Increasing the efficiency of existing sorting algorithms by using randomized wrappers

Alexandros V. Gerbessiotis
CS Department
New Jersey Institute of Technology
Newark, NJ 07102.
alexg@cs.njit.edu
Tel: (973)-596-3244
Fax: (973)-596-5777

Constantinos J. Siniolakis
The American College of Greece
6 Gravias St.
Aghia Paraskevi
Athens, 15342
Greece.

October 8, 2002

Abstract

There are several available generic sorting algorithms that are highly optimized and are provided as part of standard programming libraries such as the `qsort` implementation of quicksort available through the C Standard library. In this work we present a practical algorithmic transformation that uses random sampling and ordered search, that can convert an existing implementation of any sorting algorithm X into a sorting algorithm Y of higher observed efficiency that still utilizes X for sorting. The resulting algorithm Y is a randomized algorithm of faster running time with high probability. Experimental results based on an implementation of the transformation support the efficiency claim. The transformation could be used as a wrapper for sorting implementations to increase their efficiency and eliminate spikes of poor performance of such implementations for certain input instances.

Keywords: Algorithm design - Experimental algorithmics - Randomized algorithms - Sorting algorithm implementations

1 Introduction

Currently available software libraries provide as standard generic sorting algorithm implementations that are highly optimized. Such an example is the `qsort` implementation available through the C Standard library which is often quicksort based. Other implementations are based on heapsort and mergesort when issues such as worst-case running time, in-place sorting (same space is used for the input and output and other extra space used is constant) and stability (same valued keys retain their relative order in input and output) become important. In this work we present a conceptually simple and yet quite practical transformation, that can convert an existing implementation of any sorting algorithm X into a new sorting algorithm Y of higher efficiency that still utilizes X for sorting, and also uses random sampling and searching in an ordered table (e.g. binary search). The resulting algorithm Y is a randomized algorithm and attains certain desirable performance properties such as faster running time with high probability. The term "high probability" is to mean that a claim on running time fails to realize with probability at most $1/n^c$, for some constant $c > 0$, where n is the problem size. Experimental results based on an implementation of the transformation support the efficiency claims attributed to the transformed algorithm Y. The transformation could be used as a wrapper for sorting implementations to increase their efficiency and also eliminate spikes of poor performance for certain "bad" input instances. This way, both X, and, Y, also considered an add-on to X, could be offered as alternatives to a programmer who uses a library's sorting functions.

The proposed transformation assumes that the following two functions are minimally available.

- A sorting function, call it SORT, that sorts a sequence of keys, and
- a searching function call it ORDEREDSEARCH that searches for a key in an ordered sequence of keys.

The searching operation can be efficiently implemented through binary search; our requirement, however, is general enough to allow for other implementations. Both SORT and ORDEREDSEARCH are readily available in software libraries that support sorting operations. The C programming language for example provides access to such functions through the `qsort` and `bsearch` function calls respectively of the C standard library. Given SORT and ORDEREDSEARCH, the proposed transformation derives a randomized sorting algorithm RND_SORT that with high probability runs faster than SORT for a wide range of the parameters that describe the performance of SORT and ORDEREDSEARCH.

We note that most of the well known sequential generic sorting algorithms – that sort any data type, unlike specialized distribution based algorithms of the countsort, radixsort, and bucket-sort variety – such as quicksort, heapsort and mergesort exhibit $n \lg n$ (average case) observed practical performance to sort n keys. This performance measure includes key comparison and key exchange operations. For the case of quicksort in particular this measure is not representative of its worst-case performnce since the worst-case performance of quicksort is $\Theta(n^2)$; for heapsort and mergesort worst-case/average-case performance are $O(n \lg n)$. In order to distinguish the performance characteristics of the three algorithms, we can thus claim that the observed performance of quicksort, mergesort and heapsort are respectively $c_q n \lg n$, $c_m n \lg n$, $c_h n \lg n$, for some positive constants c_q, c_m, and c_h, and in practice, on the average, one can verfy that $c_q \leq c_m \leq c_h$. This relationship among the constants is also supported by the experimental results also described later in this work.

Given the practical rather than worst-case superiority of quicksort, this suggests that any algorithm that would compare favorably to quicksort should perform operations similar to those used in quicksort rather than to those used in mergesort (i.e. merging) or heapsort (i.e. heap adjustment operations). These observations raise the question of whether one could improve in some way the practical performance of a worst-case asymptotically fast sorting algorithm (e.g. mergesort, heapsort) that would then exhibit not only the same good theoretical behavior but also good practical behavior. We would like in other words to design an algorithm that is as fast as quicksort in practice.

In particular we claim that, if SORT is a sorting algorithm and ORDEREDSEARCH is a searching algorithm that searches for a key x in an ordered sequence of keys and if the two functions on an input of n keys have time complexity $Cn \lg^{\alpha} n$ and $c_s \lg^{\beta} n$ respectively, where $\alpha \geq \beta$, then, there exists a randomized sorting algorithm RND_SORT that requires time $(c_s + o(1)) n \lg^{\beta} n$, with high probability. The logarithm of x to the base e and two is denoted by $\log x$ and $\lg x$ respectively.

Note that although this is a general result, the interesting cases are for $\alpha = 1$ (e.g. mergesort, heapsort) and $\beta = 1$ (e.g. binary search). For the sorting algorithms mentioned earlier parameter C is one of c_m, c_h and $\alpha = 1$. The experimental results that are presented in this work also suggest that c_s is less than c_m, c_h thus making RND_SORT faster than ordinary mergesort and heapsort. In addition the experiments indicate that expected running time-wise c_s is even less than c_q thus making RND_SORT faster than quicksort.

As the new algorithmic transformation utilizes both SORT and ORDEREDSEARCH one needs to implement and possibly optimize it only once and use it repeatedly with various (SORT , ORDEREDSEARCH) pairs. No complex redesign or rewriting from scratch or fine-tuning of a new sorting algorithm is thus required. In the following sections we first review some theoretical results on random sampling, then present the pseudocode for the proposed transformation and analyze its properties. We then present some special cases of the transformation including one where SORT is not an asymptotically optimal (in the worst-case) sorting algorithm like mergesort and heapsort but a quadratic-time sorting algorithm (e.g. insertion sort). Finally we present some experimental results that were derived by implementing the proposed transformation in ANSI C for a variety of SORT functions.

2 Random Sampling

We present in this section results related to random sampling that follow those presented in [1, 2]. In the discussion and proofs to follow we use the notation originally used in [2]. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n input keys ordered such that $x_i < x_{i+1}$, for $i \in \{1, 2, \ldots, n-1\}$. We are assuming that the keys are distinct because by appending to each key its index we can make them so. For integers $k \geq 2$ and $s \geq 1$, let $Y = \{y_1, y_2, \ldots, y_{ks-1}\}$ be a randomly chosen subset of $ks - 1$ keys of X, also ordered so that $y_i < y_{i+1}$, $i \in \{1, 2, \ldots, ks-2\}$. The following k subsets $X_0, X_1, ..., X_{k-1}$ are then formed.

$$
\begin{aligned}
X_0 &= \{x : x < y_s\}, \\
X_i &= \{x : y_{is} < x < y_{(i+1)s}\}, \quad i \in \{1, \ldots, k-2\},
\end{aligned}
$$

$$X_{k-1} \quad = \quad \{x : y_{(k-1)s} < x\}.$$

For all $i \in \{0, 1, \ldots, k-1\}$, let $n_i = |X_i|$ and $P_i(j)$ be the probability that $n_i = j$. The following is shown in [2]. For the binomial coefficients, we assume that $\binom{i}{j} = 0$ if $j > i$.

Claim 1 *For all i,*

$$P_i(j) = P(j) = \frac{\binom{j}{s-1}\binom{n-j-1}{(k-1)s-1}}{\binom{n}{ks-1}}.$$

We would like to establish upper bounds on the size of the X_i's by proving that the probability of having such a set of size greater than $(1+\varepsilon)$ times the mean $(n-k+1)/k$ is sufficiently small, in particular that

$$\sum_{j \geq \lceil (1+\varepsilon)(n-k+1)/k \rceil + 1} P(j)$$

is very small, where $0 < \varepsilon < 1$. The following is shown in [2].

Claim 2 *Let $k \geq 2$, $s \geq 1$, $ks < n/2$, $n \geq 1$, $0 < \varepsilon < 1$, $\rho > 0$, and*

$$s \geq \frac{1+\varepsilon}{\varepsilon^2}\left(2\rho \log n + \log\left(2\pi k^2 (ks-1) e^{1/(3(ks-1))}\right)\right).$$

Then, the probability that any one of the X_i, for all $i \in \{0, 1, \ldots, k-1\}$, is of size more than $\lceil (1+\varepsilon)(n-k+1)/k \rceil$ is at most $n^{-\rho}$.

3 The Transformation

This section is devoted to the proof of the following theorem.

Theorem 1 *Let* Sort *be a sorting algorithm and* OrderedSearch *be a searching algorithm with running time $Cn \lg^{\alpha} n$ and $c_s \lg^{\beta} n$ respectively, for any constants $C, c_s > 0$, where $\alpha \geq \beta$. Then, there exists a randomized sorting algorithm* Rnd_Sort *that requires time $(c_s + o(1)) n \lg^{\beta} n$, with probability at least $1 - n^{1-\rho}$, for any constant $\rho > 1$.*

Proof: The transformation referred to as Rnd_Sort is described in pseudocode in Figure 1. In the pseudocode, we assume that the input is in the form of array $X[1, \ldots, n]$. We require several additional arrays: the array $Y[1, \ldots, n]$ holds the sorted output, and the arrays $B[1, \ldots, n]$, $C[1, \ldots, k]$,

$D[0,\ldots,k]$, $S[1,\ldots,ks-1]$, and $\bar{S}[1,\ldots,k-1]$ provide temporary working space. The space requirements can be reduced if we re-utilize unused space; arrays Y and $\bar{S}$ are necessary and sufficient for the algorithm to execute. A random number generator, realized by function RANDOM(r) generates uniformly distributed integers in the range $\{1,2,\ldots,r\}$. Function SORT(X,n) sorts a sequence X of n keys. Function ORDEREDSEARCH(k,A,n) searches for key k in an ordered (i.e. sorted in non-decreasing order) sequence A of n keys.

One interesting aspect of quicksort is the selection of the pivot key also referred to as a splitter. Choices for the splitter can be the first, last, middle, the median of the first, last and middle keys, and in general, the median of $2t+1$ sample keys [3]. The median of three is an early indication of the use of the technique of oversampling in splitter selection, where more than one keys are sampled from the input to generate the splitter. The power of oversampling is fully developed in the context of parallel sorting in [5] where it is combined with the idea of using more than one splitters [1] that was originally intended for external memory sorting. RND_SORT also uses the technique of oversampling and thus picks a sample of size $ks-1$ from the input, sorts the sample, and then picks $k-1$ equidistant splitters in the sorted sample that divide the input into k sets of keys based on the relative position of each input key with respect to the splitters.

We implement this idea and towards this in steps 1-5 we perform a sampling without replacement operation. Out of say a set of t keys one key is chosen uniformly at random and then removed from the set; if another key is to be subsequently sampled, it will be chosen uniformly at random from the remaining $t-1$ keys similarly. This sampling is only required for the analysis to be correct. In the subsequent implementation of RND_SORT we use sampling with replacement (where the chosen key is reinserted into the set that retains its size of t before any subsequent sampling is performed) thus simplifying the implementation; the theoretical differences in performance between the two approaches are, however, negligible.

In steps 1-4 of RND_SORT a sample S of size $ks-1$ is selected at random without replacement among the n input keys of X. The "oversampling" parameter s is chosen so that it satisfies the conditions of Claim 2. We note that the operation $X = X - X[r]$ can be easily implemented by swapping keys $X[r]$ and $X[n-i+1]$. Hence, these steps require time $O(ks)$.

In step 5 the sample S of size $ks-1$ is sorted by employing SORT. By definition of SORT, this stage takes time $Cks\lg^{\alpha}(ks)$. The sorted sample S is used in steps 6-8 to determine in $O(k)$ time the set $\bar{S}$ of $k-1$ splitters that would evenly partition S into k "buckets". In steps 9-10 the bucket destination of every key in X is determined with respect to the $k-1$ splitters by employing ORDEREDSEARCH. By definition of ORDEREDSEARCH, this stage takes time at most $c_s n\lg^{\beta} k$.

In steps 11-22 the keys in X are redistributed according to their bucket destination. The steps 11-22 are equivalent to an integer sorting operation[4], where the n integers to be sorted belong to the set $\{1,2,\ldots,k\}$ of possible bucket indices. It follows from the description of steps 11-22 that they require time $O(n)$. Subsequently, in steps 23-24 the keys in each bucket are sorted by employing SORT. The size of each bucket is available through the entries of array D.

By way of Claim 2 the size of each one of k buckets is bounded above by $\lceil(1+\varepsilon)(n-k+1)/k\rceil$ with probability at least $1-kn^{-\rho}$. Hence, by definition of SORT, this stage takes time $C(1+\varepsilon)(n-k+1)\lg^{\alpha}((1+\varepsilon)(n-k+1)/k)$ with high probability, which is k times more than the time required to sort a single bucket. We conclude that with high probability the overall time $T(n)$ of algorithm RND_SORT is bounded above, for $0<\varepsilon<1$, by

$$\begin{aligned} T(n) &= O(ks) + Cks\lg^{a}(ks) + O(k) + c_s n\lg^{\beta} k + O(n) \\ &+ C(1+\epsilon)(n-k+1)\lg^{\alpha}\frac{(n-k+1)}{k} + O(n) \\ &= c_s n\lg^{\beta} k + C(1+\epsilon)(n-k+1)\lg^{\alpha}\frac{(n-k+1)}{k} + O\left(ks\lg^{\alpha}(ks)\right) + O\left(n\right). \end{aligned} \tag{1}$$

In the expression for $T(n)$ the first term is contributed by lines 1-4, the second by line 5, the third by lines 6-8, the fourth by lines 9-10, the fifth by lines 11-22, and the remaining terms by lines 23-24. We choose values for ε, s arbitrarily small so that the conditions of Claim 2 be satisfied. Parameter k must be chosen so that ks is less than n and in a way that the contributions of low order terms be minimized. By substituting in Equation 1 for $\varepsilon = 1/(\lg n)^{1/4}$, $s=(2\rho+3)\lg^{3/2} n$, $k = n/(\rho\lg^{3/2+\gamma} n)$ and thus $ks = O(n/\lg^{\gamma} n)$, for any constant $\gamma \geq \alpha$, we obtain that with probability at least $1-n^{-\rho+1}$ algorithm RND_SORT requires time

$$T(n) = c_s n\lg^{\beta} n + O(n(\lg\lg n)^{\alpha}),$$

since the second term in Equation 1 for the choice of k becomes $O(n(\lg\lg n)^\alpha)$, and the $O(ks\lg^\alpha(ks)) = O(n\lg^{\alpha-\gamma} n)$ term is $O(n)$ for $\gamma \geq \alpha$. The theorem then follows. ■

An obvious choice for OrderedSearch is a binary search method, and therefore the running time of OrderedSearch of a key in an ordered sequence of n keys, is then given by $c_s \lg n$, for some constant $c_s > 0$, i.e $\beta = 1$. If $\beta = 1$ then the transformation can be used to speed-up any sorting algorithm Sort with (worst-case) time complexity $C\ n\lg^\alpha n$, $\alpha > 1$, and any sorting algorithm Sort with time complexity $C\ n\lg n$ provided that $C > c_s$. For sorting algorithms that have $\alpha = 1$ [3, 4], the experimental results in the following section show that the transformation indeed improves upon the performance of heapsort and mergesort confirming the claim that $c_h > c_s, c_m > c_s$ for these algorithms.

The described transformation also applies to the case where algorithm Sort has worst-case running time complexity $n^{1+\zeta}$, for any $\zeta > 0$. The reduction in running time is substantial and is summarized by the following corollary where we choose $\zeta = 1$ for simplicity.

Corollary 1 *Let* Sort *be a sorting algorithm and* OrderedSearch *be a searching algorithm with running time* Cn^2 *and* $c_s\lg^\beta n$ *respectively, for any constants* $c_s > 0$ *and* $\beta \geq 1$*. Then,* Rnd_Sort *sorts* n *keys in time* $5C(2\rho+3)^{2/3}n^{4/3}\lg n + O\left(n\lg^\beta n\right)$ *with probability at least* $1 - n^{1-\rho}$*, for any constant* $\rho > 1$*.*

Proof: The proof of Corollary 1 is similar to that of Theorem 1 and we highlight these differences only. The splitter parameters for Rnd_Sort are chosen so that $\varepsilon = 1/(\lg n)^{1/4}$, $s = (2\rho+3)\lg^{3/2} n$, $\rho > 1$, and $k = n^{2/3}/s^{2/3}$ so that the contributions of sample sorting and bucket sorting are equalized. Under this choice of the sampling/splitting process, each of the k formed buckets would be of maximum size $\lceil (1+\varepsilon)(n-k+1)/k \rceil \leq 2n/k$. We conclude that with probability $1 - n^{1-\rho}$ the overall time $T(n)$ of algorithm Rnd_Sort is bounded above by the following quantity for the time of steps 1-4, 5, 6-8, 9-10, 11-22, 23-24 respectively.

$$\begin{aligned} T(n) &= O(ks) + C(ks)^2 + O(k) + c_s n\lg^\beta k + O(n) \\ &+ Ck\left((1+\epsilon)\frac{(n-k+1)}{k}\right)^2 + O(n) \end{aligned}$$

$$= \quad c_s n \lg^\beta k + 4Cn^2/k + C(ks)^2 + O(n). \qquad (2)$$

By substituting in Equation 2 above for ε, s, k and ρ we get the following bound.

$$\begin{aligned} T(n) &= c_s n \lg^\beta k + 4Cn^{4/3}s^{2/3} + Cn^{4/3}s^{2/3} + O(n) \\ &= 5Cn^{4/3}s^{2/3} + O\left(n \lg^\beta n\right) \\ &= 5C(2\rho+3)^{2/3} n^{4/3} \lg n + O(n \lg^\beta n). \end{aligned}$$

■

If however we retained ζ, then the exponent of n in the running time instead of $4/3$ would be $1 + (\zeta^2/(1+2\zeta))$. The reason that the time bound of RND_SORT in Corollary 1 is superlinear, though subquadratic, is due to the fact that sample sorting is slow. If for example sample sorting is done efficiently in $c_\alpha \lg^\alpha n$ time then the improvements would be as good as those of Theorem 1. Corollary 2 below summarizes such improvements.

Corollary 2 *Let* SORT, AUXSORT *be two sorting algorithms, and* ORDEREDSEARCH *be a searching algorithm with running time Cn^2, $c_\alpha \lg^\alpha n$, and $c_s \lg^\beta n$ respectively, for any constants $C, c_s > 0$ and $\beta \geq 1$. Then,* RND_SORT *modified by using* AUXSORT *in line 5 for sample sorting, sorts n keys in time $c_s n \lg^\beta k + 4Cn\rho \lg^{3/2+\gamma} n + O(n)$ with probability at least $1 - n^{1-\rho}$, for any constant $\rho > 1$ and $\gamma \geq \alpha$.*

Proof: The proof of Corollary 2 is similar to that of Theorem 1 and Corollary 1 and we highlight these differences only. Line 5 of RND_SORT uses now AUXSORT instead of SORT. The splitter parameters for RND_SORT are chosen so that $\varepsilon = 1/(\lg n)^{1/4}$, $s = (2\rho+3)\lg^{3/2} n$, and $k = n/(\rho \lg^{3/2+\gamma} n)$, for any $\gamma \geq \alpha$. Under this choice of the sampling/splitting process, each of the k formed buckets would be of maximum size $\lceil (1+\varepsilon)(n-k+1)/k \rceil \leq 2n/k$. We conclude that with probability $1 - n^{1-\rho}$ the overall time $T(n)$ of algorithm RND_SORT is bounded above by the following quantity for the time of steps 1-4, 5, 6-8, 9-10, 11-22, 23-24 respectively.

$$T(n) \quad = \quad O(ks) + c_\alpha ks \lg^\alpha (ks) + O(k) + c_s n \lg^\beta k + O(n)$$

$$
\begin{aligned}
&+ \quad Ck\left((1+\epsilon)\frac{(n-k+1)}{k}\right)^2 + O(n) \\
&= \quad c_s n \lg^{\beta} k + c_\alpha ks \lg^{\alpha}(ks) + 4Cn^2/k + O(n). \qquad (3)
\end{aligned}
$$

By substituting in Equation 3 above for ε, s, k and ρ we get the following bound.

$$
\begin{aligned}
T(n) &= c_s n \lg^{\beta} k + c_\alpha ks \lg^{\alpha}(ks) + 4Cn^2/k + O(n) \\
&= c_s n \lg^{\beta} k + c_\alpha O(n \lg^{\alpha-\gamma} n) + 4Cn^2/k + O(n) \\
&= c_s n \lg^{\beta} k + c_\alpha O(n) + 4Cn\rho \lg^{3/2+\gamma} n + O(n) \\
&= c_s n \lg^{\beta} k + 4Cn\rho \lg^{3/2+\gamma} n + O(n).
\end{aligned}
$$

This completes the proof. ■

With regard to Corollary 2 we note that for $\alpha = \beta = \gamma = 1$, the running time of RND_SORT becomes $O(n \lg^{5/2} n)$ with probability $1 - n^{1-\rho}$.

The transformation retains the stability properties (i.e. duplicate handling) of SORT provided that the following modifications to the transformation are realized: (a) the information (sample) collected in step 3 includes the position r of key $X[r]$ in the input sequence, (b) a stable SORT would use both pieces of information collected in step 3 in the sorting of step 5, (c) the information (splitters) collected in step 7 also includes the position of a splitter key in the input sequence, and (d) the ORDEREDSEARCH step resolves key equality using the information related to the position of the splitters in the input sequence.

4 Theory vs Practice

The algorithmic transformation described as RND_SORT has been implemented in ANSI C and its performance studied for a variety of SORT functions that satisfy Theorem 1, Corollary 1 and also under the modification implied by Corollary 2 with ORDEREDSEARCH implemented in all instances as binary search. As already mentioned, the only deviation from the RND_SORT description is the use of sampling with replacement rather than the sampling without replacement step of RND_SORT.

We used an Intel 400Mhz Pentium II Linux Red-Hat 7.1 workstation to run our experiments with problem size varying from $16K = 16384$ up to $1024K = 1024 \times 1024$. The algorithmic transformation and the author-written generic implementations of quicksort, heapsort, mergesort and insertion-sort used for SORT are written in ANSI C. The source code was compiled using `gcc` with option `-O3` and using otherwise the default compiler and library installations. Indicated timing results (wall-clock time in seconds) are the averages of three experiments. Insertion sort timing results for large problem sizes were not measured due to high running time. In general the difference in the running time of the fastest from the slowest of the three experiments is no more than 5%. The input consists of strings 40-byte long. Each byte takes values uniformly distributed between 32 and 96 (the equivalent of generating some printable ASCII characters).

The empirical results were obtained using five sequential benchmark sorting algorithms for SORT: the standard C library version of quicksort called *qsort* and referred to as *qs* hereafter, and the authors generic implementations of quicksort, mergesort, heapsort, and insertion sort [3, 4] referred to hereafter as *os*, *ms*, *hs*, and *is*. The variants of RND_SORT resulting after the application of the transformation are referred to as iqs, ios, ims, ihs, and iis respectively. Table 1 reports results obtained from a pure implementation of RND_SORT as used in Theorem 1 and Corollary 1. Table 2 reports results obtained from the variation of RND_SORT implied by Corollary 2 where sample sorting uses AUXSORT. For AUXSORT we used both ms and os with no differences in timing results. The timing results obtained correspond to some generic choices of splitter size and oversampling according to Theorem 1. For the seven problem sizes of the experiments the chosen splitter size and oversampling factor used by RND_SORT was (49,14), (82,15), (140,16), (241, 17), (419, 18), (733,19), and (1289,20) respectively. Because of this the reported results are not the best possible. One could improve the performance of RND_SORT by fine-tuning splitter size and oversampling factor separately for each one of the algorithms. We thus performed additional experiments to study this effect.

The best timing results for iqs, ims, and ios were observed either for the default values of splitter size and oversampling factor or for neighboring values; in this latter case the gains in running time were no more than 5%. For *ihs* and *iis* significant improvements were possible and the splitter sizes for which the best timing results were obtained are reported in Table 3 and Table 4 respectively for the case of the original RND_SORT and the variant of Corollary 2 that uses AUXSORT. In all cases of those tables the oversampling factor was 15. In general varying the oversampling factor from that value did not improve performance noticeably. In the additional experiments splitter size varied in steps that were multiple of 100.

In both Table 1 and Table 2 the transformed algorithm RND_SORT outperforms the corresponding SORT algorithm. For the pair (iqs,qs) the improvement in performance ranges from 25% for $n = 16K$ to 37% for $n = 1024K$. For (ims,ms), (ios,os), and (ihs,hs) the improvements range from 23%, 7%, 29% respectively to 31%, 22%, 45% respectively for Table 1 and similar conclusions apply to Table 2. For iis the reduction in running time was significant even for the smallest of the problem sizes. For problem size at least $64K$ running time was not measured for is, because it was very high. In no problem size did SORT outperform RND_SORT. From the experimental results summarized in the tables, it is also apparent that qs, the qsort of the Standard C library on the used Linux Platform, is a mergesort-based implementation. Because of this the base reference algorithm in the remainder of this discussion is the (ios, os) pair.

With reference to the results in Table 1, the ratio ms/os (running time of mergesort to that of our os) varies from 1.2 to 1.3, whereas the ratio ims/ios is smaller and varies around $1.06 - 1.08$. For heapsort the corresponding ratio hs/os varies from 2.26 to 2.42 and for ihs/ios from 1.6 to 1.8. Similar results are available through Table 2. The only observable difference between Table 1 and Table 2 are the times for iis that are better in the latter table than the former. This is because of the improvements obtained in Corollary 2 compared to Corollary 1. The ratio ihs/ios is $1.6 - 1.7$ if the results of Table 3 or Table 4 are taken into consideration since the running times for ihs are smaller there than in Table 1 and Table 2. Similarly, if the results of Table 4 are used for iis, the ratio iis/ios is between 1.8 and 2.0 as well. Comparing the results of Table 3 to those of Table 4 the effect of using AUXSORT for sample sorting is evident in the performance of iis which is expected anyway if one compares the result of Corollary 1 to the one of Corollary 2. For ihs the improvements are mainly due to the use of larger splitter size values.

What is more important however is a comparison of the running times of iqs and ims to those of qs (original qsort) and os (our version of quick sort). Both ims and iqs are faster than qs and os by as much as 20% as it can be seen in Table 1 for example. Of course under our algorithmic transformation even quick-sort can benefit. The running time of ios is thus faster than those of iqs and ims but by not much. For ihs the improvements are also considerable. Whereas hs is 2-3 times slower than os, ihs (Table 3 and Table 4) is barely 1.25 times slower than os for large problem sizes. Even the running time of iis (Table 4) converges to that of os for large problem sizes.

5 Conclusion

We have presented an algorithmic transformation that can improve the performance of generic sorting algorithm implementations available in various programming libraries. Although one could prove theoretically the benefits of using this transformation, we have implemented the transformation under various Sort functions and studied the performance of the original and the transformed algorithms. The conclusion drawn as a result of this experimental study is that the transformation consistently improves the performance of even highly optimized sorting algorithm implementations.

The research of the first author was partially supported by NSF MRI grant NSF 9977508. Part of this work originated while both authors were with the University of Oxford, UK, where the work of the first author was supported in part by EPSRC(UK) under grant GR/K16999, and the second author was supported by a Bodossaki Foundation Graduate Scholarship.

References

[1] Frazer W. D. and McKellar A. C (1970) Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496-507.

[2] Gerbessiotis A. V. and Valiant L. G. (1994) Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267.

[3] Hoare C. A. R. (1962) Quicksort. *The Computer Journal*, 5:10-15.

[4] Knuth D. E. (1973) *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, Reading.

[5] H. J. Reif and L. G. Valiant. (1987) A logarithmic time sort for linear size networks. *Journal of the ACM,* 34:60-76.

	iqs	qs	ims	ms	ios	os	ihs	hs	iis	is
16K	0.12	0.16	0.13	0.17	0.12	0.13	0.22	0.31	0.83	38.80
32K	0.27	0.36	0.29	0.39	0.26	0.30	0.47	0.68	2.04	173.38
64K	0.56	0.79	0.60	0.81	0.56	0.65	0.97	1.49	5.05	*
128K	1.18	1.74	1.23	1.77	1.15	1.33	2.00	3.24	12.39	*
256K	2.45	3.76	2.56	3.66	2.39	2.91	3.96	7.03	32.22	*
512K	5.10	7.89	5.32	7.89	4.99	6.16	8.57	15.13	79.53	*
1024K	10.62	16.91	11.04	16.20	10.38	13.37	17.72	32.38	225.05	*

Table 1: Experimental results for original RND_SORT. $*$ indicates test was not run.

	iqs	qs	ims	ms	ios	os	ihs	hs	iis	is
16K	0.12	0.16	0.14	0.18	0.12	0.13	0.22	0.31	0.78	37.67
32K	0.27	0.35	0.28	0.39	0.26	0.30	0.46	0.68	1.85	171.67
64K	0.57	0.79	0.60	0.81	0.56	0.65	0.96	1.49	4.41	*
128K	1.17	1.75	1.24	1.77	1.15	1.32	1.97	3.25	10.17	*
256K	2.45	3.81	2.57	3.66	2.41	2.91	4.08	7.01	23.32	*
512K	5.09	7.89	5.28	7.84	4.99	6.17	8.42	15.14	53.32	*
1024K	10.57	16.96	11.31	16.26	10.39	13.40	17.42	32.34	121.60	*

Table 2: Experimental results for modified RND_SORT. $*$ indicates test was not run.

	ihs		iis	
Size	Time	Splitters	Time	Splitters
16K	0.21	100	0.70	100
32K	0.45	100	1.85	100
64K	0.93	400	4.36	200
128K	1.91	600	11.17	400
256K	3.93	1000	27.54	600
512K	8.09	2000	69.02	800
1024K	16.72	2000	176.71	1400

Table 3: Optimal splitter sizes and corresponding times for ihs and iis in original RND_SORT.

	ihs		iis	
Size	Time	Splitters	Time	Splitters
16K	0.20	200	0.21	400
32K	0.42	200	0.44	800
64K	0.86	600	0.91	1800
128K	1.77	1200	1.92	3500
256K	3.67	2000	4.01	6000
512K	7.50	4000	8.53	11000
1024K	15.59	7500	19.63	24000

Table 4: Optimal splitter sizes and corresponding times for ihs and iis in modified RND_SORT.

```
Rnd_Sort(X, n, Sort, OrderedSearch)

for( i = 1; i ≤ ks − 1; i = i + 1 ) // Sampling
    r = Random(n − i + 1);
    S[i] = X[r];
    X = X − {X[r]};
Sort(S, ks − 1); // Sample Sorting
for( i = 1; i ≤ k − 1; i = i + 1 ) // Splitter Selection
    S̄[i] = S[is];
    X = X ∪ S;
for( i = 1; i ≤ n; i = i + 1 ) // Binary Search of keys into splitters
    B[i] = OrderedSearch(X[i], S̄, k − 1);
for( i = 1; i ≤ k; i = i + 1 ) // Initialize buckets
    C[i] = 0;
for( i = 1; i ≤ n; i = i + 1 ) // Find bucket Cardinality
    C[B[i]] = C[B[i]] + 1;
D[1] = C[1];
for( i = 2; i ≤ k; i = i + 1 ) // Note bucket Cardinality
    C[i] = C[i] + C[i − 1];
    D[i] = C[i];
for( i = n; i >= 1; i = i − 1 ) // Place keys in buckets
    Y[C[B[i]]] = X[i];
    C[B[i]] = C[B[i]] − 1;
D[0] = 0;
for( i = 1; i ≤ k; i = i + 1 ) // Sort buckets
    Sort(Y[D[i − 1] + 1 ... D[i]],  D[i] − D[i − 1]);
```

Figure 1: Algorithm Rnd_Sort.