst[lo..]
                 BREAK
           ELSE
              dst[lo++] = src[right++]         // append a larger element
              IF right > hi                    // the right sub-array is empty
                 src.copy(dst, lo, left, mid)  // append src[left..mid] to dst[lo..]
                 BREAK
```

2 Bottom-up implementation is another algorithm of mergesort.

The following example demonstrates the behavior of this process.

```
A: ( Y ,  Z, X )          Duplicate and divide the input array A into B.
B: ((Y), (Z, X))

A: ( - , ((Z), (X)))      Divide (Z, X) in B to (Z) and (X) in A.
B: ((Y), ( - ,  - ))      - is regarded as an empty element.

A: ( - , ((Z), (-))       Move X from A into B.
B: ((Y), ( X ,  -))

A: ( - , ((-), (-))       Append (Z) from A to (X) in B.
B: ((Y), ( X ,  Z))       Z and X from A are merged to (X, Z) in B.

A: ( X ,  -, - )          Move X from B to A.
B: ((Y), (-, Z))

A: ( X ,  Y, - )          Append Y from B to (X) in A.
B: ((-), (-, Z))

A: ( X ,  Y, Z )          Append (Z) from B to (X, Y) in A.
B: ((-), (-, -))          The input array is sorted.

A: (X, Y, Z)              Discard B.
```

The following chart shows the relative CPU time consumed by quicksort and mergesort for various N. The Y-axis is the normalized `CPU_time/nlog(n)`, where the value of asymm_qsort() is 1 at N=2^{20}=1M. The function strcmp(3) is used here to compare strings in two array elements. The size of an element is 16 bytes. To reduce data dependency, the series the chart is based on is the average of 10 random sequences, and to reduce measurement error, each data point is the mean of 10 CPU times. Thus, the value of a point is the average of 100 (=10x10) CPU times. The test program is written in C language with Eclipse 3.8.1 on Linux Mint. Eclipse generates two executables to debug and to release. The consumed CPU times[3] are measured in a release build program on Live[4] ArchLinux for 64bit CPU.

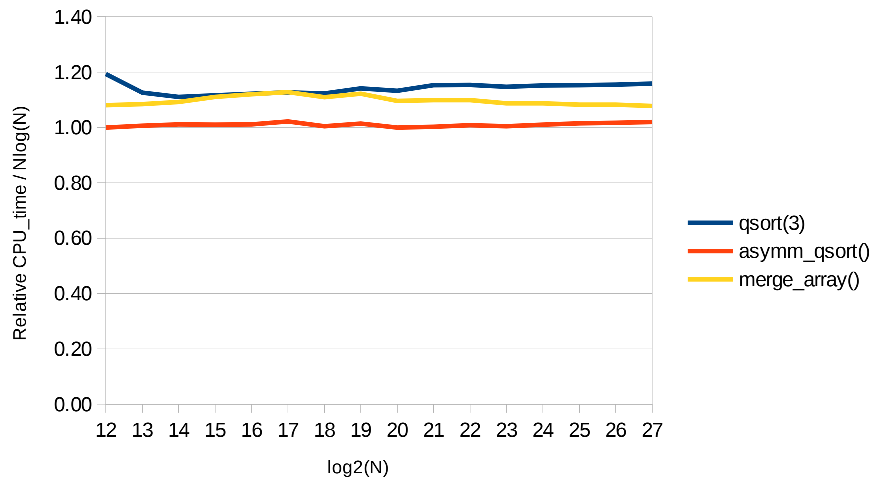

Fig. 1: Relative CPU times for quicksort and mergesort with various N (16 bytes/element).

3 The consumed CPU time of a process is measured by the function clock_gettime(2).

4 It is very lightweight since it has no Windows system.

The following list explains the functions evaluated.

qsort(3): A function in GNU C library to sort an array.
  Section 3 enclosed in parentheses refers to Library calls. This series is a mere reference.
  The version of this library is shown in the following list.
asymm_qsort(): Asymmetric quicksort called by the -q option.
merge_array(): Implemented pseudocode of mergesort called by the -m option.

asymm_qsort() is the fastest algorithm. To demonstrate why, the following list shows some numbers of times measured in the debug build program.[5] The time complexity depends on the number of comparisons. The number of copies is derive from the comparisons, wasting the CPU time, since during a comparison, the array elements are moved by a function, memcpy(3), which is more expensive than a substitution A[i] = A[j]. However, the CPU times following “usec =” are inaccurate because the debug build program executes many debugging statements according to the tracing level.

```
$ 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 -mqV 1    # Sort 100,000 elements
arguments: -N 100000 -mqV 1
qsort(3)      usec = 53983 call = 0     compare = 1536227 copy = 0
asymm_qsort() usec = 55314 call = 59178 compare = 1689458 copy = 889049
merge_array() usec = 53865 call = 99999 compare = 1536227 copy = 1636226
$ /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 the 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.
-N *xx*: Command option to set the number of elements in the input array.
-V x: Tracing level to debug.
head: Linux command to output the first part of files.

The number of comparisons in merge_array() is about **10%** less than that in asymm_qsort(). This is an **advantage** of mergesort over quicksort. However, the number of copies in asymm_qsort() is about **half** of that in merge_array(). This is an **advantage** of quicksort over mergesort, and the reason quicksort is sometimes faster than mergesort. Mergesort copies an element after comparison and appends a remaining sub-array to the merging array as another sub-array empties in every call. Thus, the number of copies is the sum of comparisons and calls[6]: `99999+1536227=1636226`. On the other hand, quicksort copies an element with probability **0.5** after comparison, so the number of copies is about half of the comparisons, even though quicksort both compares elements to choose a pivot and copies elements to save and restore a pivot in every call.

Therefore, if the size of an array element is huge, mergesort becomes much slower than quicksort. Fig. 2 demonstrates this for the case of 256 bytes/element using the same functions as in Fig. 1. The scale of the Y-axis is the same as Fig. 1; consumed CPU time of asymm_qsort() is 1 at N=2^{20} when the size of an element is 16.

5 The number of calls and copies in qsort(3) are hidden.

6 The number of calls in mergesort is N-1 since every call inserts a virtual gap between adjacent elements.

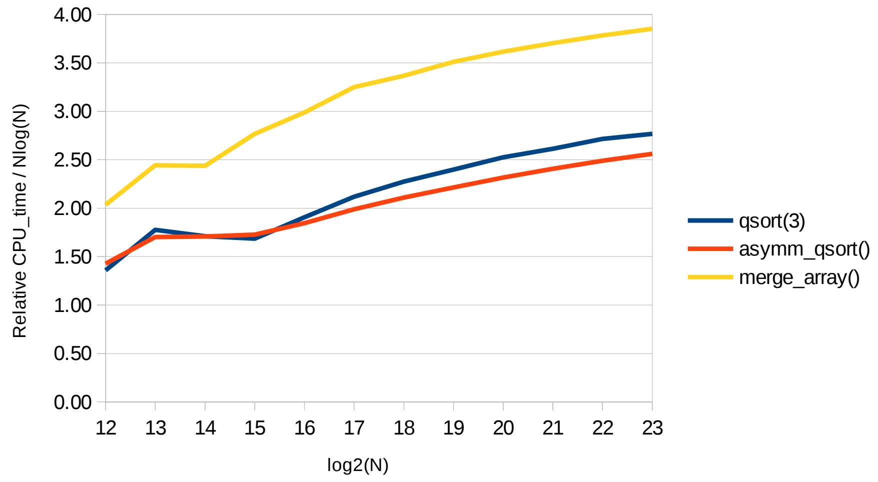

Fig. 2: Relative CPU times for quicksort and mergesort with various N (256 bytes/element).

All of the series slow down compared with Fig. 1, especially merge_array(), as expected.

The time complexity of quicksort and mergesort are $O(n\log(n))$[7] in the case of a random data sequence. so the series in this chart should be nearly flat as in Fig. 1. However, asymm_qsort() bends upward when $N>2^{15}$, and merge_array() bends upward when $N>2^{14}$. That is, their time complexities are actually greater than $O(n\log(n))$. A key reason for this is the size of the **cache memories**[8]

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

```
CPU        AMD FX(tm)-8300 Eight-Core Processor (64bits)
L1 cache   128KiB[9] / core (64KiB for instruction + 64KiB for data)
L2 cache   2MiB / 2core
L3 cache   8MiB shared
Memory     16GiB DDR3-1330MHz
```

Cache memory is a small random access memory (RAM) that reduces the average cost of accessing data from/to the main memory. The Level one (L1) cache is the nearest to the CPU and is the fastest, but has small capacity. By contrast, the L3[10] cache is the nearest to the main memory and is the slowest, despite its large capacity.

When $N=2^{15}$, the size of the array is 256 bytes*2^{15}=0.25KiB*32K=**8MiB**. Thus, the whole array can be loaded onto the L2 and L3 caches, even though other processes share cache memories. Therefore, quicksort's performances does not slow down when $N<=2^{15}$. Similarly, mergesort, which duplicates the input array, does not have slower performance when $N<=2^{14}$. In fact, when N is small enough to load the whole array onto the L2 cache, asymm_qsort() and merge_array() become faster since the L2 cache is faster than the L3 cache. However, when the array is large enough to overflow the cache memories, the CPU often stalls by waiting for the

7 Their depth of calls are about $\log_2 N$, and the comparisons in each depth are about N. Thus the total comparisons are about $n\log_2 n$.

8 Cache memory is integrated directly into the CPU chip except early age, so the cache memory is also known as the CPU cache.

9 KiB (Kibibyte)=2^{10} bytes, MiB (Mebibyte)=2^{20} bytes, and so on.

10 Some recent CPUs have an L4 cache.

completion of the main memory access, making the time complexity of the sorting algorithms greater than $O(n\log(n))$.

Data in this section are available at https://gitlab.com/qmisort/acquisition/blob/master/sort/indirect/imsort

3. Indirect mergesort

This section estimates the indirect sorting algorithm applied to mergesort.

Mergesort has the following two disadvantages compared with quicksort: the duplication of input array wastes the heap, and the number of copies in mergesort is about two times that in quicksort. Nevertheless, the indirect sorting algorithm reduces these issues by creating and sorting an index,[11] and then reordering the input array according to the sorted index.

The following example[12] demonstrates the indirect mergesort.

```
( b,  c,  a)      Input array enclosed in parentheses
[&b, &c, &a]      Create an index enclosed in square brackets.
[&a, &b, &c]      Sort the index by mergesort.
                  &a points to the third element, &b points to the first element,
                  and &c points to the second element.
( -,  c,  a) {b}  Pick up and save the first element b to a temporary buffer {}.
                  The previous place of b is regarded as an empty hole, denoted by -.
( a,  c,  -) {b}  The hole is at the first position, and the first pointer &a points
                  to the third element. Thus, move the third element a to the hole,
                  and change the pointer &a to point to the first element.
                  As a result, the element a has moved to the correct position,
                  and the empty hole has moved to the third position.
( a,  -,  c) {b}  The hole is at the third position, and the third pointer &c points
                  to the second element. Thus, move the second element c to the hole,
                  and change the pointer &c to point to the third element.
                  As a result, the element c has moved to the correct position,
                  and the empty hole has moved to the second position.
( a,  b,  c) {-}  The second pointer &b points to the first element, which corresponds
                  to the picked up position of b. Thus, move the saved element b to the
                  hole and change the pointer &b to point to the second element.
                  As a result, the hole is filled with element b,
                  and the temporary buffer is cleared. A permutation has completed.
( a,  b,  c)      Pick up the second pointer &b. Because it points to the second
                  element, do nothing.
( a,  b,  c)      Pick up the third pointer &c. Because it points to the third element,
                  do nothing. The array is sorted.
[ ]               Discard the index.
```

Since elements move by cyclic permutation in the reordering, {} ←b ←a ←c ←{b}, the number of copies reduces to about N in the case of a random data sequence as demonstrated by the following.

```
$ N=10000; random.awk $N | Debug/Sort -N $N -mqI m -B $N -V 1
arguments: -N 10000 -mqI m -B 10000 -V 1
qsort(3)      usec = 4337 call = 0    compare = 120510 copy = 0
asymm_qsort() usec = 6172 call = 6763 compare = 135848 copy = 74178
merge_array() usec = 3555 call = 9999 compare = 120510 copy = 130509
imsort(&)     usec = 3767 call = 9999 compare = 120514 copy = 10010
```

imsort(&): Indirect mergesort called by the -I m option where (&) refers to the indirect sort.

11 Indirect sorting is also called Index sorting.

12 A sample of source code is provided at glibc: https://code.woboq.org/userspace/glibc/stdlib/msort.c.html lines 241 ~ 280.

-B $N: Command option to change the algorithm from asymmetric quicksort to an indirect sort called by the -I option. Thus, the -B $N option skips the asymmetric quicksort and starts an indirect sort from the beginning.

Since the size of a pointer is 8 bytes in C language on a 64 bit CPU, the total size of the duplicated index in indirect mergesort is 16N bytes. Thus, if an element size is larger than 16 bytes, the required workspace is smaller than the duplicated input array, and the cost of moving pointers is smaller than that of moving elements. Therefore, the indirect sorting algorithm is usually better for mergesort.

Fig. 3 adds a series of indirect mergesort to Fig. 2.

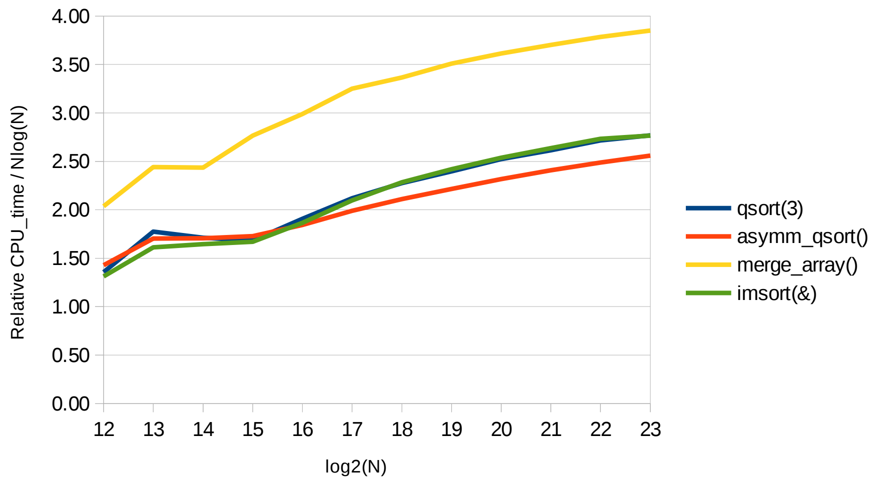

Fig. 3: Relative CPU times for algorithms, including indirect mergesort (256 bytes/element).

The performance of imsort(&) is better than merge_array(), but is similar to the quicksort algorithms. However, the effect of the indirect sorting algorithm depends on element size. Thus, Fig. 4 illustrates the same series with larger elements (1024 bytes/element). Here, imsort(&) becomes the fastest algorithm[13].

13 qsort(3) is also the fastest because it works as indirect mergesort.

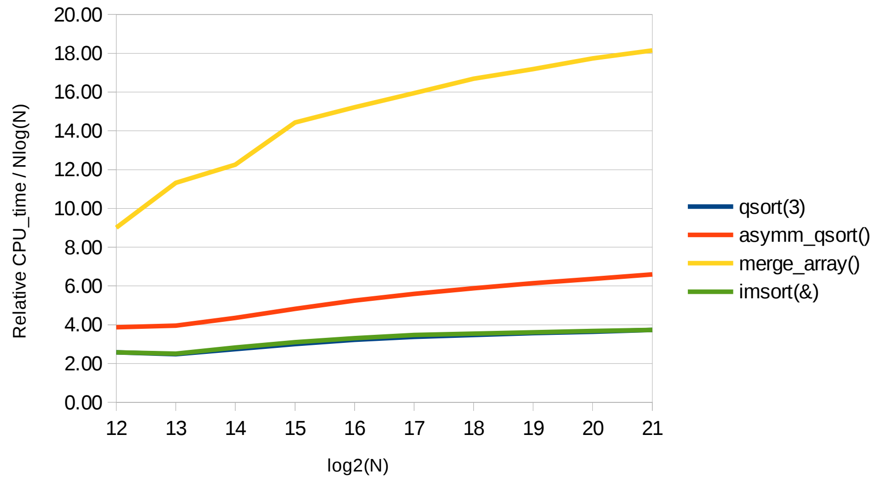

Fig. 4: Relative CPU times of algorithms, including indirect mergesort (1024 bytes/element).

By contrast, for small element sizes, the advantages of indirect mergesort are lost, as demonstrated by Fig. 5, which shows the relative CPU time in the case of 16 bytes/element.

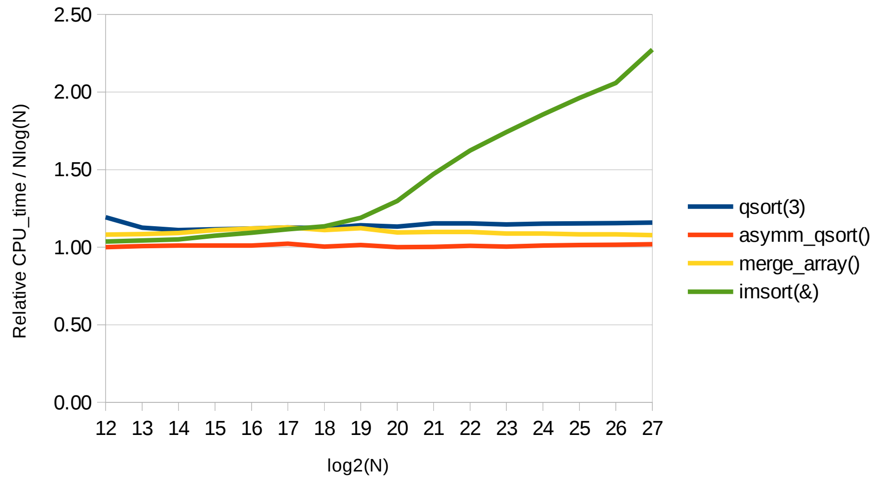

Fig. 5: Relative CPU times of algorithms, including indirect mergesort (16 bytes/element).

Here, imsort(&) is the slowest when N>=2^{18} because N=2^{18} is the maximum number where the L3 cache can load the input array and two indexes: (16+2*8)N=32*2^{18}=2^{23}=8MiB, and greater N causes cache overflow. Thus, the CPU sometimes stalls to write a part of an old merged index from the cache onto the main memory. Further, an element is referred for comparison with another element while merging, so a part of the array element including the sorting key is repeatedly read from the main memory onto the cache memories. Therefore, sorting keys and two indexes scramble for cache memories, making the time complexity greater than $O(n\log(n))$.

By contrast, asymm_qsort() do not slow down with increased N because, after several recursive divisions, sub-arrays shrink their size to less than the L3 cache. For example, when N=2^{27} and after **8** recursive divisions, the size of the shorter sub-arrays become smaller than 8MiB: $16*2^{27}/2^{\mathbf{8}}=2^{4+27-\mathbf{8}}=2^{23}$. Therefore, quicksort has an **advantage** over indirect mergesort where the size of an element is small.

The following two charts, Figs. 6 and 7, show the relative CPU time of indirect mergesort and asymmetric quicksort for various element sizes. The Y-axis is the normalized `CPU_time/nlog(n)` as in Fig. 1.

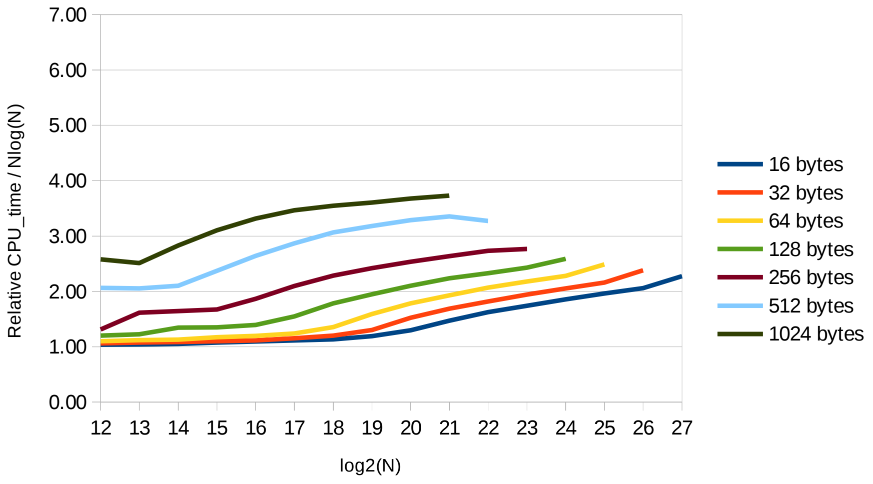

Fig. 6: Relative CPU times for indirect mergesort with various element sizes.

Each series rises as N increases. Thus, the time complexity of indirect mergesort is larger than $O(n\log(n))$.

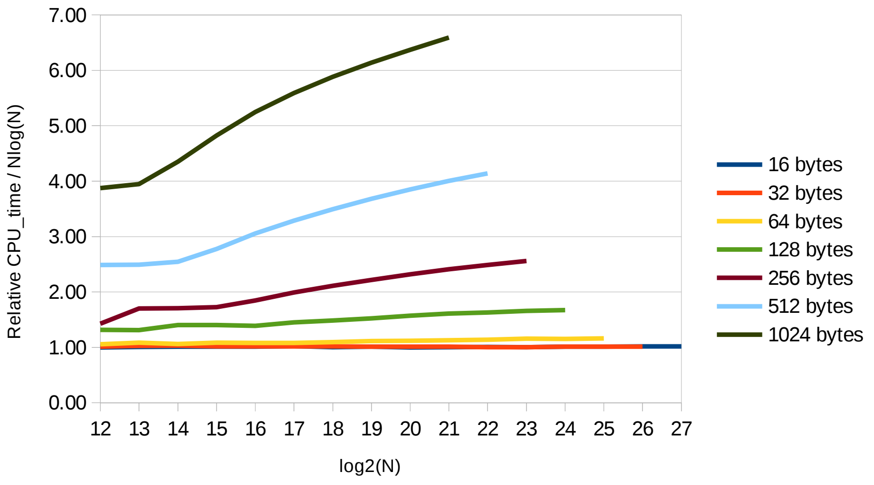

Fig. 7: Relative CPU times for asymmetric quicksort with various element sizes.

The time complexity of asymmetric quicksort is greater than $O(n\log(n))$ except when the element size is small.

When the size is less than 256 bytes/element, asymmetric quicksort is faster than indirect mergesort; otherwise, indirect mergesort is faster. Note that 256 bytes/element is just a boundary size, not a magic number. The following chart shows the same case as in Fig. 3 performed on the secondary PC, which has the following hardware 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
```

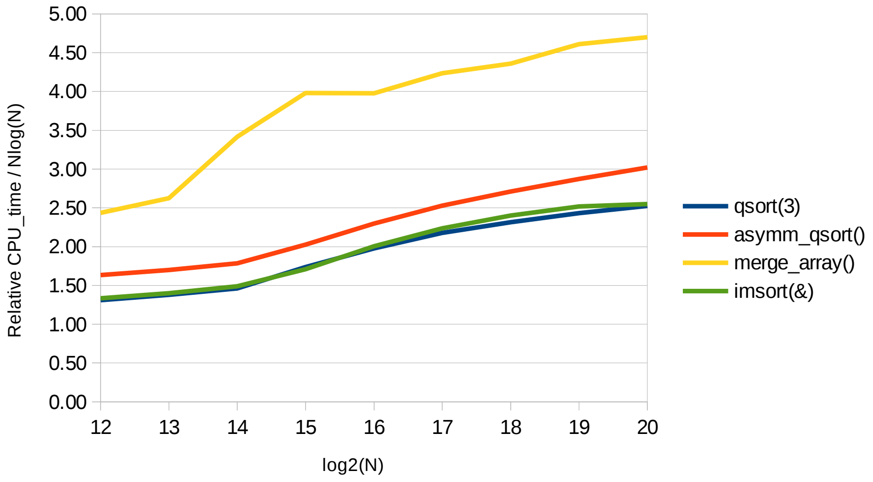

Fig. 8: Relative CPU times on a secondary with 256 bytes/element.

Asymmetric quicksort is slower than indirect mergesort because the total size of the cache memories is smaller than that on the main PC.

In conclusion, indirect mergesort is faster than asymmetric quicksort if the size of an element is huge; otherwise, it is the reverse.

Data in this section are available at https://gitlab.com/qmisort/acquisition/blob/master/sort/indirect/imsort

4. Indirect asymmetric quicksort

This section investigates indirect asymmetric quicksort for optimization.

The indirect sorting algorithm used to sort the index is applicable to any sorting algorithm, including quicksort. Asymmetric quicksort chooses a pivot from several types of median-of to get a good pivot. While N is huge, asymmetric quicksort chooses a median-of random $((\log_2 N)/2)|1$[14] elements, and then as N is decreased by partitioning, asymmetric quicksort changes the choice of pivot to random median-of-five, median-of-three, and, finally, the middle element, thus reducing the calculation cost.

To properly use the indirect asymmetric quicksort, it is necessary to determine thresholds to change the median-of. Fig. 9 shows the relative CPU time of three types of median-of with various thresholds to switch them to the choice of the middle element. The X-axis is the threshold number in $\log_2 N$. The Y-axis is the normalized `CPU_time/nlog(n)`, where the average of imsort(&) is 1.0. The element size is 128 bytes/element and $N=2^{20}$.

14 The operation "/2" reduce the number of elements to pick up, and "|1 " sets the lowest bit to make an odd number.

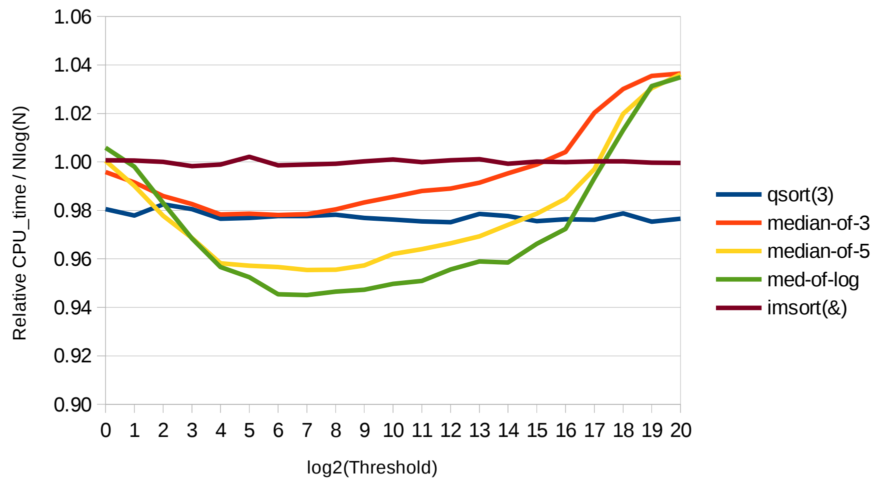

Fig. 9: Relative CPU times for different thresholds to change the pivoting for various median-of types.

Here, median-of-3, median-of-5, and med-of-log are measured commonly using the indirect asymmetric quicksort iqsort(&), called by the **-I q** option, and are renamed to distinguish one from the other.

Data are available at https://gitlab.com/qmisort/acquisition/blob/master/sort/indirect/pivot

The minimal points on the chart are the best thresholds[15]. Thus, we chose a common threshold of **2^6**(=64).

The mixed-median-of could reduce the total calculation cost for asymmetric quicksort. Thus, we suggest the following ranges of N to apply each median-of.

```
type          range of N   ((log₂N)/2)|1
-----------   ------------ -------------
median-of-L   4096 -       7,9,11,...
median-of-5    256 – 4095  5
median-of-3     64 – 255   3
middle             - 63    1
```

Fig. 10 shows the relative CPU time of two indirect asymmetric quicksort algorithms that use mixed median-of and median-of-logarithmic in the case of 256 bytes/element. The Y-axis is the same as in Fig. 3.

15 The rightmost of the series chooses the middle element from the beginning; thus, they converge. The leftmost of series chooses each median-of entirely; thus, the median-of-logarithmic is slowest because of the computational complexity.

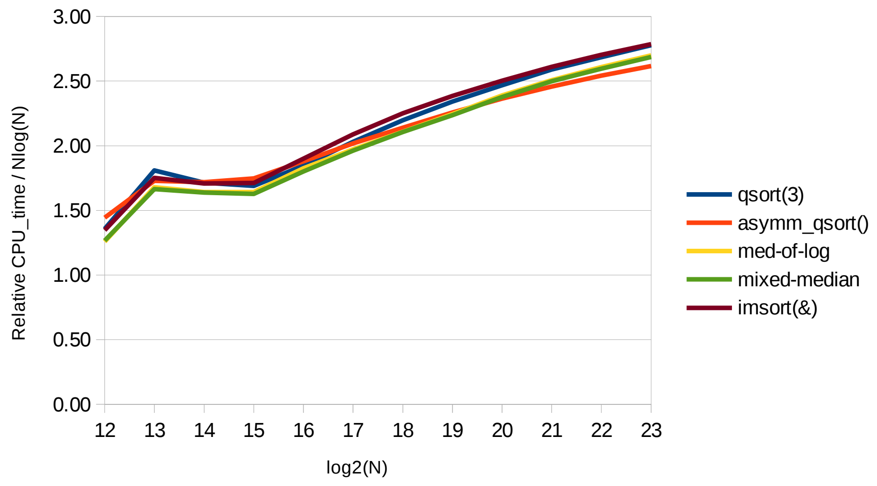

Fig. 10: Comparison of mixed-median-of and median-of-logarithmic.

Here, mixed-median is also measured by iqsort(&), and is renamed to distinguish it from med-of-log as shown in Fig. 9.

Data are available at https://gitlab.com/qmisort/acquisition/blob/master/sort/indirect/mixed

Mixed median-of is slightly faster than median-of-logarithmic, but the difference is less than 0.5%. Thus, the median-of-logarithmic is considered adequately effective, and median-of-5 and median-of-3 are unnecessary.

5. Hybrid asymmetric quicksort with indirect sorting algorithms

This section examines hybrid asymmetric quicksorts with indirect quicksort or indirect mergesort.

If the total sizes of indexes and sub-arrays are small enough to load onto the cache memories, the disadvantage of indirect sorting algorithms disappears. Also, quicksort shrinks the size of sub-arrays after several recursive divisions. Thus, it is worth investigating the combination of these approaches. Fig. 11 shows the relative CPU time of the hybrid algorithms for various thresholds of N to switch from simplified asymmetric quicksort[16] to indirect quicksort or mergesort. The size of each element is 256 bytes. The X-axis is the threshold in $\log_2 N$. The Y-axis is the normalized `CPU_time/nlog(n)` where the average of asymm_qsort() is 1.0.

16 The original asymmetric quicksort chooses a pivot using median-of-logarithmic, median-of-5, median-of-3, and the middle element. However, median-of-5 and the followings are applied when the sub-arrays are shortened. Therefore, the simplified asymmetric quicksort uses only median-of-logarithmic.

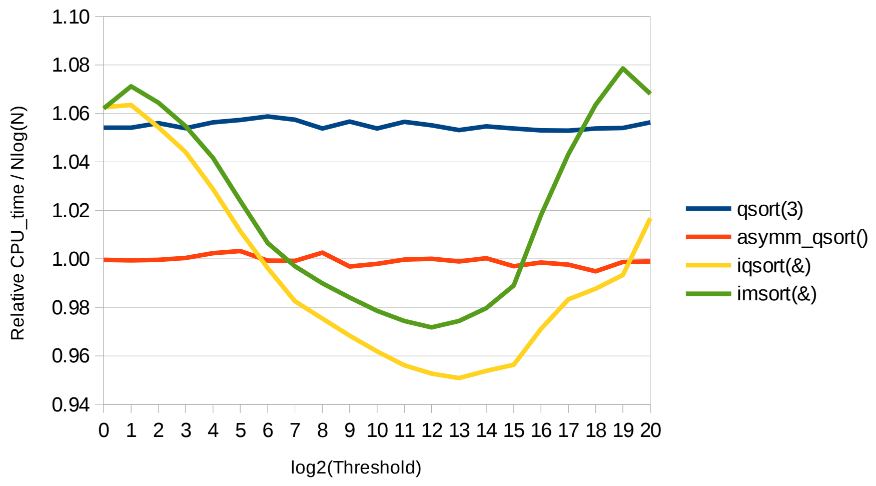

Fig. 11: Relative CPU times for hybrid sorting algorithms (256 bytes/element).

Data are available at https://gitlab.com/qmisort/acquisition/blob/master/sort/indirect/hybrid

A hybrid sorting algorithm with indirect quicksort is the fastest[17] at N=2^{13}, where the size of the index is 8*2^{13}=2^{16}=64KiB and the size of the sub-array is 256*2^{13}=2^{21}=2MiB, allowing them to fit on the L1 and L2 cache memories. This element size is advantageous for quicksort and indirect sorting algorithms. If the element size is huge or tiny, this approach loses that advantage. Therefore, we should estimate the performances for various element sizes.

Fig. 12[18] shows the relative CPU time of a hybrid sorting algorithm with indirect quicksort for various element sizes to find the best threshold number to switch. The X-axis is the threshold in $\log_2 N$. The Y-axis is the normalized `CPU_time/nlog(n)`, where the average of asymm_qsort() is 1.0 for each element size.

17 The rightmost points of iqsort(&) and imsort(&) are the initial individual indirect sorting functions. Their leftmost points are the simplified asymmetric quicksort that chooses the median-of-logarithmic.

18 The series in this chart is not an average of random sequences.

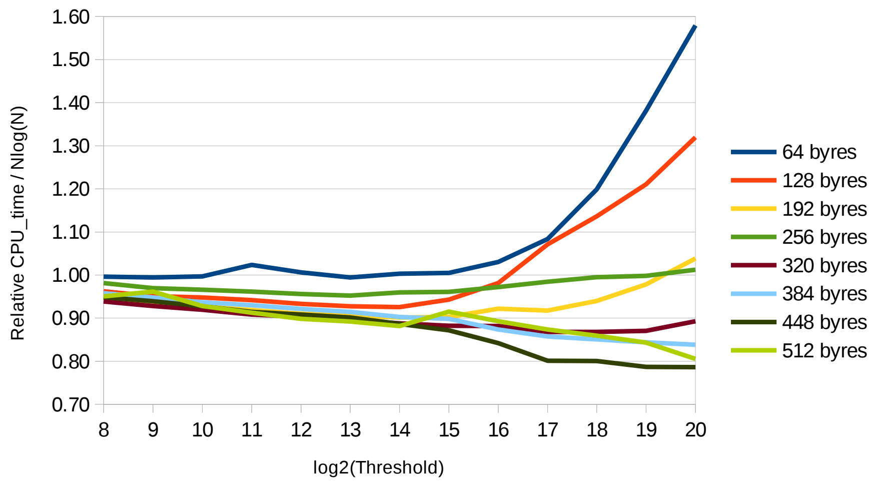

Fig. 12: Relative CPU times for hybrid quicksort for various element sizes.

When the size is small, the leftmost point is the fastest, so indirect sorting is unnecessary. By contrast, when the size is large, the rightmost point is the fastest, so asymmetric quicksort is unnecessary. Therefore, hybrid sorting is effective when the size is mid-range. Fig. 13 shows the best performance of hybrid quicksort at the minimal points for various element sizes. The X-axis shows the size of the element. The Y-axis is the normalized *`CPU_time/nlog(n)`*, where the average of each asymm_qsort() is 1.0.

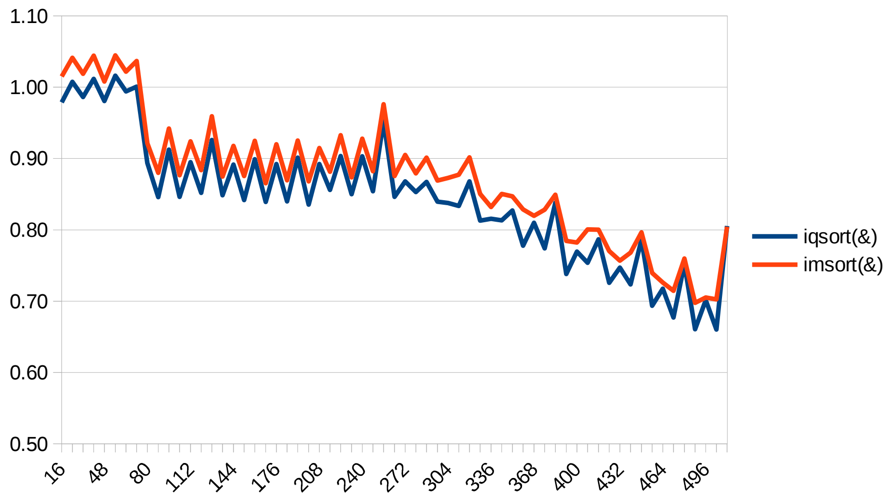

Fig. 13: Minimal points for various element sizes.

Here, iqsort(&) combines with indirect asymmetric quicksort, and imsort(&) combines with indirect mergesort.

As seen in Fig. 13, these series fall into three zones: the fastest function with asymmetric quicksort in the left zone with up to 80 bytes/element, indirect sorting in the right zone with more than around 300 bytes/element, and hybrid sorting in the middle zone. Nevertheless, the boundary sizes of zones depend on the size of the cache memories. To demonstrate this, we perform the same calculations on the secondary PC, as shown in Fig. 14.

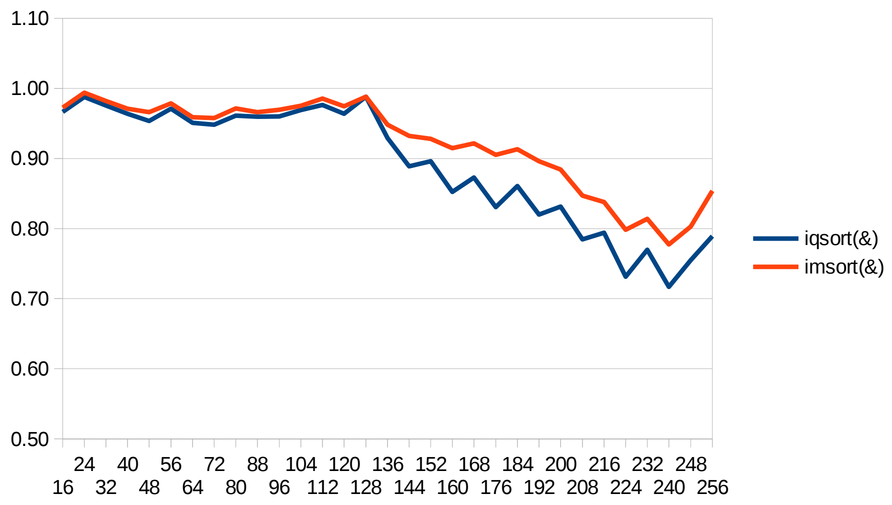

Fig. 14: Minimal points for various element sizes on the secondary PC.

Data up to here in this section are available at https://gitlab.com/qmisort/acquisition/blob/master/sort/indirect/hybrid

In Fig. 14, there is no left zone, and the middle zone (hybrid sorting) is up to 136 bytes. Therefore, the choice of the best algorithm depends on the size of the cache memories and element size. Figs. 13 and 14 show that indirect asymmetric quicksort is faster than indirect mergesort, regardless of element size.

Fig. 15 shows the relative CPU time of hybrid quicksort for various element sizes with the following thresholds to switch from asymmetric quicksort to indirect asymmetric quicksort on the main PC. The Y-axis is the same as in Fig. 7.

```
Size [bytes]   Threshold
------------   ----------
      – 64     2^12 (4096)
  128 – 256    2^14 (16384)
  512 -        (full indirect quicksort)
```

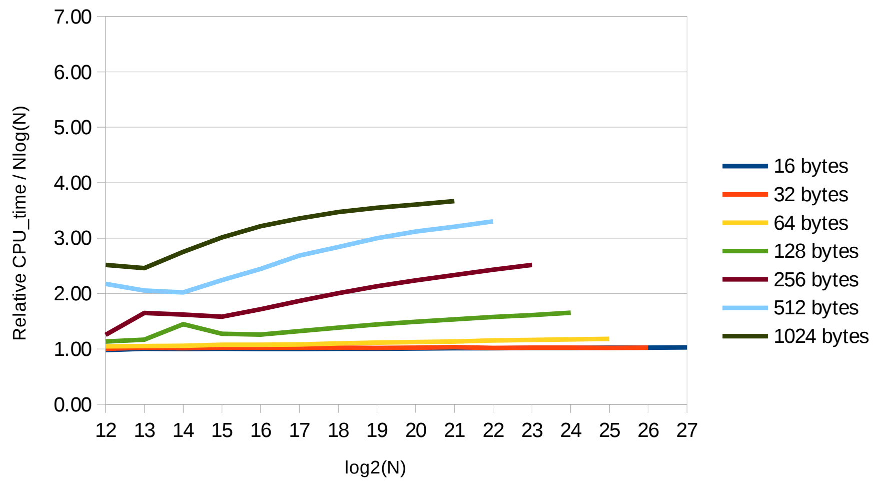

Fig. 15: Hybrid asymmetric quicksort for various element sizes.

Data are available at https://gitlab.com/qmisort/Sort/raw/master/acquisition/indirect/ticket

When the element size is small, the series is flat, as expected. When the element size of is large, the series are smaller than in Fig. 7, but the time complexity is still greater than $O(n\log(n))$.

6. Ticket sort

This section suggests how to reduce the time complexity to $O(n\log(n))$.

As described above, sorting keys and indexes scramble for cache memories in indirect sorting while N is large, making the time complexity greater than $O(n\log(n))$. Nevertheless, if the element size is small, the time complexity of asymmetric quicksort is $O(n\log(n))$. Therefore, we suggest extending the index by including a sorting key such as the following.

```
Typedef struct {
   char    key[16];       // Sorting key
   void    *body;         // points to an array element
} TICKET;
```

The size and data type of “key” are arbitrary, so users have to define them. The function for comparison should be defined to adjust the new data structure except when the key is located at the first position in the array element.

The key is copied from each array element to prevent it from being accessed during the index sorting. If the index is sorted by asymmetric quicksort, the performance does not slow down if the extended index is small. Fig. 16 shows the relative CPU time in the case of 256 bytes/element. The Y-axis is the same as in Fig. 3.

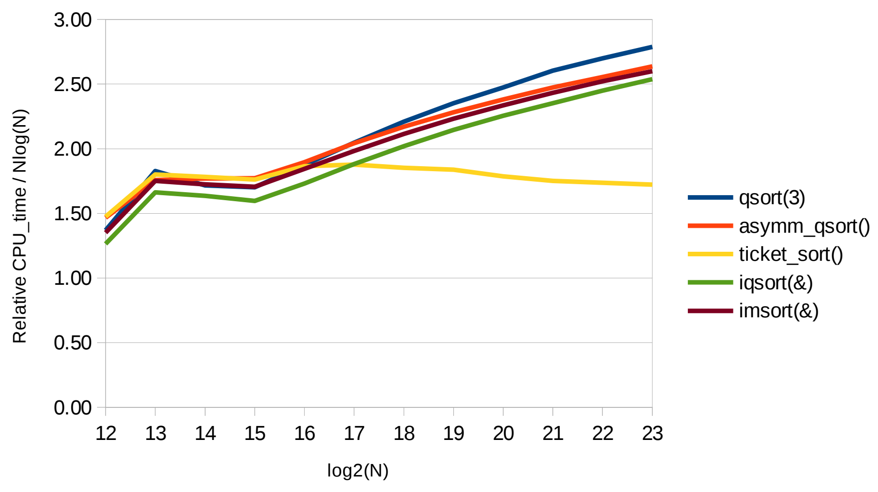

Fig. 16: Relative CPU times for ticket sort (256 bytes/element).

Here, ticket_sort(), called the **-k** option, uses the extended index.

From Fig. 16, the performance of ticket_sort() does not slow down, and is fastest when N is large.

Fig. 17 shows the relative CPU time of ticket_sort() for various element sizes. The Y-axis is the same in the previous chart.

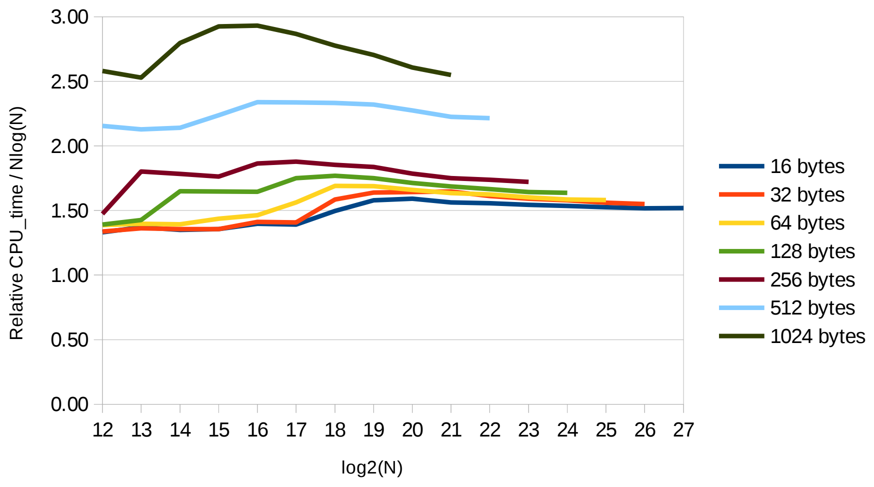

Fig. 17: Relative CPU times for ticket_sort() with various element sizes.

The series of ticket_sort() converges almost flat when N is large, so the time complexity is $O(n\log(n))$. We suggest calling this algorithm "ticket sort" because the indirect sort is conventionally called a "tag sort"[19].

Data in this section are available at https://gitlab.com/qmisort/acquisition/blob/master/sort/indirect/ticket

19 One Java program of tag sort is provided at https://www.geeksforgeeks.org/tag-sort.

7. Stability

Since the array elements do not move while the index is sorted, we can judge which element is initially at the lower position when the two keys are equal. Thus, we can make indirect sorting algorithms stable. For example, see the following code to stabilize the tag sort and the ticket sort.

```
// cf. Appendix A. tag sort
static int my_comp(const void *p1, const void *p2) {
   int rtn = compare(*(const void **)p1,  *(const void **)p2);
   if (! rtn)     // two elements are equal.
       rtn = *(const void **)p1 > *(const void **)p2? 1: -1;
   return  rtn;
}

// cf. Appendix B. ticket sort
static int my_comp(const void *p1, const void *p2) {
   int rtn = compare(p1, p2);    // You have to modify here.
   if (! rtn)     // two elements are equal.
       rtn = ((TICKET *)p1)->body > ((TICKET *)p2)->body? 1: -1;
   return  rtn;
}
```

* my_comp() in the ticket_sort supposes that the compare() compares two elements like strcmp(3).

The following two command lines demonstrate the stability of ticket_sort().

```
$ random.awk | adn.awk | tee data | cat -v
1^@1
6^@2
3^@3
0^@4
1^@5
4^@6
3^@7
```

tee(1): Linux command to read from standard input and write to standard output and files.
cat(1): Linux command to concatenate files and print on the standard output.

First, random.awk generates seven random numbers. Then, adn.awk adds line numbers following a null character '\0'[20], tee command writes to a file “data” from standard input passing them to standard output, and the -v option of cat command shows the non-printing characters following a caret. Here, “^@” refers to a null character.

```
$ Debug/Sort -kp data | cat -v
qsort(3)        usec = 20
ticket_sort()   usec = 32
0^@4
1^@1
1^@5
3^@3
3^@7
4^@6
6^@2
```

The -k option of Debug/Sort calls a ticket_sort(), and the -p option prints out the result. The line numbers of array elements “1” and “3” are ordered ascendingly. Therefore, ticket_sort() is stable[21].

20 A null character terminates a string in C language.

21 If the ticket_sort() fails to allocate an index in the heap, the stability is lost.

8. Conclusion

Asymmetric Quicksort is faster than top-down mergesort for random input data. The time complexity of the simple indirect sorting is greater than $O(n\log(n))$. By contrast, the time complexity of quicksort is $O(n\log(n))$ if the element size is small. Ticket sort extends the index of indirect quicksort to guarantee the time complexity to $O(n\log(n))$, regardless of element size. In addition, it is easy to make the ticket sort and the tag sort stable.

9. Links

Spreadsheets of LibreOffice Calc: Main PC, Secondary PC
Repository: https://gitlab.com/qmisort/Sort
How to evaluate and build: https://sites.google.com/site/qmisort/qmisort

10. References

1. Quicksort: http://algs4.cs.princeton.edu/23quicksort
2. Mergesort: https://algs4.cs.princeton.edu/22mergesort
3. Takeuchi Leorge. “Asymmetric Quicksort”. viXra.org. 2016-08

Appendix A. Tag sort

The following program source code shows an example of a stable tag sort, which is an implementation of the simplest indirect sorting algorithm.

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

extern void asymm_qsort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *));

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

static int (*comp)(const void *, const void *);
static int my_comp(const void *p1, const void *p2) {
   int rtn = comp(*(const void **)p1,  *(const void **)p2);
   if (! rtn)        // two elements are equal.
       rtn = *(const void **)p1 > *(const void **)p2? 1: -1;
   return   rtn;
}

void tag_sort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *)) {
    if (nmemb <= 1) return;
    void    **tags = calloc(sizeof(void *), nmemb); // Allocate an index.
    if ( ! tags)   // failed to allocate memory
        asymm_qsort(base, nmemb, size, compare);
    else {
        comp = compare;
        char *src = base;
        void **tag = tags;
        for (size_t i = 0; i < nmemb; i++) {   // Make an index.
            *tag++ = src; src += size;
        }
        asymm_qsort(tags, nmemb, sizeof(void *), my_comp); // Sort the index
        // reorder array elements
        char save[size];
        void **t, *dst, *idx;
        tag = tags; dst = base;
        for (size_t i = 0; i < nmemb; i++) {
            if (*(t = tag) != dst) {    // an element is not placed at the correct position
                idx = dst; copy(save, idx); // save an element
                do {
                    copy(idx, src = *t);        // move an element
                    *t = idx;                   // reset the address
                    t = &tags[((idx = src) - base) / size];   // points the new hole
                } while (*t != dst);            // cyclic permutation
                copy(*t = src, save);   // restore saved element
            }
            dst += size; tag++;
        }
        free(tags);
    }
}
```

Appendix B. Ticket sort

The following program shows the source code of a stable indirect sorting algorithm that uses an extended index including sorting keys to reduce the time complexity to $O(n\log(n))$.

```
#include "stdlib.h"
#include "string.h"

extern void asymm_qsort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *));

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

typedef struct {
    void    *key1, *key2;     // sorting key data
    void    *body;            // refers to an array element
} TICKET;

static int (*comp)(const void *, const void *);
static int my_comp(const void *p1, const void *p2) {
    int rtn = comp(p1, p2);
    if (! rtn)          // two elements are equal.
        rtn = ((TICKET *)p1)->body > ((TICKET *)p2)->body? 1: -1;
    return  rtn;
}

void ticket_sort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *)) {
    if (nmemb <= 1) return;
    TICKET *tickets = calloc(sizeof(TICKET), nmemb);
    TICKET  *tic = tickets;
    if ( !tic)   // failed to allocate memory
        asymm_qsort(base, nmemb, size, compare);
    else {
        comp = compare;
        char    save[size], *body = base;
        for (size_t i = 0; i < nmemb; i++) {    // Build up an index.
            tic->body = body;                   // Point an array element.
            tic->key1 = ((TICKET *)body)->key1; // Copy the first 8 bytes.
            tic->key2 = ((TICKET *)body)->key2; // Copy the next 8 bytes.
            tic++; body += size;
        }
        asymm_qsort(tickets, nmemb, sizeof(TICKET), my_comp); // Sort the index
        // reorder array elements
        TICKET  *t;
        void    *src = base, *dst;
        tic = tickets; body = base;
        for (size_t i = 0; i < nmemb; i++) {
            if ((t = tic)->body != body) {  // an element is not placed at the correct position
                copy(save, dst = body);         // save an element
                do {
                    copy(dst, src = t->body);   // move an element
                    t->body = dst;              // reset the address
                    t = &tickets[((dst = src) - base) / size];   // points the new hole
                } while (t->body != body);      // cyclic permutation
                copy((t->body = src), save);    // restore saved element
            }
            body += size; tic++;
        }
        free(tickets);
    }
}
```