Simple Symmetric Sustainable Sorting
— the greeNsort® article

Jens Oehlschlägel

2024-02-02

Contents

Abstract

We explored an uncharted part of the solution space for sorting algorithms: the role of symmetry in divide&conquer algorithms. We found/designed novel simple binary Quicksort and Mergesort algorithms operating in contiguous space which achieve improved trade-offs between worst-case CPU-efficiency, best-case adaptivity and RAM-requirements. The *greeNsort®* algorithms need less hardware (RAM) and/or less energy (CPU) compared to the prior art. The new algorithms fit a theoretical framework: *Footprint* KPIs allow to compare algorithms with different RAM-requirements, a new *definition* of sorting API-targets simplifies construction of stable algorithms with mirrored scan directions, and our ordinal machine model encourages robust algorithms that minimize access *distance*. Unlike earlier *Quicksorts*, our *Zacksort*, *Zucksort* and *Ducksort* algorithms optimally marry CPU-efficiency and tie-adaptivity. Unlike earlier *Mergesorts* which required 100% distant buffer, our *Frogsort* and *Geckosort* algorithms achieve similar CPU-efficiency with 50% or less local buffer. Unlike natural Mergesorts such as *Timsort* which are optimized for the best case of full-presorting, our *Octosort* and *Squidsort* algorithms achieve excellent bi-adaptivity to presorted best-cases without sacrificing worst-case efficiency in real sorting tasks. Our *Walksort* and *Jumpsort* have lower Footprint than the impressive low-memory *Grailsort* and *Sqrtsort* of Astrelin. Given the current climate-emergency, this is a call to action for all maintainers of sorting libraries, all software-engineers using custom sorting code, all professors teaching algorithms, all IT professionals designing programming languages, compilers and CPUs: check for better algorithms and consider symmetric code-mirroring.

Introduction

The UK Parliamentary Office of Science and Technology (2022) reported a global energy consumption of 555 TWh for data centers and user devices in 2020 (excluding cryptomining, networks and TVs) and is projected to grow to 720 TWh in 2030. IBM estimated that sorting - a basic component of software - is responsible for 25% of computing (Mehlhorn (1977))[1]. If these old estimates hold, sorting costs 11.5 12TWh Nuclear Power Stations (NPS) in 2020 or 15 NPS in 2030, plus embodied energy resp. greenhouse gas for the production of data centers and user devices. Software developers usually consume sorting from few off-the-shelf sorting libraries, improving these libraries can massively save runtime energy (operational costs) and required hardware (embodied costs). Particularly reducing hardware requirements allows using older machines longer, which leads to more sustainable amortization of embodied costs. This paper reports the results of the *greeNsort®* project (2010 - 2023), which developed simple sorting algorithms that need less runtime energy and/or less required hardware.

---

[1]The first non-analog computer was a pure sorting machine designed by Herman Hollerith and sold by a predecessor of IBM in 1890

Sustainability measures

The classic academic KPI for empirically comparing algorithms is runtime in seconds ($runTime$). However, runTime ignores embodied costs and does not allow fair comparison of different sorting algorithms because they require different amounts of memory relative to data size

$$\%RAM = (dataRAM + bufferRAM)/dataRAM$$

with an expected trade-off: lower memory costs (embodied) correlate with higher runtime costs (operational). A straightforward Footprint KPI combining embodied and operational costs is the integral of required hardware over runtime, i.e.

$$tFootprint = avg(\%RAM) * runTime$$

Because operational energy (*Energy* measured at RAPL Socket) is highly correlated with runtime, similar reasoning leads to the energy footprint calculated as

$$eFootprint = avg(\%RAM) * Energy$$

Note that cloud *Functions as a Service* (FaaS) such as AWS Lambda has the payment metric *Memory over Time*, this is exactly *tFootprint*.

Scope of *greeNsort®*

greeNsort® investigates binary divide&conquer algorithms for general comparison in-memory sorting. This investigation and the design of new algorithms is guided by the following values and principles: *greeNsort®* aims on algorithms that are *sustainable*, *general*, directly *stable*, *robust*, *resilient*, *scalable*, *reliable*, *adaptive*, potentially *parallel*, *simple* and *beautiful*.

Values and principles

Sustainability

The overarching goal of *greeNsort®* is providing algorithms that are sustainable in the sense of minimizing *eFootprint* and that are suitable to replace less sustainable algorithms in many places. Hence the following values guided the development:

Generality

The new algorithms should be applicable in as many contexts as possible. This implies comparison sorting on any data types, including the possibility to sort elements of varying size.

Stability

The new algorithms should directly support stable sorting

Robustness

The new algorithms should perform robustly on different types of hardware, whatever the particular features regarding random access, cache-size, branch prediction etc., hence comparisons are done between algorithms not tuned to specific hardware. This implies that *greeNsort®* does not assume a specific machine model like a random-access model with constant costs for single-element access or a disk-model with constant costs for access to blocks of data. It is simple assumed, that there is a monotonic relation between access distance and access-cost.

9

Resilience

The algorithmic portfolio should include algorithms that perform in extreme situations and on old hardware with little (RAM) resources.

Scalability

The new algorithms should reliably perform $N \log N$ in the worst-case, hence only divide&conquer algorithms are considered.

Reliability

The new algorithms should reliably perform $N \log N$ or better for any patterns of input data. Deterministic algorithms have advantages.

Adaptivity

The new algorithms should be adaptive, but not tuned to best cases. The main task of sorting algorithms is to sort data that is not already sorted, hence algorithms are developed to perform well in worst (or average) cases.

Concurrency

The new algorithms should allow proper and easy parallelization, hence the algorithms are described and demonstrated as top-down divide&conquer, not bottom-up. Experts can still implement without recursion.

Simplicity

The new algorithms should be easy to adopt, i.e. write, to debug and to verify. Hence the new algorithms must be simple. This is almost the most important single value, except for the following:

Beauty

The new algorithms should be beautiful. Computer scientists might be surprised that an aesthetic value is listed here. Note: *Simplicity* is an aesthetic value. Spoiler: *Symmetry* is *the* single most important ingredient of the new sustainable algorithms.

Symmetry

In many disciplines symmetry plays an important role, particularly bi-symmetry. Examples are vertebrates in biology, vehicles in engineering and churches in architecture. By contrast, in writing program code symmetry seems to play no relevant role! The von-Neumann-Machine is rife with asymmetries:

- **Access-asymmetry** the fact that memory access is asymmetric either the left element first then the right one or the right element first and then the left one
- **Buffer-asymmetry** the fact that buffer placement relative to data is asymmetric, data may either be placed left of buffer memory (DB) or right of buffer memory (BD)

as is the topic of sorting

- **Order-asymmetry** the fact that that 'order' is asymmetric and reaches from 'low' to 'high' 'keys'
- **Pivot-asymmetry** the fact that a binary pivot-comparison (one of LT, LE, GT, GE) assigns an element equal to the pivot either to one partition or the other
- **Tie-asymmetry** the fact that stable ties are asymmetric, they may represent their original order either from left to right (LR) or from right to left (RL)

Bi-Symmetry as a design principle is rare, and if, bi-symmetry is mostly used on a loop-level. The secret ingredient of the new *greeNsort®* algorithms is the *symmetric-asymmetry* principle often found in nature, engineering and arts: design low-level asymmetry and turn it into high-level symmetry by *mirroring*. For algorithmic-code this means: do not fight but embrace loop-level asymmetries and create recursion-level symmetry. Instead of the usual divide&conquer self-recursive functions, *greeNsort®* uses mutual-recursive code-mirrored functions. Code-mirroring enhances the algorithmic solution space, which contains many unexplored areas and surprising solutions. This begins with a new definition of 'sorting':

Definition of sorting

According to Knuth Knuth (1998), p. 1 sorting is *"the rearrangement of items into ascending or descending order"* and can be distinguished into *stable* and *unstable* sorting. This definition creates four different goals for sorting "*unstable ascending*", "*unstable descending*", "*stable ascending*", and "*stable descending*". Sorting libraries implement all four or a subset of these four. From Knuth's mathematical perspective the definition of sorting is perfect.

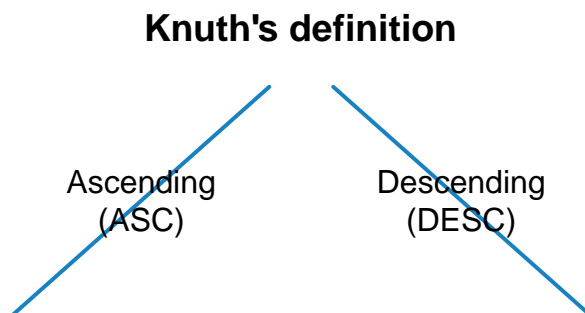**Knuth's definition**

Ascending
(ASC)

Descending
(DESC)

Figure 1: Knuth's definition of sorting

However, in the context of computers, algorithms are not abstract, they operate on *elements* that are stored in *memory* that is addressed from left to to right address *locations* (address locations are notated here from left to right in order to not confuse this with low and high sorting keys). Habitually ascending and descending sequences are written from *left* to *right* :

The two abstract sorting sequences *asc* and *desc* correspond to four concrete sorting sequences in memory: *ascleft*, *ascright*, and *descleft*, and *descleft*. The Difference between *descleft* and *ascleft*, is a reverted - but stable - sequence of ties!

The *greeNsort*® definition is powerful because it facilitates reasoning in an increased solution space; there is similarity to the invention of the *imaginary*

Usually interpreted from left to right

Ascending
(ASC)

Descending
(DESC)
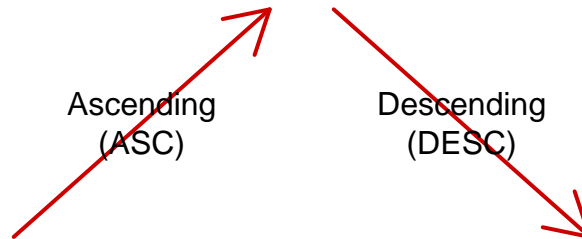
Figure 2: Conventional interpretation: from left to right

greeNsort definition

Ascending
from Left
(AscLeft)

Ascending
unstable
(AscPoor)

Ascending
from Right
(AscRight)

Descending
from Left
(DescLeft)

Descending
unstable
(DescPoor)

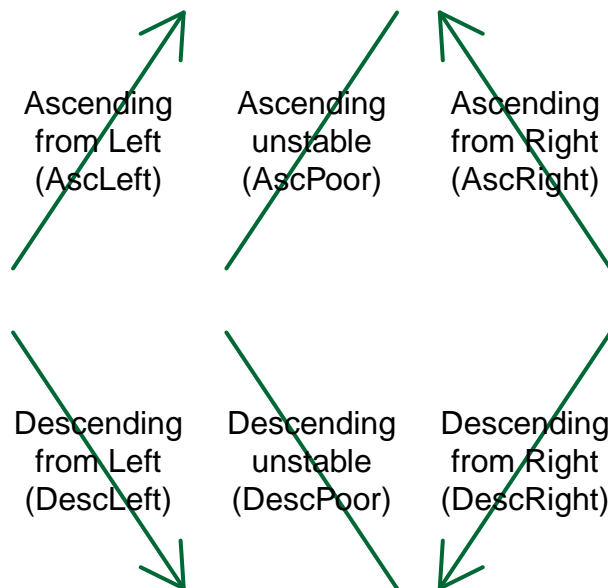Descending
from Right
(DescRight)

Figure 3: greeNsort definition: unstable sorting has two API targets (AscPoor, DescPoor) but stable symmetric sorting has four API targets (AscLeft, AscRight, DescLeft and DescRight)

part which increased the number space from *real numbers* to *complex numbers*, such that suddenly the square-root of a negative number was defined.

Merging or partitioning?

Divide&Conquer sorting falls into two categories: Split&Merge (most work in the merge phases) or Partition&Pool (most work during partitioning phases). Which is better? A prototypical stable deterministic balanced binary Mergesort uses 100% buffer and uses per recursion-level one read-scan with comparisons for the input-data and one write-scan for the merge. A prototypical stable deterministic balanced binary Partitionsort uses 100% buffer, one read-scan and further writing and reading for calculating an (approximate) median as pivot, a read-an-compare-scan for counting the size of the resulting (two) partitions and finally a read-scan and a write-scan for the actual partitioning. Certain sacrifices allow to save some of this work: dropping reliability (determinism) saves the work for calculating the median, further dropping stability and generality allows to saves the buffer and the counting-scan and leads to an attractive family of algorithms: Quicksorts.

Quicksort dilemma

Instead of writing stable partitions with the help of a buffer, the Quicksort family of algorithms is characterized by SWAPing pairs of elements in-place, hence sacrificing[2] stability. Quicksort is tricky, wrong implementations easily lead to quadratic runtime, particularly if deterministic pivots are used, or even leads to endless loops, particularly if sentinels are used. Note that its inventor Tony Hoare defined Quicksort as a probabilistic algorithm using random pivots. Many historic attempts to tune Quicksort with deterministic pivots have finally been regretted because it made Quicksort vulnerable for unexpected runtime and algorithmic complexity attacks. In the following we assume random pivots.

Compared to inefficient approaches like the Lomuto-scheme, the brilliant idea of Hoare in *Quicksort1* was to write the two partitions outside-in from the left and the right, this saves a counting-scan and promised to isolate pivot-ties in a third partition in the middle (1961a; Hoare, 1961b). Hoare's symmetric design obviously aimed on efficient scaling (average $\mathcal{O}(N \log N)$ operations) and early-terminating on ties ($\mathcal{O}(N \log D)$, where D is the number of distinct keys). However, on certain data inputs *Quicksort1* degenerated to $\mathcal{O}(N^2)$.

Several attempts have been made to avoid quadratic runtime for arbitrary data inputs. Sedgewick (1977) compared Hoare's algorithm with two different approaches: asymmetric partitioning and a symmetric partitioning going back to Singleton (1969), where the two pointers from the left and right both stop on pivot-ties before SWAPing. This guarantees a balanced partitioning (and hence average $\mathcal{O}(N \log N)$), but it SWAPs ties and it does not early terminate on ties (hence not $\mathcal{O}(N \log D)$). Sedgewick concluded that the asymmetric approach was vulnerable to quadratic runtime and recommended the symmetric version: *Quicksort2* became the standard for many years.

In an attempt to fix quadratic runtimes in *Quicksort2* implementations with deterministic pivots and in order to gain early-termination on ties, Wegner (1985) and later Bentley and McIlroy (1993) developed *Quicksort3*[3] that collects

---

[2]Note that SWAPing directly equally-sized elements directly sacrifices generality, indirect sorting by SWAPing pointers to variable-sized elements causes random-access and sacrifices robustness.

[3]we report a simpler version that is slightly faster in the worst-case of untied keys and

pivot-ties in a third partition in the middle between the low and high partition. *Quicksort3* achieves $\mathcal{O}(N \log D)$ for tied data, but due to extra-operations for identifying and placing the ties it is slower than *Quicksort2* for untied data. This trade-off between worst-case efficiency and best-case efficiency we term the 'Quicksort dilemma'.

Yaroslavskiy (2009) achieved a notable improvement with dual-pivot Quicksort (*Dupisort*[4]) that uses its extra-operations to create a real third partition. This is faster and no longer a binary sort: it improves the average $\mathcal{O}(N \log_2 N)$ algorithm to $\mathcal{O}(N \log_3 N)$ regarding moves, but the algorithm is strongly serial and difficult to implement branchless. Then, Edelkamp and Weiß (2016) published an even faster and simpler binary Block-Quicksort that reduced branch-mispredictions. However, Block-Quicksort had only a rudimentary and expensive early-termination mechanism, that we skipped in *Quicksort2B*. At this point, history returned in a huge cycle[5] to *Quicksort2*, still leaving the quicksort dilemma unresolved.

Stepping back and analyzing the commonalities of all those attempts, it seems that all authors assume that partitioning must by symmetric, and for early-termination on ties a third partition is necessary. However, the *greeNsort®* analysis of the problem finds that

- for early-termination not three, not two but one partition is sufficient, the algorithm just needs to be able to diagnose an all-tied partition and stop recursing deeper into this branch
- in order to get pure partitions where ties are not in multiple partitions, the partitioning must be Mutually Exclusive and Collectively Exhaustive (MECE), this requires to make a clear (asymmetric) decision to which side pivot-ties go
- it is possible to provably optimally modify an asymmetric partitioning for detection of all-tied data (DIET-Method)
- it is possible to make asymmetric partitioning reliable against any adverse input pattern (FLIP-method)

DIET-method

Distinct Identification for Early Termination (DIET) works as follows: in the main loop of asymmetric binary partitioning, when searching for a pair of elements that require SWAPing, one pointer stops at pivot-ties, the other does not stop on pivot-ties. Assume the algorithm begins to search with the pointer that stops on ties. DIET adds a pre-loop that searches from the same side for a non-pivot-tied-key. If this pre-loop reaches the other side without finding a

---

slightly slower in the best case of tied keys, see also von Leitner (2007)

[4]our slightly better *Tricksort* removes a dead-branch from Yaroslavskiy's code

[5]Indeed a cycle: the optimization published by Edelkamp&Weiß (2016) was already suggested in a little known paper of Hoare (1962) in which he gives context and explanations for his 1961 algorithm

non-tie, all data are tied and the algorithm can exit recursion. If the pre-loop finds a non-tie, almost no work is lost and can be reused: only the last (non-tie) elements needs to be compared a second time against the pivot to find out to which side it belongs. I.e. we just set the pointer back one element and enter the main loop. That's it. And it provably costs exactly one extra comparison (and one extra pointer increments and decrement) per partitioning, hence less than N extra comparisons. That is the unavoidable price of tuning for early-termination on ties. If we use this DIET-partitioning in the usual self-recursive manner, we obtain *Zocksort*, which still is vulnerable for certain (asymmetric) inputs. Now we reached a similar point like Sedgewick, but we do not try to fix these beautifully efficient loops (or give up). Instead we FLIP.

FLIP-method

Fast Loops In Partitioning (FLIP) means to embrace the fast loops of the asymmetric partitioning and to create symmetry on the recursion level by left-right-mirroring. An input pattern that fools an asymmetric partitioning must itself be asymmetric, for example a *Zocksort* that partitions pivot-ties to the lower partition is fooled by data with two distinct keys, many high, and one low: random pivots are mostly high, hence all data goes to the left partition and the algorithm does not progress. Note that this data input cannot fool a Zocksort that partitions pivot-ties to the higher partition. By recursively alternating between an asymmetric partitioning and its left-right-mirrored twin, no asymmetric pattern can fool the resulting algorithm. Zig-zagging between the two twins is called *Zacksort* ('Zack' is the German word for 'zag' and implies a connotation of 'quick'). An elaboration is the *Zucksort* algorithm which flips the partitioning asymmetry only in the recursive branch that can contain the pivot-ties, in the other branch it does not matter ('Zuck' is the German word for 'twitch' hence also implies a connotation of 'quick').

POET-method

Pre-Order Early Termination (POET) is a tuning replacement of the DIET pre-loop in order to early-terminate on presorted data: instead of looping while equaling the pivot, the algorithm loops while the data does not violate the desired sorting order. If the loop reaches the other side the algorithm can early terminate, if it detects an out-of-order element, it resets the pointer to the starting point and enters the main loop. Note that Early-Termination on ties is a special case of POET. Note that POET unlike DIET is not provably optimal for early-termination on ties because the work of the pre-loop cannot be reused. The resulting algorithm is called *Ducksort*.

Partial sorting

Several variants of partial sorting are easily derived from *Quicksort*. A quite generic version (*Quickpart*) takes two parameters position `l` and position `r` in sorted order and guarantees sorted order between `l` and `r`. Other definitions of partial sorting are special cases thereof. For example the C++ `std::partial_sort` guarantees the smallest elements until position `middle` to be sorted, this is achieved by setting `l=middle` and `r=max` position. Specifying `l==r` gives the popular *Quickselect* published by Hoare as *FIND* (1961a; Hoare, 1961c). Note that *Quickselect* still does partial sorting work in order to guarantee that the l==r-th element is in the l==r-th position and that all elements smaller than the selected element are left of `l` and that all elements greater than the selected element are right of `r`. Note further that *Quickselect* does not define which element it selects in case of ties. But the well-defined tie-handling of *Zackselect*, *Zuckselect* or *Duckselect* returns the leftmost and rightmost positions of elements tied with the element at the desired position, for some tricky details see the code.

Branch-prediction

Like Block-Quicksort from Edelkamp and Weiß (2016) these *greeNsort®* algorithms such as *Ducksort*, *Duckpart*, *Duckselect* can be tuned to behave branchless, hence faster (named *DucksortB*, *DuckpartB*, *DuckselectB*). Note that *Pdqsort* of Orson Peters (2014; Peters, 2015; Peters, 2021b; Peters, 2021a) is a related asymmetric algorithm that when necessary involves a mirrored asymmetric partitioning. It does not achieve high-level symmetry and is not provably optimal, for example uses some heuristic shuffling. However Peters excellent C++ implementation is branchless with tuning for ties and tuning for presorted data and little overhead and it has been formally proven correct [Lammich:2020], hence for all practical purposes I can highly recommend it as mostly faster than the *greeNsort®* research implementations. However, once combined with an expensive comparison function such as localized string comparison `strcoll`, *Pdqsort* becomes slower than *DucksortB*. Note also that *Pdqsort* is implemented with deterministic pivots and fallback to potentially slower *Heapsort*, and note that the *greeNsort®* algorithms can be implemented like this as well, without any need for heuristic shuffling. Finally note that the simple branch-parallel *PDucksortB* outperforms *Pdqsort*, which has a serial dependence as the author states himself.[6]

---

[6]"I don't have plans currently. I would have to do some research on modern standard C++ parallel programming, and there are some tricky things in PDQsort if you want to parallelize. In particular it is assumed the left partition is recursed on first." https://github.com/orlp/pdqsort/issues/16#issuecomment-823145493

Quicksort conclusion

Embracing low-level asymmetry and addressing high-level bi-symmetry elegantly solves the decade-old Quicksort-dilemma in an easily provable way, which suggests applying symmetric recursion to other algorithms as well: to overcome the limitations of *Quicksorts*. While *Quicksorts* are memory parsimonious and have good cache-locality, they lack stability, deterministic worse-case guarantees and the ability to sort elements of varying size, and they are difficult to parallelize. Hence let's turn to *Mergesorts*.

Mergesort trilemma

In the Quicksort-dilemma there were trade-offs between efficiency and adaptivity. In merging there are trade-offs between efficiency, adaptivity and reducing buffer memory: one cannot have all three of them: full $\mathcal{O}(N \log N)$ efficiency and full $\mathcal{O}(N)$ adaptivity and low-memory.

Forgotten art

Let's rehearse how a generic efficient potentially parallel *Mergesort* is organized. Naive implementations allocate 100% buffer in the merge, copy the data to the buffer (100% moves), merge the two input streams back (100% comparisons and 100% moves) and deallocate the buffer. Obviously better is to allocate the buffer once before recursing and de-allocating once done, however what about the 200% moves? *Timsort* (Peters (2002)) allocates not more than 50% buffer, copies note more than 50% out and merges 100% back, that is 150% moves. Sedgewick (1990) taught a *Nocopy-Mergesort* that only merges (100% moves) without copying out by alternating between two memory regions of size 100% each.

Sedgewick teaches *Nocopy-Mergesort* with three loop-checks: two on the two input-streams and one on the output loop, obviously two loop-checks on the two input-streams ar enough. Sedgewick explains how to get away with one loop-check on the output streams by using sentinels in a mutual-recursive *Bitonic-Mergesort*, which sorts the left half ascending and the right half descending (from left), which is however not stable. Using the symmetric sorting definition we can fix this, by sorting the right half not descending from left but ascending from right which makes the algorithm nicely symmetric (*Bimesort*). However there are still disadvantages: this sentinel-approach must move data in each merge, hence cannot skip all merging for presorted data. Interestingly this sentinel-approach is not even necessary to get down to one loop-check: Knuth (1973) had teached 20 years earlier that one loop-check is sufficient, if only that input-sequence is checked for exhaustion from which the last element was merged. Combining this with *Nocopy-Mergesort* gives an efficient algorithm we name *Knuthsort* in honor of Donald Knuth. Note that Katajainen and Träff (1997) showed that the

number of loop-checks can be further reduced to ½ by investigating which of the two input-sequences exhausts first and only checking that one. Be warned that Katajainen's code reads beyond the last element, we fixed this and named the result *Katasort.* Note further, that Katajainen's optimization costs an extra comparison, hence we consider it tuned and prefer *Knuthsort* as the prior-art reference for efficient prior-art binary *Mergesort* in contiguous space.

For random data *Knuthsort* (and *Katasort*) are much more efficient than *Timsort.* Conventional wisdom assumes, that Timsort's inefficient merge is needed to reduce buffer to 50% and to achieve $\mathcal{O}(N)$ adaptivity to presorted data: after the merge *Timsort* has the data in the same memory-region as before the merge, hence it is easy to skip copying-out and merging-back in case of presorted keys. However, conventional wisdom is wrong, we can have one of the two without compromising on efficiency, either buffer-reduction or full adaptivity. Let's start with the latter.

Full adaptivity

The alternating merge of *Nocopy-Mergesort* finally completes in the data memory, not in the buffer. However, for odd recursion depths this starts merging from the buffer memory, hence Sedgewick copies the data to the buffer memory before recursion. Our *Omitsort* no longer uses a pre-determined alternating merge, instead we let the merge functions decide whether it merges (or whether it can omit the merge in case of presorted data): the merge function simply returns the location of the data and the sort function checks whether after its two merges the data are in the same memory region, if not, it copies one to the data region. For fully presorted data no moves are needed anymore, not even Sedgewicks initial moves. *Omitsort* uses $\mathcal{O}(N)$ comparisons for diagnosing presortedness as non-overlap of the two input sequences.

For descending data, an ascending *Omitsort* cannot omit: each merge must do its work, hence the total cost are $\mathcal{O}(N \log N)$. Cheaper would be to do nothing and only reverse the total sequence after checking presortedness throughout the recursion. Our *Octosort* achieves this by no longer requiring a specific order during the merging: the resulting order is simply data-driven and returned by the recursive sort function. Only if the left and right recursive sorts return different directions, the desired (ascending) order is enforced. For perfectly ascending and descending data the are no moves during the recursion, for descending data only a final reversal is needed, hence *Octosort* is like *Timsort* $\mathcal{O}(N)$ bi-adaptive for pre-sorted data, but unlike *Timsort* much more efficient for non-sorted data. *Octosort* can also be fully parallelized, while *Timsort* has inherently serial tasks.

Distance minimization

> "For decades, the machine balance of compute speed to memory
> bandwidth has increased 15%–30% per year [. . .] Projections for
> future machines only exacerbate the current data movement crisis"
> (Dongarra, 2022).

We all have learned that sorting cost is $\mathcal{O}(N \log N)$ . . . for constant access costs
in the RAM-model. We all know that the RAM-model is wrong and access costs
are not constant: todays memory hierachies are deep, L1, L2, L3, RAM on local
socket, RAM on foreign socket, and todays virtualization and cloud techniques
might even give us memory on different machines, in different LAN and across
WAN networks.

The traditional answers to the data movement crisis are k-ary algorithms (which
reduce the number of memory passes), block-access (which is only suitable for
one of the cache-layers) and cache-oblivious algorithms (which are complicated).
greeNsort® takes a different perspective, the perspective of an *Ordinal Machine
Model*: there is a cost for moving data over distance, but the exact cost of a
move (or access) over distance d is unknown. It could be anything between $\mathcal{O}(1)$,
$\mathcal{O}(\log d)$ and $\mathcal{O}(d)$. An example for the latter - linear move costs - would be
sorting N cars in a row in a parking slot next to a road. Let's assume another
free N positions of buffer parking space. A buffer next to the N cars is the best
we can hope for, hence merging N cars to the buffer space costs moving each car
on average N positions on each recursion level, which gives us total move cost of
$\mathcal{O}(N^2 \log_2 N)$ for all cars on all recursion levels. That's expensive.

Note that even *K-ary Mergesort* for our cars still would be quadratic:
$\mathcal{O}(N^2 \log_k N)$. Contrast this to *Quicksort*, where the move distance is halved
on each partitioning in the recursion, therefore the total move costs is $\mathcal{O}(N^2)$.
This is a little appreciated reason why *Quicksort* performs robustly on many
different machines. In other words, *Quicksort* zooms into local memory, while
Mergesort keeps merging over global distances.

Gapped-merging

A simple trick brings locality to *Mergesort*: for N elements of data take $2N$
elements of memory and alternate merging between odd and even positions,
this reduces the move distances as the algorithm divides deeper. Unfortunately
gapped-merging comes with a couple of disadvantages: reading N elements
actually scans $2N$, hence gapped *GKnuthsort* is on current CPUs slower than
Knuthsort. Also gapped-merging requires all elements to have the same size.

Buffer-merging

Locality requires starting the merging with local buffer gaps between the data. If we want a contiguous result of the sort with only two regions, data and buffer, this requires that we merge not only the data but also the buffer. Rehearse this: buffer becomes a first-class object of merging. Understand that Divide&Conquer *with* buffer and good locality requires Divide&Conquer *of* buffer. Once this is understood, it looks really weird, that in *Mergesort* for decades we only merged data but not buffer. Now let's look at some ways to merge buffer. Let uppercase letters represent data and lowercase represent buffer.

If the result of buffer merging is one contiguous data region and one contiguous buffer region, we have to make a fundamental asymmetric choice: data left and buffer right (`Ab`) or data right and buffer left (`aB`). If we assume `Ab` before splitting and after merging, standard self-recursion implies to replicate that pattern across all recursion levels, i.e. we merge from `AbCd` to `ACbd`. In order to merge `A` and `C`, a naive approach would first transport all input streams to the right: `bdAC` and then merge back to `ACbd`. 100% extra moves are expensive (*TKnuthsort*). A more efficient approach would only relocate the streams in the left half to the gaps in the right half `dbCA` before merging to the left `ACbd`. Note that the order of the streams has changed, hence care is needed to retain stability when merging (*Crocosort* with Knuth's merge). Unfortunately, with 50% extra moves, this is still as inefficient as *Timsort*.

Symmetric merging

It is possible to get rid of any extra moves if we leverage symmetric mutual recursion: if in the left branch we sort the data to the left (buffer to the right) and in the right branch we sort the data to the right (buffer left). What we obtain is the data in the outer regions and the buffer in the inner regions: the two inner buffer regions are contiguous without any extra moves. That's nice, but even nicer is that not more than 50% buffer is needed: merging is done from inner to outer input-streams such that the result is aligned at the outer border. This semi-inplace merging has the following adaptivity properties: for perfectly presorted data the number of comparisons and moves is automatically reduced to 50%, with a non-overlap test the number of comparisons can be reduced to 0%, but at lat least 50% of the data must be moved. Symmetric merging offers two variants: a symmetric variant that sorts one branch left-ascending and one branch right-ascending (*Geckosort*) and a asymmetric variant which uses the same order in both branches (*Frogsort*). *Geckosort* is 25% adaptive to ascending and 25% adaptive to descending keys. *Frogsort* is 50% adaptive to presorting in the implemented order. Note that *FROG* is an acronym of *Flip Recursive Organized Gaps.*[7]

---

[7]Gaps are crucial for stable sorting. As late as 2006, Bender et al. (2006) published *Librarysort*, a Gapped Insertion Sort. Like people leave gaps between books in bookshelves,

Frogsort0

The simple (and first) variant of *Frogsort* was *Frogsort0* (2010): it operates on *triplets* of memory elements, one buffer element symmetrically in the middle between two data elements, i.e. `AbC`. *Frogsort0* splits and merges the triplets. Rule: if there is an odd number of triplets, the surplus triplet goes to the outer side. The setup of the gapped elements can be done before Split&Merge.

Table 1: Sketch of Frogsort0

position	1	2	3	4	5	6	7	8	9
setup	A	b	C	D	e	F	G	h	I
merge	A	C	b	e	D	F			
merge	A	C	D	F	b	e	h	G	I
merge	A	C	D	F	G	I	b	e	h

Odd number of elements can be handled by using one extreme dummy value (see the implementation of *Frogsort0*) or by a specific third recursion function that handles the outermost incomplete triplet (see the implementation of *PFrogsort0*).

Frogsort1

An alternative is splitting single elements in *Frogsort1*. Rule 1: if there is an odd number of elements, the surplus element goes to the outer side. Rule 2: the size of the buffer is $N/2$ (integer division). The setup is done in an extra recursion before the Split&Merge Recursion.

Table 2: Sketch of Frogsort0

position	1	2	3	4
top-2	A	b	C	
top-1	A	C	b	D
top	A	C	D	b

Frogsort2

Textbook knowledge tells us that Divide&Conquer should be balanced in order to minimize operations. Symmetric merging allows to reduce the buffer to much less than 50%: let the inner branch be $p\%$, then symmetric merging needs only $p\%$ buffer. Yes, this increases the maximum recursion depth and the total number of

---

gaps reduce insertion costs from $\mathcal{O}\left(N^2\right)$ to amortized $\mathcal{O}(N \log N)$ in the RAM-model. While the usage of gaps in *Librarysort* is rather probabilistic, *Frogsort* uses gaps in a deterministic, 'Organized' way.

operations, but it reduces the total %RAM. Hence for the sustainable Footprint measure, we expect an U-shaped function of p. *greeNsort®* implemented this as *Frogsort2*. For sorting doubles, surprisingly, not only exists an optimal p below 10% regarding Footprint, there is also an optimal p below 20% regarding speed. This is surprising given usual space-speed trade-off expectations, that less RAM implies longer runTime. One reason for this is: imbalanced merging makes for better branch-prediction. With *Frogsort2* we have a nice algorithm, that can be tuned to specific hardware features using its parameter p. Note that *Frogsort1* is a special case of *Frogsort2* at $p = 0.5$.

Frogsort3 and 6

So far we have *split* the buffer between the two branches. An alternative is to *share* the buffer, i.e. first we send all the buffer down the left branch, and then we send the buffer down the right branch. This allows to use even less buffer. *Frogsort3* does this, until there is enough buffer at a branch to switch to *Frogsort1*. *Frogsort6* does this, until there is enough buffer at a branch to switch to *Frogsort2*. *Frogsort1* is a special case of *Frogsort3* with parameter $p = 0.5$. *Frogsort6* has two parameters $p3$ and $p2$, and *Frogsorts* 1,2,3 are special cases of *Frogsort6*.

Squidsort

Frogsort saves 50% compares and moves for presorted data, but not for reverse-sorted data. *Geckosort* saves 25% compares and moves for presorted data and for reverse-sorted data. Tuning *Frogsort* with a non-overlap comparison for presorted data reduces all other compares to 0%. Combining *Frogsort* with the data-driven lazily enforced order of *Octosort* gives *Squidsort*, which needs 0% comparisons and 50% moves for presorted and reverse-sorted data (see *Squidsort1* and *Squidsort2*). *Squidsort* beats *Timsort* and related algorithms such as *Peeksort* and *Powersort* by Munro and Wild (2018), unless data is extremely presorted.

Mergesort Conclusion

Embracing low-level asymmetry and addressing high-level bi-symmetry elegantly improves the trade-offs of the decade-old Mergesort-trilemma. The symmetric definition of sorting guides the development of stable algorithms, and exploiting buffer-asymmetry in symmetric-merging (and stable symmetric partitioning) allows to reduce the amount of buffer required, in some cases even using less energy and sorting faster (*Frogsort2*).

Further algorithms

Low-memory merging

In-place merging algorithms have enjoyed huge academic attention. Several in-place Mergesorts are known for decades but were too slow in practice. In 2008 (or earlier) Sedgewick created the term "holy sorting grail" for a practically usable stable $\mathcal{O}([N \log N)$ in-place-Mergesort. In 2013 Astrelin (2013) published *Grailsort* an implementation of Huang and Langston (1992) that was not dramatically slower than other *Mergesorts*, hence deserved its name. Since 2014 Sedgewick redefined the "holy sorting grail" and required a $\mathcal{O}(N)$ best case. Anyhow, in-place merging is overrated, Astrelin also published *Sqrtsort* which is faster than *Grailsort* and needs only a practically negligible $\mathcal{O}\left(\sqrt{N}\right)$ buffer.

The *greeNsort®* algorithm *Walksort* also requires $\mathcal{O}\left(\sqrt{N}\right)$ buffer, is faster, and its equally fast variant *Jumpsort* uses relocation-moves for distance minimization (like *Crocosort*).

Regarding speed and energy for sorting random data, these $\mathcal{O}\left(\sqrt{N}\right)$ buffer algorithms are inferior to *Frogsort* and *Squidsort*. Regarding Footprint the ranking for sorting random doubles is *Frogsort2 < Squidsort2 < Frogsort3 < Walksort < Jumpsort < Frogsort1 < Squidsort1 < Sqrtsort < Grailsort < Knuthsort*. For pre-sorted data the ranking of *Walksort* and *Jumpsort* improves: they are quite adaptive.

Partition&Pool

Some of the *greeNsort®* learnings can be transferred from *Split&Merge* to *Partition&Pool* algorithms. It is possible to turn *Zacksort* into the stable *Kiwisort* algorithm using 100% distant buffer. By partitioning not only data but also 100% local buffer we get the distance reducing *Swansort*. For a very special use case it is even possible to reduce to 50% buffer, see *Storksort*.

Size-varying

A popular method for sorting elements of varying size - such as strings, bigints or arbitrary objects with a key field - is indirect sorting of pointers to elements using *Quicksort*, or better one of *Zacksort*, *Zucksort* or *Ducksort*. However, this is inefficient because it incurs heavy random access (see *UZacksort*, and even more inefficient if the sort is stabilized by breaking ties by comparing pointer address values, see *WQuicksort2*). An alternative is directly sorting size varying elements using a buffer. Splitting can be done either by helper-pointers to elements or directly splitting at null-terminators (for the latter see *VKnuthsort* and *VFrogsort1*). These direct sorts are more efficient than the indirect Quicksorts. Only if there are many string duplicates, unstable *UZacksort* benefits from its $\mathcal{O}(N \log D)$ early termination. Hence it seems promising to implement direct stable Partition&Pool algorithms for size-varying data (*VKiwisort* and *VSwansort*, not yet implemented).

Parallel

Doing a task in parallel can save energy, Intel calls this "race to idle". All Divide&Conquer algorithms are easily implemented with parallel branches (see *PQuicksort2*, *PQuicksort2B*, *PDucksort*, *PDucksortB*). However, the partitioning of the *Quicksort* family is difficult to implement in parallel, and this involves trade-offs. By contrast, parallelizing binary merges is relatively straightforward. *greeNsort®* uses a method that allows to parallelize over an arbitrary number of processes, not only power of two processes, see *PKnuthsort*, *PFrogsort0*, *PFrogsort1*, *PFrogsort2*, *PFrogsort3*, *PVKnuthsort*, *PVFrogsort1*. As expected, parallel execution reduces not only runTime but also Energy (although to a lesser extent).

Note that the parallel speed-up of the *Frogsorts* scales almost as linear with the number of cores like the speed-up of *Knuthsort*, hence the Footprints of *PFrogsorts* are clearly lower than that of *PKnuthsort*.

Note that the benefit of *Frogsort2* over *Frogsort1* somewhat diminishes when more parallel cores are used.

Note that the linear setup-phase of *Frogsort0* somewhat better parallelizes than the recursive setup of *Frogsorts1,2,3,6*. Hence it is promising to implement variants of *Frogsorts2,3,6* which leverage a linear setup with chunks of a predefined mixture of data and buffer elements.

Results

Methods and measurement

For energy measurement we use the RAPL counters of the linux `powercap` kernel module, see `lib_energy.h` and `lib_energy.c`. All measurements reported here are done on an Intel i7-7700 CPU under ubuntu.20.04 with the 5.13.0-44-generic kernel and compiling our testbed with gcc.9.4.0. The CPU is run with hyper-threading switched of in the bios. The algorithms are measured on the following input data patterns:

- `permut`: randomly permuted numbers from $1 \ldots n$
- `tielog2`: random sample of $\log_2 n$ distinct values
- `ascall`: n distinct ascending numbers
- `asclocal`: n distinct numbers randomly put into $\sqrt{n}$ presorted sequences of length $\sqrt{n}$
- `ascglobal`: n distinct numbers cut into ascending $\sqrt{n}$ quantiles of length $\sqrt{n}$ and randomly permuted per quantile

Measurements for these 5 patterns are averaged to a `TOTAL` KPI for ranking of algorithms. Furthermore the following 3 patterns are measured

- `descall`: n distinct descending numbers
- `desclocal`: n distinct numbers randomly put into $\sqrt{n}$ reverse-sorted sequences of length $\sqrt{n}$
- `descglobal`: n distinct numbers cut into descending $\sqrt{n}$ quantiles of length $\sqrt{n}$ and randomly permuted per quantile

The descending patterns are interesting with regard of the symmetry of adaptivity, but not included in the `TOTAL` KPI. We believe that adaptivity to ascending data is more important than to descending, furthermore, including them would give too much weight to easy patterns.
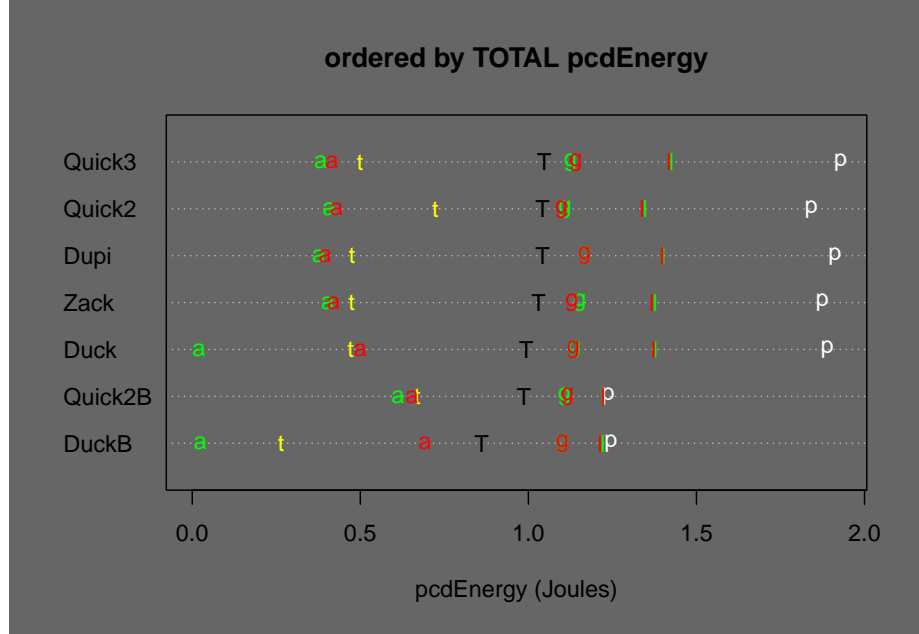
35

Quicksort results

Figure 4: Medians of Quicksort alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

Table 3: Ducksort / Quicksort2 (ratios of medians and p-values from two-sided Wilcoxon tests)

	r(%M)	r(rT)	r(pcdE)	d(pcdE)	p(rT)	p(pcdE)
TOTAL	1	0.95	0.95	-0.05	0	0
ascall	1	0.05	0.05	-0.39	0	0
descall	1	1.17	1.16	0.07	0	0
ascglobal	1	1.02	1.02	0.03	0	0
descglobal	1	1.02	1.03	0.04	0	0
asclocal	1	1.02	1.02	0.03	0	0
desclocal	1	1.02	1.03	0.03	0	0
tielog2	1	0.66	0.65	-0.25	0	0
permut	1	1.02	1.03	0.05	0	0

Table 4: Ducksort / Quicksort3 (ratios of medians and p-values from two-sided Wilcoxon tests)

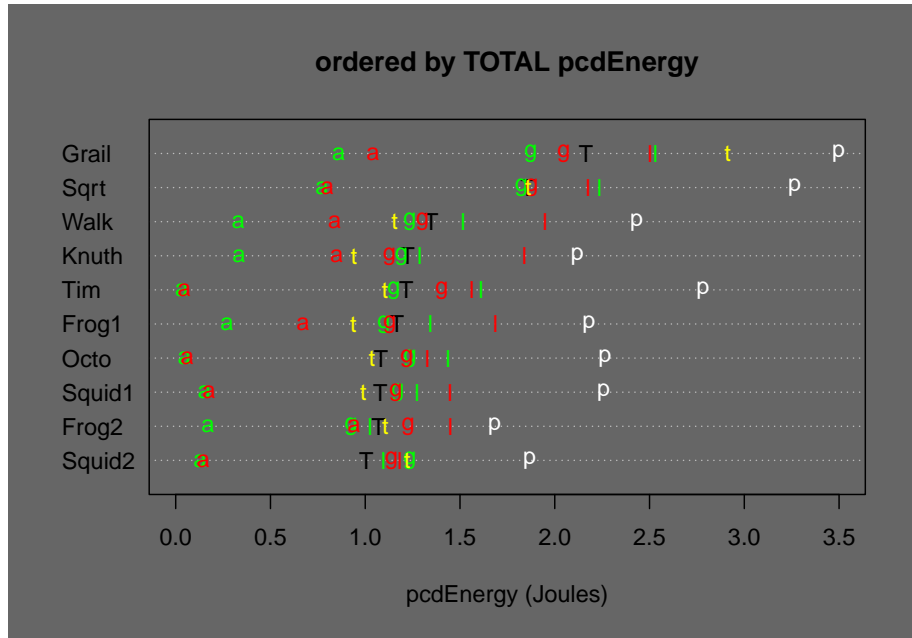|	r(%M)	r(rT)	r(pcdE)	d(pcdE)	p(rT)	p(pcdE)
TOTAL	1	0.96	0.95	-0.05	0	0.0000
ascall	1	0.05	0.05	-0.36	0	0.0000
descall	1	1.20	1.20	0.08	0	0.0000
ascglobal	1	1.02	1.01	0.01	0	0.0992
descglobal	1	1.00	0.99	-0.01	0	0.0160
asclocal	1	0.97	0.97	-0.05	0	0.0000
desclocal	1	0.98	0.97	-0.05	0	0.0000
tielog2	1	1.02	0.95	-0.03	0	0.0000
permut	1	0.98	0.98	-0.04	0	0.0005

Mergesort results

Figure 5: Medians of Mergesort alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal
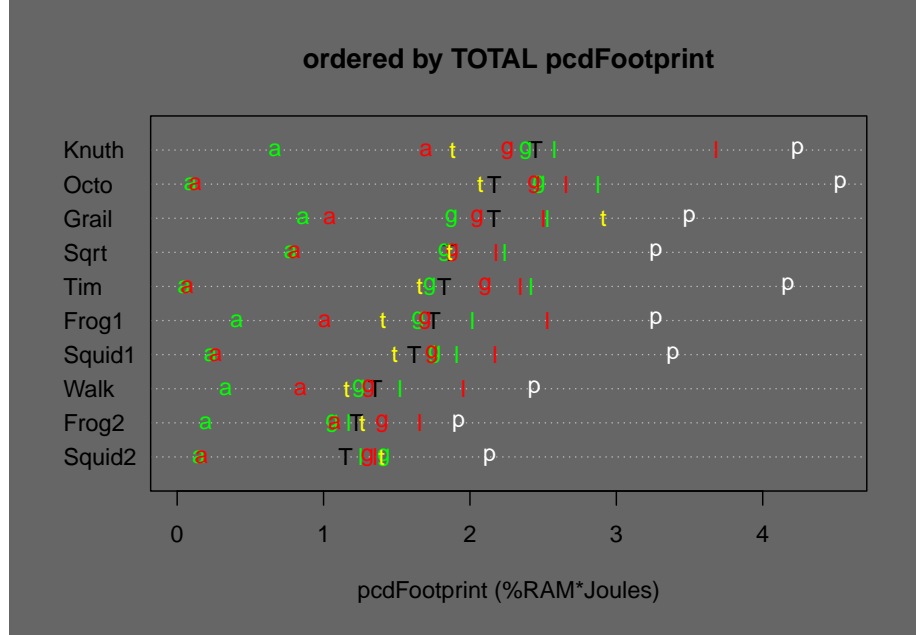
Figure 6: Medians of Mergesort alternatives ordered by TOTAL eFootprint.
T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red:
a=descall, g=descglobal, l=desclocal

Table 5: Squidsort2 / Knuthsort (ratios of medians and p-values from two-sided
Wilcoxon tests)

	r(%M)	r(rT)	r(pcdE)	r(pcdF)	d(pcdE)	p(rT)	p(pcdE)	p(pcdF)
TOTAL	0.57	0.81	0.82	0.47	-0.22	0.0000	0.0000	0
ascall	0.57	0.38	0.38	0.22	-0.21	0.0000	0.0000	0
descall	0.57	0.15	0.17	0.10	-0.70	0.0000	0.0000	0
ascglobal	0.57	1.04	1.04	0.59	0.04	0.3595	0.4566	0
descglobal	0.57	1.00	1.01	0.58	0.01	0.1750	0.0023	0
asclocal	0.57	0.84	0.85	0.49	-0.19	0.0000	0.0000	0
desclocal	0.57	0.64	0.64	0.37	-0.66	0.0000	0.0000	0
tielog2	0.57	1.30	1.30	0.74	0.28	0.0000	0.0000	0
permut	0.57	0.87	0.88	0.50	-0.25	0.0000	0.0000	0

Related work

K-ary sorting algorithms can dramatically reduce movements compared to binary algorithms. Radix sorting algorithms go further and also reduce comparison cost, but are only available for certain data types and collation orders. Axtmann et al. (2017) have developed the k-ary *In-place Parallel Super Scalar Samplesort (IPS4o)* with the motivation to provide a more efficient alternative to quicksort variants in standard libraries. Their recent study improved and compared it to other algorithms including radix sorts and shows that *IPS4o* outperforms binary and other k-ary algorithms for sufficiently large data sets (Axtmann et al., 2022). Also their radix algorithms outperformed the quit generic library of Skarupke (2016). This C++ library of the KIT is highly recommended. Some limitations of *IPS4o* are its code complexity, that it is not stable and its limited adaptivity to presorted data.

Future work

Algorithms

We expect that it is possible to make *IPS4o* stable using the permutation logic
of our *Walksort*. We suspect, *Squidsort* could be made even more adaptive
by replacing the static splits by searching for a optimal splitting point as in
Peeksort. For elements of varying size that have no easily searchable separator
like the null-terminator in strings, versions of *Frogsorts* should be developed
which use pointers to elements (for direct sorting). Finally, we believe that
parallel *Frogsorts* 2,3,6 can be written using predefined chunks with a fixed
number of data and buffer elements. Beyond sorting, we would not be surprised,
if the symmetric recursion and particularly Divide&Conquer with buffer could
be used to design other algorithms, e.g. clustering.

Sorting APIs

Current sorting APIs limit the efficiency of sorting, for example C's library
(Quicksort) expects a ternary comparison function (using two binary comparisons
under the hood), where an API expecting a binary comparison function would
allow to sort with less comparisons. Optimal sorting APIs is an under-explored
topic.

Code-Mirroring

In the testbed we have mirrored code sections by hand. In order to reduce the
manual work of code-mirroring and to make errors less likely, code-mirroring
could be done in meta-programming or programming languages accompanied
with IDEs that support code-mirroring. In order to reduce binary code size,
we envision code-mirroring CPU-instructions and compiler support. This is a
promising novel field of research.

Call to action

Despite interesting open research questions, the currently available portfolio of algorithms - particularly from KIT and *greeNsort®* - is rich enough to provide more sustainable alternatives for older established algorithms. In times of climate crisis it is imperative to reduce the energy and hardware consumption now. Hence we call all maintainers of sorting code, in languages, libraries and software to review their sorting-requirements and replace their algorithms with more sustainable ones. For further guidance turn to *greensort.org* and *github.com/greensort*.

This call to action is not limited to researchers and software-engineers, we also call all teaching computer scientists to update their didactic materials on algorithms and sorting such as textbooks, MOOCs etc. The power of the *greeNsort®* methods and insights should be available to future generations of computer scientists without restriction from the outset.

Conclusion

greeNsort® presents a new perspective on binary divide&conquer sorting: *Simple Symmetric Sustainable Sorting* deliberately departs from the state of the art and previous habits, enabling new algorithms. Our *Footprint* KPIs combine traditional operational costs (time, energy) with hardware investment requirements (memory) and enable ranking of algorithms with different memory requirements. Our symmetric definition of sorting (four stable API targets instead of two) clarifies how to write robust algorithms. Our ordinal machine model promotes the development of robust algorithms by focusing on minimizing distances. New algorithms are enabled in an enlarged solution-space by embracing asymmetry (of partitioning and of loops in von-Neumann machines) and delegating symmetry into (mutual) recursion design. Our main results are are new Quicksort which solves the *Quicksort-Dilemma* and a new Mergesort which resolves the *Mergesort-Trilemma* (and novel variations of those algorithms).

60 years after Hoare's *Quicksort*, we present the algorithm that Hoare wanted to invent: a symmetric probabilistic algorithm that early terminates on ties without compromising efficiency. *Zacksort* resp. *Zucksort* combine the symmetric *FLIP* method with the lean *DIET* method, which is provably optimal. It is possible to integrate adaptivity on presorted data: *Ducksort* uses the *POET* instead of the *DIET* method: good in practice but no longer optimal.

75 years after John von Neumann's *Mergesort*, we present stable algorithms that combine the buffer-locality of *Quicksort* with the generality of *Mergesort*. Instead of 100% distant buffer, *Frogsort* and *Geckosort* need only 50% (or less) local buffer. Furthermore they are automatically half-adaptive to presorted data, and we show how to achieve bi-adaptivity to pre-sorted and reverse-sorted data: *Octosort* and *Squidsorts* embrace existing order and only lazily enforce the desired order. Unlike natural Mergesorts such as *Timsort* which are optimized for the best case of full-presorting, the *greeNsort®* algorithms are optimized for the worst case of real sorting work and can be parallelized to multiple CPU-cores. They also can be implemented for size-varying elements (unlike Quicksort, which incurs expensive random-access when used indirectly). Our low-memory algorithms *Walksort* and distance-reducing *Jumpsort* have lower Footprint than the impressive *Grailsort* and *Sqrtsort* of Astrelin, and hence allow resilient use of old hardware with little RAM.

Author & Project

Jens Oehlschlägel is a psychologist and Ph.D. statistician who has programmed computers since 1977. He has worked in different roles and industries, including 10 years as Data Scientist at McKinsey. Jens has done innovative work in psychological testing (*Frankfurter Aufmerksamkeits-Inventar*), statistical software (*bit*, *bit64*, *ff*) and cluster-analysis meta-algorithms (*truecluster.com*). Since 2011 Jens is a member of the ACM, works as a database architect and devotes his spare time to the *greeNsort®* project.

For more details, illustrations and measurement results see the *greeNsort®* Innovation-Report and Code at *github.com/greeNsort*. For further information consult *greensort.org*. For news please follow us on twitter at *@greeNsort_algos*, or if twitter goes bankrupt, on mastodon at @greeNsort@scicomm.xyz .

greeNsort® is a protected trademark and may not be used without permission. For free open-source software there will be a self-certification program, that will allow those projects to promote their use of *greeNsort®* algorithms. Beyond the self-certification program, for consulting or certification with the *greeNsort®* brand or logo, please contact us by mail at *consulting[at]greensort.eu* or *certification[at]greensort.eu* .

Bibliography

Astrelin, A. (2013). Grailsort.

Axtmann, M., Witt, S., Ferizovic, D., and Sanders, P. (2017). In-Place Parallel Super Scalar Samplesort (IPSSSSo). In Pruhs, K. and Sohler, C., editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Axtmann, M., Witt, S., Ferizovic, D., and Sanders, P. (2022). Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.*, 9(1).

Bender, M. A., Farach-Colton, M., and Mosteiro, M. A. (2006). Insertion sort is o(n log n). *Theory of Computing Systems*, 39(3):391–397.

Bentley, J. L. and McIlroy, M. D. (1993). Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265.

Dongarra, J. J. (2022). The evolution of mathematical software. *Commun. ACM*, 65(12):66–72.

Edelkamp, S. and Weiß, A. (2016). Blockquicksort: How branch mispredictions don't affect quicksort. *CoRR*, abs/1604.06697.

Hoare, C. A. R. (1961a). Algorithm 63: Partition. *Commun. ACM*, 4(7):321.

Hoare, C. A. R. (1961b). Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321.

Hoare, C. A. R. (1961c). Algorithm 65: Find. *Commun. ACM*, 4(7):321–322.

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, 5(1):10–16.

Huang, B.-C. and Langston, M. A. (1992). Fast stable merging and sorting in constant extra space. *j-COMP-J*, 35(6):643–650.

Katajainen, J. and Träff, J. L. (1997). A meticulous analysis of mergesort programs. In Bongiovanni, G., Bovet, D. P., and Di Battista, G., editors, *Algorithms and Complexity*, pages 217–228, Berlin, Heidelberg. Springer Berlin Heidelberg.

Knuth, D. (1973). *The Art Of Computer Programming, vol. 3: Sorting And Searching.* Addison-Wesley.

Knuth, D. E. (1998). *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Mehlhorn, K. (1977). *Data Structures and Algorithms*, volume 1: Sorting and Searching. Teubner, Stuttgart, Leipzig.

Munro, J. I. and Wild, S. (2018). Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs. In Azar, Y., Bast, H., and Herman, G., editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 63:1–63:16, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Peters, O. (2014). [patch] bug 20837 - libc++ std::sort has o(n²) worst case, standard mandates o(n log(n)). llvm bug report.

Peters, O. (2015). pdqsort. github code.

Peters, O. R. L. (2021a). Agreed on pdqsort ... Hacker News.

Peters, O. R. L. (2021b). Pattern-defeating quicksort. *CoRR*, abs/2106.05123.

Peters, T. (2002). I wrote a sort for python. 'Sorting' mail to the Python-Dev list.

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Inf.*, 7:327–355.

Sedgewick, R. (1990). *Algorithms in C.* Addison-Wesley Professional, 2 edition.

Singleton, R. C. (1969). Algorithm 347: An efficient algorithm for sorting with minimal storage [m1]. *Commun. ACM*, 12(3):185–186.

Skarupke, M. (2016). I wrote a faster sorting algorithm.

The UK Parliamentary Office of Science and Technology (2022). Energy consumption of ict.

von Leitner, F. (2007). diet libc - a libc optimized for small size.

Wegner, L. M. (1985). Quicksort for equal keys. *IEEE Trans. Computers*, 34(4):362–367.

Yaroslavskiy, V. (2009). Dual-pivot quicksort.