Symmetric Sustainable Sorting — the greeNsort® innovation-report

Jens Oehlschlägel

2024-02-02

"In the opinion of the German electrical and digital industry, greater energy efficiency and energy saving are at least as important as the expansion of the green electricity grid"

"Eine höhere Energieeffizienz und mehr Energiesparen sind nach Auffassung der deutschen Elektro- und Digitalindustrie mindestens ebenso wichtig wie der Ausbau des Ökostromnetzes."

source: Süddeutsche Zeitung

Contents

Content

This *greeNsort*® report describes the innovations of the *greeNsort*® project more detailed than the *greeNsort*® article: the role of symmetry in divide&conquer sorting algorithms. The report describes the motivation for the *greeNsort*® project and the *greeNsort*® perspective on innovation. It defines a taxonomy for the innovations, describes and classifies them and clarifies the dependencies between those innovations (and to prior art). After introducing innovative techniques and measurement methods, the report dives into Quicksorts and develops the new *Zacksorts*, dives into Mergesorts and develops the new *Frogsorts* and algorithmic variations thereof, and finally transfers the learnings to stable partitioning algorithms. The report then expands on parallel and incremental sorting, before discussing the potential impact of these innovations. The report closes with Conclusion&Outlook, which also serve as summary of the summaries of the covered sections. The appendix contains several glossaries and bibliography.

Note that despite its length, the report is still giving parsimonious description of the *greeNsort*® innovations written for subject matter experts – *(P)erson (H)aving (O)rdinary (S)kill (I)n (T)he (A)rts (PHOSITAs)* – in the art of sorting. In case of doubt consult the C-code. A textbook is being prepared for teaching students that does not require any specialist knowledge.

Chapter 1

Introduction

1.1 Document-Structure

- in the *Introduction* I explain my *Motivation&Scope*, the *greeNsort® Perspective on Innovation* and its *Innovation-Taxonomy*
- in the section *Paradigmatic innovations* I describe the fundamental *greeNsort®* innovations before diving into particular algorithms
- in the section *Methods&Measurement* I explain the employed techniques for measuring energy and other KPIs as well as the analytic methods used
- in the section *Quicksort Algorithms* I describe the *The Quicksort-Dilemma*, its *greeNsort®* solution and explain consequences for partial sorting. A more detailed description including code-examples is in the Quicksort draft chapter of the *greeNsort®* book.
- the *Split&Merge Algorithms* section is the main part of *greeNsort®*: prior art reference algorithms and the *The Mergesort-Dilemma* are explained, then more memory parsimonious *greeNsort®* algorithms are developed step-by-step and variations of the algorithms are explained.
- the *Partition&Pool Algorithms* section briefly transfers the learning of the *Quicksort Algorithms* and *Split&Merge Algorithms* sections to stable sorting via partitioning.
- the *Parallel sorting* section explains and analyzes parallel versions of some prior art and *greeNsort®* algorithms
- the *Incremental sorting* section explores implications for sorting a stream of data INSERTs and data structures which allow DELETEs and queries
- the *Impact* section gives an outlook for the impact of the *greeNsort®* innovations in various areas
- the *Conclusion&Outlook* section gives a summary of the report
- the *Author&Project* section explains who am I and how to reach me
- the final *Terms&Tables* sections list important classes of terms (*Definition&Convictions*, *Bold techniques*, *Algorithms*, *Extended&Further*), and

then *Abbreviations*, *Glossary* and *[Bibliography]*.

1.2 Motivation&Scope

Sorting is a fundamental operation in IT, in my eyes even more fundamental than hashing, see the empirical tests in Oehlschlägel and Silvestri (2012). Sorting emits CO2 by consuming electricity during operation and by requiring certain hardware that embodies CO2. The idea of *greeNsort®* is to bring CO2 savings to thousands of pieces of software and to billions of devices without a need to re-write software or a need to produce new hardware. The idea of *greeNsort®* is to reduce the CO2 during sorting and to reduce the required hardware, and to develop better sorting-algorithms that are so simple, that they can easily be implemented and replace inefficient algorithms that today are found in many central libraries of popular programming languages.

The core scope of *greeNsort®* is *simple general binary sorting in contiguous space with Divide&Conquer algorithms that can be described by recursion.* Why *general*? Because I don't want to optimize just a special case, there are already some very efficient sorting-algorithms for special cases, for example radix-sorting algorithms. Why *simple*? Because simplicity enables quick low-cost implementation. Why *binary*? Because binary is simple, teaching sorting starts with binary and ends with k-ary; there are already some very efficient k-ary sorting-algorithms, but those are more complicated and pay-off only on hardware with particular properties. Why *contiguous space*? Because contiguous space is simple and algorithms operating in contiguous space can be designed to have good cache-properties; algorithms that operate in block-managed-memory[1] can reduce hardware-memory requirements, but they are more complex and introduce random-access on block-level. Why *Divide&Conquer*? Because Divide&Conquer guarantees that the costs of sorting are limited at $\mathcal{O}(logN \cdot f(N))$, for example $\mathcal{O}(logN \cdot N)$ in the random-access-model. Why *by recursion*? Because recursion is an important *mental method* for developing algorithms, teaching algorithms, understanding algorithms and analyzing algorithms[2].

Having said this, out-of-scope are for example *in-place-merging* such as *Grailsort* (complicated and slow), algorithms such as *Funnelsort* that leverage the continuous van-Emde-Boas-Layout (k-ary, complicated and not practical for N that is not a power of two), *Lazy-Funnelsort* (k-ary, not contiguous) or *(I)n-place (P)arallel (S)uper (S)calar (S)ample(So)rt (IPS4o)* (k-ary, complicated, not contiguous, not completely general, not yet stable). All of these have been academically successful, none of these has found widespread adoption in core-libraries. *greeNsort®* does not claim to beat these algorithms in their domain, although I expect that joining forces could create synergies, for example result in a stable IPS4o with still excellent CO2-properties.

[1] although with *Jumpsort* and *Walksort greeNsort®* has two particularly efficient algorithms that use random access to blocks of data (not strictly contiguous)

[2] notwithstanding the fact that each recursive algorithms can be re-written in loops

1.3 Perspective on Innovation

greeNsort® is a *web of interrelated intangible innovations* with the intention to *forever alter the way how sorting is taught and used.* The changes are fundamental and deconstruct many prior art beliefs that stand since 1945 resp. 1961, hence from an academic perspective, it could be appropriate to classify *greeNsort®* as a *Paradigm Change* in the sense of Kuhn (1962). But could it be a *Leapfrog Innovation*? I believe *greeNsort®* is a leapfrog innovation, in spite of three aspects of leapfrog innovations that I believe are myths about leapfrog innovations. Let's begin with common ground: *Leapfrog innovations are innovations that change the world.*

1.3.1 Are leapfrog innovations tangible products?

A common myth about leapfrog innovations is that they are *tangible products.* For example the list of published in the Atlantic *The 50 Greatest Breakthroughs Since the Wheel* mentions almost exclusively tangible innovations (45), with the exception of three borderline cases (*paper money*, *electricity* and *the internet*) and two clearly intangible innovations (*alphabetization*, *the Gregorian calendar*). Here is a counterexample missing from the list without the world today would not be possible: *the logarithm*, I could list hundreds of such intangible innovations that shaped the world as we know it today. Tangible innovations are just the visible part of the iceberg, they are born out of an interrelated web of intangible innovations that made tangible innovations possible. Leapfrog innovations often are intangible ideas that change the world.

1.3.2 Do leapfrog innovations create profitable markets?

The next myth about leapfrog innovations is that they create *profitable markets* and that they make private inventors rich. Here is a counterexample that that changed peoples lives towards more wealth: the invention of the *three-field system* did not create a market or made its inventor rich. The knowledge about three-field system was simply a *public good.* Leapfrog innovations, particularly intangible leapfrog innovations often create public goods and their inventors remain poor. Leapfrog innovations often have no business model and require upfront sponsoring or meritocratic rewards afterwards.

1.3.3 Are leapfrog innovations monolithic?

The last myth about leapfrog innovations is that they are single impressive innovations like a flashing idea of a genius. Here is a counterexample: the iPhone, not technically new, was rather a *design driven, symbolic innovation*, see Steffen (2010). Leapfrog innovations can result from a web of interrelated smaller innovations, where each of the smaller innovations does not look convincing, but in combination they change the rules of the game. As the iPhone did. And as *greeNsort®* can do, if supported.

1.3.4 The lackmus test

In 1998 John Chambers from Bell Labs was awarded the *ACM Software System Award* for *The S system, which has forever altered how people analyze, visualize, and manipulate data*. After the commercial *S+* implementation of *S* we got the free open-source *R* interpreter, we got libraries that allow S-style statistical analysis in *Python*, and we got *Julia*, a language that tries to combine the best of many worlds (*C++*, *R*, *Python*, *Matlab*). Except for SQL, virtually anyone today doing statistical analyses is using one of *R*, *Python*, *Julia*[3]. I consider the S system a leapfrog innovation, although it is *intangible*, largely *free and open-source* and not a single monolithic invention. If you agree on this, you might well consider *greeNsort®* a leapfrog innovation (in line with its logo): *greeNsort®* is intangible, designed for the public good and is a composed web of interrelated innovations, that together enable more efficient, more sustainable algorithms.

[3]Often using *Jupyter notebooks*: Project Jupyter's name is a reference to the three core programming languages supported by Jupyter, which are Julia, Python and R, and also a homage to Galileo's notebooks recording the discovery of the moons of Jupiter

1.4 Innovation-Taxonomy

In order to guide the reader through the *greeNsort*® web of innovations, I use the following symbols to classify the innovations:

Table 1.1: Innovation-classes and -counts

Type	Symbol	Level	Count
paradigmatic change	(*)	implications beyond sorting	21
	D	Definitions	1
	C	Convictions (Core beliefs)	12
patentable methods	B	Bold techniques	20
	A	Algorithms	21
	E	Extended algorithms	9
	F	Further algorithms	8
prior art	T	useful Techniques	12

Each section of this report that describe innovations is preceded by formal entries of those innovations and their dependencies to prior innovation or prior art. For the sake of simplicity - although the classification allows “Extended algorithms” - I have refrained from adding tuned versions of the algorithms to the dependency net. The complete web of innovations can be visualized as a *(D)irected (A)cyclic (G)raph (DAG)*:

The *greeNsort*® innovations will be introduced step-by-step in the following sections.

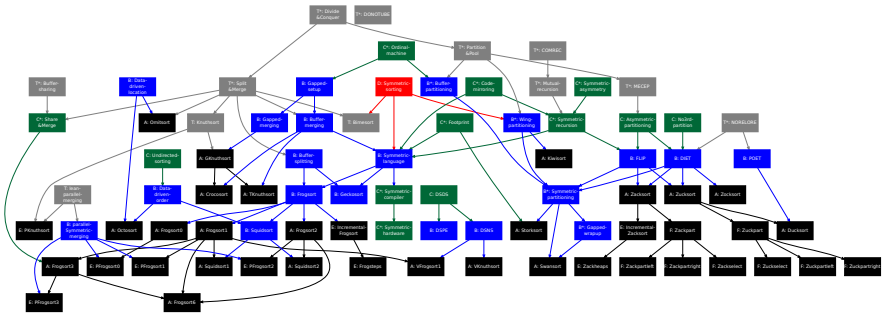

Figure 1.1: Dependencies of innovations (red), core-beliefs (green), methods (blue), algorithms (black), prior art dependencies (grey)

Chapter 2

Paradigmatic innovations

2.1 Ordinal-machine model

```
innovation("Ordinal-machine","C*","minimize monotonic cost of distance")
```

The prior art uses specific machine models, like the *random-access-model* or the *IO-model* with specific assumptions on the costs for accessing and moving data. Those machine models are the basis for analyzing algorithms and proving certain properties. This precise mathematical approach suggests that those machine models represent real machines, which is rarely the case. Furthermore this approach focuses academic efforts on analysis and proof rather than on development of better algorithms. Algorithms and even software have a longer lifetime than particular machines, therefore they should be *robust* against variations in technology and *resilient* in environment with little hardware resources. The algorithm developer cannot know on which machines his code will run, more and more he cannot even know the details of current machines, because they are more and more virtualized in the cloud. Hence *greeNsort*® follows a different approach. A non-specific *Ordinal-machine* model simply assumes that the *cost of access and moves* is a *monotonic function of distance* in a memory address space that reaches from *left* to *right*. For a constant distance cost in the classic random-access-model there is a constant $\mathcal{O}(1)$ cost per move, linear $\mathcal{O}(N)$ cost per level and $\mathcal{O}(N \log N)$ for the total cost of *Divide&Conquer* sorting. Now consider the popular *AlgoRythmics* videos which teach sorting by folk-dances: the movement cost of the dancers on stage is a linear function of distance, hence the cost of the same sorting algorithm is suddenly $\mathcal{O}(N)$ per move, quadratic $\mathcal{O}\left(N^2\right)$ cost per level and $\mathcal{O}\left(N^2 \log N\right)$ for the total cost of *Divide&Conquer* sorting. Today's computers have distance costs in between, say $\mathcal{O}(\log N)$ per move, $\mathcal{O}(N \log N)$ per level and $\mathcal{O}\left(N \log N^2\right)$ for the total cost of *Divide&Conquer* sorting.

From a practical point of view, assuming a *Ordinal-machine* focuses on quick iteration of developing new algorithms, experimenting with them and improving them, rather than proving mathematical properties based on pseudo-exact machine-assumptions. For *greeNsort®* this implies a focus on *distance minimization* and *sustainable measurement*.

2.2 Sustainable measurement

```
innovation("Footprint","C*","measure variableCost x %fixedCost")
```

The prior art empirically evaluates algorithms by measuring *RUNtime*, *CPUtime* or by counting certain operations such as, comparisons, moves, swaps, cache-misses etc. This addresses the CO2 variable cost of running the software but ignores the CO2 fixed cost embodied in producing the hardware[1]. *greeNsort®* introduces the *Footprint*, which multiplies variable cost with relative fixed-cost (per element, not absolute). *greeNsort®* uses:

1. ***(R)andom (A)ccess (M)emory (RAM)*** as a proxy and ***%RAM*** as a normalized proxy for fixed cost hardware requirements,
2. ***Energy*** (RAPL), or simpler ***CPUtime*** or ***RUNtime*** as proxies for variable costs of running the software
3. ***tFootprint***, ***cFootprint*** and ***eFootprint*** as single-scale measures of variable costs penalized for fixed costs (*runTime*, *CPUtime* and *Energy* multiplied with *%RAM*)

Footprints allow to compare algorithms with different memory-speed trade-offs. Optimizing for Footprints avoids the pitfalls of extreme choices: just minimizing *CPUtime* or *Energy* can lead to excessive memory (and hardware) consumption; attempting in-place sorting with zero-buffer-memory has been of great academic interest but never lead to practically convincing algorithms. This new paradigmatic choice of measuring algorithms has enabled *greeNsort®* algorithms with better memory-speed trade-offs aka better trade-offs between execution CO2 of electricity and embodied CO2 of hardware. For details on measuring RAPL-energy and *eFootprints* see the *Methods&Measurement* section.

[1]The *Green Software Foundation* has recently (2021) suggested the Software Carbon Intensity (SCI), which attempts an end-to-end measurement of variable costs and fixed costs on a common CO2-scale. This, however, is not pragmatic: it considers factors that are not attributes of software and that are not available to software developers (energy/instruction of hardware, CO2/energy of location). Furthermore the SCI has the paradoxical feature that in locations with 100% renewable energy the electricity is assumed to have zero costs such that purely the hardware amortization determines the SCI. For more details see my *comment to the SCI* and my *presentation at the GSF*

2.3 Symmetry principle

```
innovation("Symmetric-asymmetry","C*","low-level-asymmetry, high-level-symmetry")
```

2.3.1 Symmetry

In biology, engineering and architecture *symmetry* plays an important role, particularly *bi-symmetry*. But strangely not in Computer Science! Yes, there are publications on *symmetry detection* algorithms, but there symmetry is only the object, not the subject of the design: Symmetry is rarely considered a major design principle for algorithms and code. Although a one-dimensional left-right-memory-address-space should be an invitation for bi-symmetry.

A notable exception is *Quicksort1*, which was designed symmetrically by Hoare (1961b), Hoare (1961a): two pointers move from left and right until they meet an element that is on the wrong side. Unfortunately the search *cannot* be symmetric in a von-Neumann-Machine: first one pointer searches, then the other, this can create heavy imbalance, hence can lead to $\mathcal{O}(N)$ recursion depth and $\mathcal{O}\left(N^2\right)$ execution cost for certain inputs. The popular *Quicksort2* which goes back to Singleton (1969) stops pointers on pivot-ties: this guarantees a balanced partition but at a costly price: pivot-ties are swapped with no benefit and worse the partitioning is not *MECE* and does not early-terminate on ties.

Symmetric pointer search can be approximated by alternating between moving the left and right pointer. The *greeNsort*® algorithm *Chicksort* implements this, and it indeed avoids swapping pivot-ties, but it is *slower* than *Quicksort2* due to the more complex loop structure. I did not explore block-wise alternating pointer search, because there is a much better solution (see the *The Quicksort-Dilemma*).

In biology, engineering and architecture *symmetry* is often composed high-level from low-level asymmetries. *greeNsort*® is built on transferring the *Symmetric-asymmetry* concept to computer science: C-level innovation – embracing low-level-asymmetry within high-level-symmetry, see *Symmetry principle*

2.3.2 Asymmetries

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, see *Asymmetries*
- *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), see *Asymmetries*

as is the topic of sorting

- *Order-asymmetry* the fact that that 'order' is asymmetric and reaches from 'low' to 'high' 'keys', see *Asymmetries*
- *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, see *Asymmetries*
- *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), see *Asymmetries*

How does one embrace low-level-asymmetry within high-level-symmetry? Let's start with a new definition of sorting.

2.4 Definition of sorting

```
innovation("Symmetric-sorting","D","'ascleft', 'ascright', 'descleft', 'descright'")
```

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.

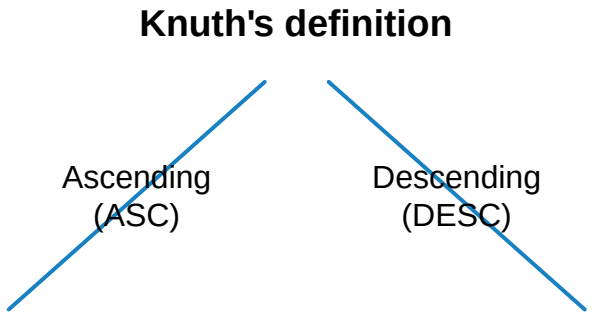

Figure 2.1: Knuth's definition of sorting

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 *Ascending (Asc)* and *Descending (Desc)* correspond to four concrete sorting sequences in memory: *Ascending from Left (AscLeft)*, *Ascending from Right (AscRight)*, and *Descending from Left (DescLeft)*, and *DescLeft*. The Difference between *DescLeft* and *AscLeft*, is a reverted - but stable - sequence of ties!

A first example for the immediate benefits of the *greeNsort®* definition is found in section *Reference algo (Knuthsort)*. The *greeNsort®* definition is powerful because it facilitates reasoning and increases the size of the solution space, like the invention of the *imaginary part* increased the number space from *real numbers* to *complex numbers*, such that suddenly the square-root of a negative number was defined.

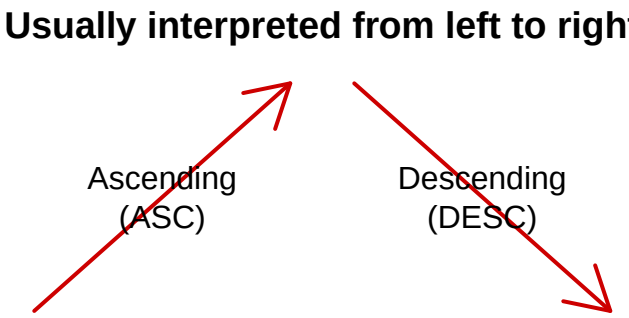

Figure 2.2: Conventional interpretation: from left to right

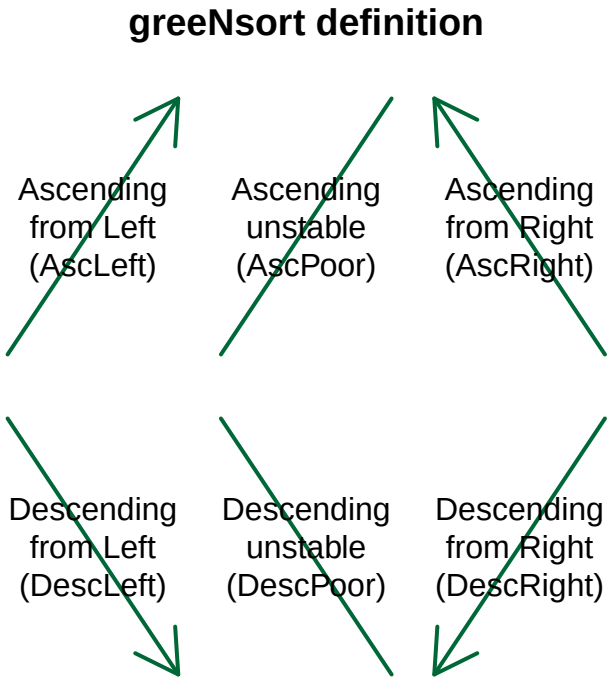

Figure 2.3: greeNsort definition: unstable sorting has two API targets (AscPoor, DescPoor) but stable symmetric sorting has four API targets (AscLeft, AscRight, DescLeft and DescRight)

2.5 Recursion model

```
innovation("Mutual-recursion","T*","is more expressive than self-recursion")
innovation("Symmetric-recursion","C*","symmetric mutual recursion")
dependency("Symmetric-recursion", "Mutual-recursion")
dependency("Symmetric-recursion", "Symmetric-asymmetry")
dependency("Symmetric-recursion", "Code-mirroring")
```

Some people consider *recursion* to be a technical topic, and *tail-recursion* indeed is. However, recursion is first and foremost a mental tool, that helps humans to organize code and understand algorithms. Proof? Because each recursion can be written as loops. QED.

In the context of sorting algorithms prior art recursion usually means *self-recursion*. However, there are good reasons – explained in the book – to assume that *Mutual-recursion* is a more natural and powerful mental problem-solving device than self-recursion. Even without this assumption: mutual recursion is definitely *more expressive* than self-recursion. Mutual-recursion is not new[2], but undervalued. Why? Because mutual-recursion is a natural fit for binary *Divide&Conquer* algorithms.

Divide&Conquer in sorting means dividing and conquering *address ranges*, i.e. ranges from *left* to *right*. *greeNsort®* is built on *Symmetric-recursion*, a new class of mutual-recursions where two functions co-operate recursively in a more or less symmetric fashion, and where the symmetry axis is left-right memory, not ascending-descending order.

And here is the final answer to the question how to embrace low-level-asymmetry within high-level-symmetry: instead of the failing prior art approaches to achieve symmetry in loops, *greeNsort®* embraces asymmetry in loops and creates a higher-level symmetry in mutual-recursion. You will soon see this at work.

[2]In sorting it is rare, I am aware only of Sedgewick's "superoptimization" which combines *nocopy-Mergesort* with a *bitonic merge strategy*, see *First algo (Bimesort)*

2.6 Mirroring code

```
innovation("Code-mirroring","C*","left-right mirroring of code")
innovation("Symmetric-language","C*","symmetric programming language")
dependency("Symmetric-language", "Code-mirroring")
innovation("Symmetric-compiler","C*","compiling symmetric language")
dependency("Symmetric-compiler", "Symmetric-language")
innovation("Symmetric-hardware","C*","instructions for mirroring code")
dependency("Symmetric-hardware", "Symmetric-compiler")
```

The *Wikipedia entry for binary search* contains one of the few references to symmetric behavior: searching for the leftmost or rightmost element in case of ties. However, that the two functions given behave symmetrically is not easy to see, because the code is not symmetric. Achieving symmetry by mirroring the code itself has advantages: it is easier to verify correct and code-mirroring can be done formally and automatically.

As a research project, the *greeNsort*® algorithms have been developed and written largely manually. But the symmetric properties of the two functions calling mutually in *Symmetric-recursion* are an invitation to create one of them by mirroring code sections of the other, either with meta-programming techniques or with new language constructs that support *Code-mirroring*, see the demonstration of the feasibility of Code-mirroring in a (interpreted) R-implementation of *Zacksort* in the *Languages* section. If code-mirroring is supported by languages, compilers and hardware, then the cost of code-duplication in *Symmetric-recursion* can be avoided. The opportunity is: writing elegant parsimonious code once, mirror-it with language constructs and the compiler translates this into hardware-instructions for mirroring sections of instructions. This is a promising new field of research.

2.7 Further techniques

```
innovation("Divide&Conquer","T*","Recursive divide and conquer paradigm")
innovation("COMREC","T*","COMputing on the RECursion")
innovation("NORELORE","T*","NO-REgret LOop-REuse")
innovation("DONOTUBE","T*","DO NOt TUne the BEst-case")
dependency("Mutual-recursion","COMREC")
```

In the broadest sense, mutual recursion is a special case of a broader technique: *T-level technique - (COM)puting on the (REC)ursion (COMREC)*. *COMREC* is a simple powerful technique, not as complicated and expressive as meta-programming. *Meta-programming* tends to be not only expressive but also inefficient, particularly *programming on the language* in interpreted languages such as *R* or *Python*[3]. *COMREC* can be easily compiled to efficient executable code. Beyond *Symmetric-recursion*, further examples of this technique are nested recursive functions, where different functions are used for different problem sizes. The most popular use-case is tuning with *Insertionsort* for problem sizes smaller than a cut-off, the *greeNsort*® algorithms use an `INSERTIONSORT_LIMIT` of 64.

Equal tuning is an example of another *greeNsort*® principle: equal implementation quality for fair comparison. An exception is *Timsort*, which violates the simplicity-principle and hence the test-bed uses an existing implementation, see 2nd Reference (Timsort).

Finally, three principles should be mentioned, that are immediately applied in the section on *Quicksort algorithms*:

- **Robustness**: *greeNsort*® Partition&Pool algorithms are using *random pivots* which guarantees a $\mathcal{O}(N \log N)$ average case for *any* data input. Using deterministic pivots cannot guarantee an $\mathcal{O}(N \log N)$ average case, the analyses of Sedgewick (1977a) which claim this are based on the nonsense-assumption that instead the input data be random, this is irresponsible for production use (unless recursion-depth is guarded and limited as in *Introsort*).
- ***NORELORE***: *greeNsort*® algorithms use optimizations that avoid extra-operations and memory scanning by reusing and melting loops for different purposes.
- ***DONOTUBE***: *greeNsort*® algorithms may invest extra-operations to enable best-case behavior but do not invest extra-operations to speed-up best-cases.

[3]for the inefficiency of Python see *Enhancing the Software Carbon Intensity (SCI) specification of the Green Software Foundation (GSF)*

Chapter 3

Methods&Measurement

3.1 Implementation

The basic philosophy of the *greeNsort®* test-bed is implementing all algorithms with a equally low degree of tuning for fair scientific comparison. Some algorithms have additionally implemented with well-defined tuning to presorted values (postfix 'A') or a simple approach to avoid branch-misprediction (postfix 'B'). The latter works well on partitioning, with few compiler-versions on merging. All algorithms have been thoroughly tested (also for stability) and checked for memory-leaks through valgrind.

Each sorting algorithm has been implemented as a static function in a separate C module that contains *all* required subroutines, this insures fair code locality and reduces linker artifacts. In production code the implied code-duplication for multiple used functions such as *Insertionsort* is usually avoided, here in the test-bed this redundancy gives more fair comparisons.

Older implementations often have two versions of the sorting algorithm (one rather using pointers, one rather using indexed arrays). It turned out that pointer implementations are more practical and tended to be faster in merging, indexed array implementations were more readable and not slower in partitioning, hence newer implementations did not duplicate and compare this anymore. There are usually two context-embeddings of the sorting algorithm:

- a simpler `ex-situ` version which assigns new memory for data and buffer, copies data to new memory, sorts, and copies back (copying back allows quality assurance)
- a more complex `in-situ` version that only allocates new memory for buffer and re-uses original memory for sorting (which can be more complicated)

To get the picture, see the C and R code for `Knuthsort` and `Frogsort1` in the `src` and `R` folder of the `greeNsort` R-package.

3.2 Simulation

All measurements in this document refer to `in-situ` versions. For the *Split&Merge* algorithms with less than 100% buffer, the final `in-situ` merge is necessarily imbalanced and potentially more expensive, hence choosing the `in-situ` setting gives conservative evaluations of some *greeNsort®* algorithms.

For details see the scripts `serial.R`, `parallel.R` and `string.R` in the `inst` folder of the `greeNsort` R-package.

3.3 Basic measurement

For *Energy* measurement the test-bed uses the *(R)unning (A)verage (P)ower (L)imit, Energy measurement of Intel (RAPL)* counters of the linux `powercap` kernel module, see `lib_energy.h`, `lib_energy.c` and `perf.R`. The RAPL counters occasionally overflow, the test-bed detects and corrects overflow (assuming no multiple overflow, which does not happen for the duration of the measurements). RAPL has separate counters for *RAM* energy (`dram`) and *package* energy, which is the sum of *CPU* energy (`core`), *GPU* energy (`uncore`) and a rest that is not counted separately. After overflow-treatment, the test-bed decomposes those counters into `core`, `unco` and rest `base`. All sorting algorithms in the *greeNsort®* test-bed return performance data:

- n: number of elements
- b: average bytes per element
- p: number of active processes (usually 1, 0 for `perfsleep`)
- t: number of active threads per process
- size: *%RAM* relative to data size
- secs: *RUNtime* in seconds from a high-resolution clock[1]
- base: basic Energy
- core: CPU Energy
- unco: GPU Energy[2]
- dram: RAM Energy

[1] see lib_timing.h and lib_timing.c
[2] on my Intel i7-7700 not in RAPL, hence 0

3.4 Calibration

An idle computer still consumes energy, for keeping the RAM alive, keeping the CPU ready, for background services etc. The test-bed provides functions that measure idle energy (`perfsleep`, `calibrate`), estimate idle energy for a given *RUNtime* (see function `calibrate`) and optionally adjust measurements for idle energy. The R function `optperf` returns measurements according to settings in `options("greensort_perf_calc")` by calling one of `rawperf` (raw unchanged measurement), `difperf` (difference to idle measurements) or `adjperf` (keeps total energy constant by reducing all energy measurements but `base` and adding the reduction to `base`). The evaluations in this paper use the `greensort_perf_calc='adj'` setting.

3.5 Evaluation

The test-bed provides a couple of functions extracting well-defined KPIs from the measurements. For sorting algorithms GPU-energy is not interesting, hence `bcdEnergy`, the sum of `base`, `core`, `dram` is a reasonable energy measurement (and is not influenced by `options("greensort_perf_calc")`). The `pcdEnergy` measurement is similar but adds only the thread-fraction of `base` (counting only hardware-threads, not hyper-threads, it is influenced[3] by `options("greensort_perf_calc")`). This paper uses:

- *pcdEnergy*
- *pcdFootprint* (*pcdEnergy* * *%RAM*)
- *RUNtime*
- *tFootprint* (*RUNtime* * *%RAM*)

[3]Note that the reported energy-ratios are more or less the same whatever choice of energy-measure and idle-correction is used

3.6 Benchmark data

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 latter are interesting with regard of the symmetry of adaptivity, but not included in the former `TOTAL` KPI. The decision to include ascending and exclude descending has been made on the rationale that adaptivity to ascending data is more important than to descending because ascending data arises more naturally (e.g. ascending datetime) or can be created at little cost (by reversing). Furthermore, including another three presorted patterns would give too much weight on easy patterns.

3.7 Scripts and results

The measurement are done in three parts *serial*, *string* and *parallel*, most important results are integrated into this document, for more details see:

- `serial.R` with results of serial algorithms on double data in `serial.RData` (n=2^21, 100 replications)
- `string.R` with results of serial algorithms on null-terminated strings in `string.RData` (n=2^17, 100 replications)
- `parallel.R` with results of parallel algorithms on double data in `parallel.RData` (n=2^24, 50 replications for each combination of 1,2,4,8 processes with 1,2,4,8 threads)

Complementing the charts in this paper, we have prepared PDFs (`serial.pdf`, `string.pdf`, `parallel.pdf`) with charts that visualize raw data for exploratory checks such as outliers or suspicious trends (to us the raw data looks good). To deal with few expected outliers, we report the *median* instead of the *mean* of measurements.

3.8 Hardware, OS, compiler

All measurements reported here are done on an Intel[4] i7-7700 CPU under ubuntu.20.04 with the 5.13.0-44-generic kernel and compiling the test-bed with gcc.9.4.0. The CPU is run with hyper-threading switched-off in the BIOS: this goes with a clear expectation that algorithms can successfully satisfy up to 4 cores (more threads are unlikely to speed-up) and provides a clear model for the attribution of the `base`-energy to 4 cores in `pcdEnergy`.

[4]Measurements on AMD® Ryzen 7 pro 6850u were also done giving more or less similar results, however, RAPL measurements on Intel are more precise. The 8-core AMD has hyperthreading switched on, I report some results of simple tests (one measurement of big problem after 1 second CPU sleep), see Update: Powersort and Update: AMD

Chapter 4

Quicksort Algorithms

4.1 The Quicksort-Dilemma

I have described the *greeNsort*® resolution of the *Quicksort-Dilemma* in full length in the introductory chapter of the forthcoming *greeNsort*® book. I repeat this here in the short form typical for this disclosure.

The *Quicksort1* algorithm of Hoare (1961a);Hoare (1961b) tried to be efficient (average $\mathcal{O}(N \log N)$) and early-terminating on ties ($\mathcal{O}(N \log D)$[1]), but on certain data inputs it degenerates to $\mathcal{O}\left(N^2\right)$. Wrong implementations can even lead to endless loops $\mathcal{O}(\infty)$, particularly when sentinels are used.

The popular *Quicksort2* which goes back to Singleton (1969) and has been promoted by Sedgewick (1975);Sedgewick (1977a);Sedgewick (1977b);Sedgewick (1978) stops pointers on pivot-ties: this guarantees a balanced partitioning and hence an average $\mathcal{O}(N \log N)$, however it will not early-terminate on ties.

[1] where D is the number of Distinct keys

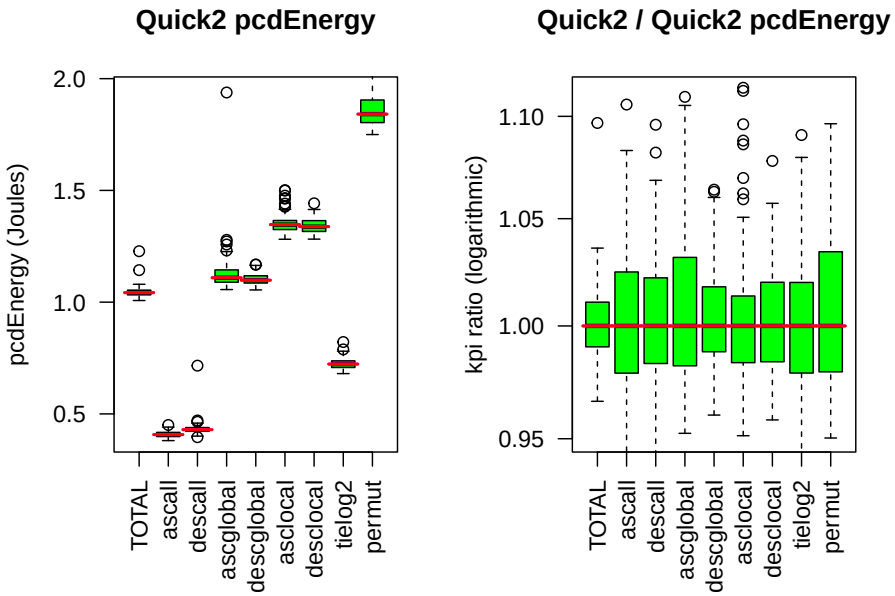

Figure 4.1: Quicksort2 compared to Quicksort2. The left chart compares the absolute energy consumed for 8 input patterns and the TOTAL (of 5). The right chart shows the respective ratios to the median of the reference algorithm (here Quicksort2). The red lines show the reference medians.

The also popular *Quicksort3* developed by Bentley and McIlroy (1993) collects pivot-ties in a third partition between the low and high partition, this gives early termination but at the cost of extra operations and hence lower efficiency.

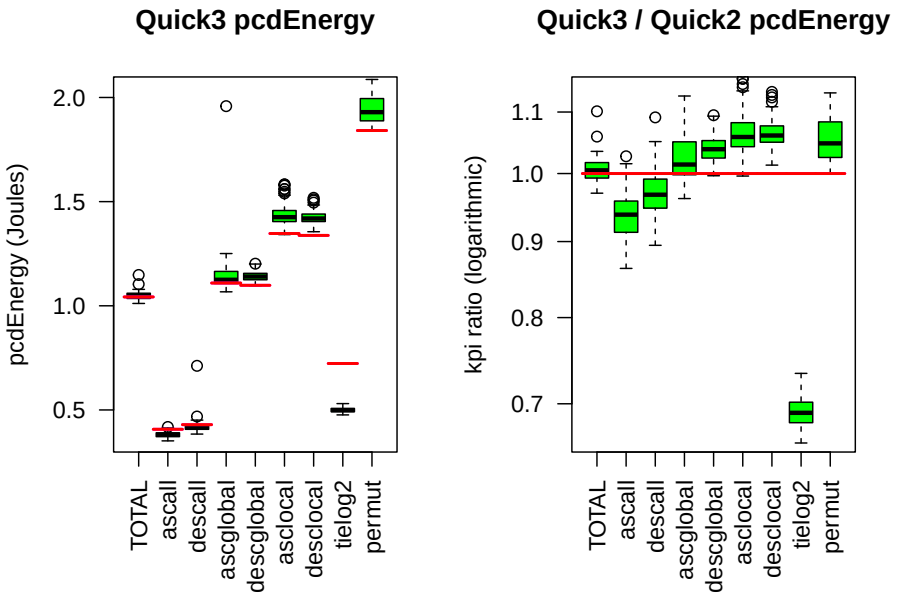

Figure 4.2: Quicksort3 compared to Quicksort2

The right chart shows less *Energy* consumption for the tied data (`tielog2`), but more Energy consumption for the hard sorting tasks such as completely random data (`permut`).

This is the *Quicksort-Dilemma* of the prior art: either you forego efficiency for non-tied data or you forego early termination on ties.

Here are some measurement results of for *Quicksort2*:

Table 4.1: Quicksort2 (medians of absolute measurements)

	%M	rT	pcdE
TOTAL	1	0.0870918	1.0424445
ascall	1	0.0314271	0.4071824
descall	1	0.0330232	0.4294158
ascglobal	1	0.0926678	1.1091954
descglobal	1	0.0929887	1.0982830
asclocal	1	0.1144985	1.3470172
desclocal	1	0.1147733	1.3378996
tielog2	1	0.0588416	0.7229768
permut	1	0.1571141	1.8413511

And here *Quicksort3* compared to *Quicksort2* with results of conservative two-sided Wilcoxon tests:

Table 4.2: Quicksort3 / 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	0.99	1.00	0.01	0.0000	0.0022
ascall	1	0.93	0.94	-0.03	0.0000	0.0000
descall	1	0.97	0.97	-0.01	0.0000	0.0000
ascglobal	1	1.00	1.01	0.02	0.4566	0.0056
descglobal	1	1.01	1.04	0.04	0.0000	0.0000
asclocal	1	1.05	1.06	0.08	0.0000	0.0000
desclocal	1	1.04	1.06	0.08	0.0000	0.0000
tielog2	1	0.65	0.69	-0.22	0.0000	0.0000
permut	1	1.04	1.05	0.09	0.0000	0.0000

Compared to *Quicksort2*, *Quicksort3* needs only 68% Energy on strongly tied data, but without a TOTAL benefit, because it needs 4% more Energy on randomly permuted data.

Recent attempts to solve the *Quicksort-Dilemma* lead to significant improvements but still not to algorithmically convincing solutions:

- Yaroslavskiy (2009) published the dual-pivot Quicksort (*Dupisort*) used such extra-operations to create a real third partition, this 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. *greeNsort®* slightly improved the algorithm by removing a redundant branch (*Tricksort*).
- Edelkamp and Weiß (2016),Edelkamp and Weiss (2016) published the much faster and simpler *Block-Quicksort* (see *Quicksort2B*), but only with a rudimentary and expensive early-termination mechanism[2].
- Peters (2014);Peters (2015);Peters (2021b) combined a branchless algorithm with a proper tuning for ties and a proper tuning for presorted data, the tuning overhead of extra operations in his *Pattern-Defeating Quicksort* (*Pdqsort*) is very little, so it is close to optimal, see the *excellent C++ implementation on github*. However, instead of being algorithmically clean, Pdqsort uses some *heuristic shuffling*. Furthermore, the fallback to *Heapsort* can slow down performance (see Peters (2021a)), and once combined with an expensive comparison function such as localized string comparison `strcoll`, it becomes slower than *Quicksort2*.

[2]Fun fact: 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

4.2 *greeNsort®* solution

The mentioned approaches of the prior art share that they attempt to achieve symmetry on the loop-level, which contradicts the *greeNsort® Symmetric-asymmetry* principle. Asymmetric-partitioning on the loop level has been banned by Sedgewick (1977b):

> *Program 3, due to its asymmetrical nature, can produce unbalanced partitions [...] unary files represent the worst case for Program 3 [...] Program 3 is also quadratic for binary files [...] The evidence in favor of stopping the scanning pointers on keys equal to the partitioning element in Quicksort is overwhelming* — Sedgewick (1977)

This was the point in history where *Quicksort2* became standard, until *Quicksort3* was engineered and declared "optimal" by Sedgewick and Bentley (2002). It is time to revise these decisions now.

4.2.1 DIET method

```
innovation("MECEP","T*","MECEP partitioning")
dependency("MECEP","Partition&Pool")
innovation("No3rd-partition","C","Early Termination without 3rd partition")
innovation("Asymmetric-partitioning","C","Asymmetric-partitioning is good")
dependency("Asymmetric-partitioning","MECEP")
innovation("DIET","B","Distinct Identification for Early Termination")
dependency("DIET","No3rd-partition")
dependency("DIET","Asymmetric-partitioning")
dependency("DIET","NORELORE")
```

First I drop the prior art assumption, that early-termination on ties requires collecting pivot-ties in a third (middle) partition. Early-termination on ties not even needs two partitions. An algorithm can early-terminate in a single partition. Once the algorithm knows that one partition is purely composed of pivot-ties (Sedgewick's 'unary files') it can stop further partitioning and recursion. QED. So far this early-termination works only for a special simple case, but simplicity wins. How can this generalized to early terminate for multiple distinct tied values? By using *T-level technique - (M)utually (E)xclusive and (C)ollectively (E)xaustive (P))artitioning (MECEP)*. QED. As an example consider a file with four distinct values: a binary *MECEP* can result in two files with two distinct values each, after another recursion it is possible to have four unary files and early-terminate. How does one do MECE binary partitioning? By revising Sedgewick's ban on *Asymmetric-partitioning*: all pivot-ties go to one side. How can unary files be identified efficiently? By applying the *T-level technique – (NO)-(RE)gret (LO)op-(RE)use (NORELORE)* principle. In Asymmetric-partitioning, one pointer search contains equality (`LE` or `GE`) the other not (`LT` or `GT`), if the first pointer search runs `while(EQ(x[i],pivot)` and continues with `while(LE(x[i],pivot)` or `while(GE(x[i],pivot)`, then it is possible to identify unary files at almost no extra cost: the algorithm simply starts with a no-regret pre-loop `while(EQ(x[i],pivot)`. If this pre-loop reaches the other side the algorithm knows to have a unary file and can early terminate. If this pre-loop finds a key that is different from the pivot, then no work has been wasted because the same decision was made as if the first pointer search of the main loop had been used. That is, the result of this pre-loop can be re-used. Only for the last key different from the pivot the query needs to find out whether this element belongs to the same or the other partition. This implies that instead of $N-1$ comparisons for binary partitioning only *one* single extra comparison is needed to identify unary files. I call this the *B-level innovation – (D)istinct (I)dentification for (E)arly (T)ermination (DIET)* method. Since with $N-1$ comparisons it is not possible to binary partition *and* identify unary input, with only one more comparison DIET is optimal. QED.

4.2.2 Engineering Zocksort

```
innovation("Zocksort","A", "self-recursive DIET Quicksort")
dependency("Zocksort", "DIET")
```

Coding the *DIET*-method into a prior art self-recursion function gives an optimal early terminating Quicksort (named *Zocksort*) with an Achilles heel. For example, if the *Asymmetric-partitioning* is defined as

left partition	right partition
`LE(x[i], pivot)`	`GT(x[i], pivot)`
`111011111111111`	"

then having input data composed of one 0 and many 1's is likely to sample 1 as a pivot and partition *all* values into the left partition. Only if 0 is sampled as the pivot, 0 and 1's are separated into two partitions. This implies an expected $\mathcal{O}(N^2)$ worst case. The risky word 'Zock' comes from German 'zocken' which means 'gambling' and is a onomatopoeia resembling smashing cards on the table.

4.2.3 FLIP method

```
innovation("FLIP","B","Fast Loops In Partitioning")
dependency("FLIP","Asymmetric-partitioning")
dependency("FLIP","Symmetric-recursion")
```

Because of this asymmetric risk Sedgewick (1977b) condemned *Asymmetric-partitioning*. Instead of fighting asymmetric loops, *greeNsort®* embraces them and delegates symmetry-creation to *T-level technique - (COM)puting on the (REC)ursion (COMREC)*. Instead of self-recursion, which perpetuates asymmetry, *greeNsort®* uses *Symmetric-recursion*. Note that any input data that can fool the above Asymmetric-partitioning, cannot fool the left-right-mirrored Asymmetric-partitioning:

left partition	right partition
`LT(x[i], pivot)`	`GE(x[i], pivot)`
`0`	`11111111111111`

Mutually recursing between those two mirrored Asymmetric-partitioning functions prevents being fooled by *any* asymmetric input data. QED. I name this the *B-level innovation – (F)ast (L)oops (I)n (P)artitioning (FLIP)* method. FLIP is an acronym and a mnemonic for mirroring.

4.2.4 Engineering Zacksort

```
innovation("Zacksort","A", "zig-zagging DIET-FLIP Quicksort")
dependency("Zacksort", "DIET")
dependency("Zacksort", "FLIP")
```

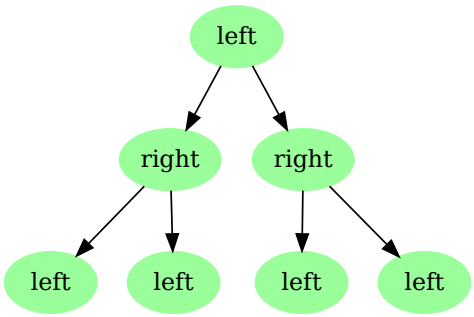

Figure 4.3: Zacksort zig-zagging symmetric recursion

One way to combine the *DIET* with the *FLIP* method is zig-zagging between putting pivot-ties into the left partition on one recursion-level and into the right partition on the next recursion-level. The mutual-recursion is: the left function calls the right function in left and right branches, the right function calls the left function in the left and right branch. The German translation of zig-zag is 'Zick-Zack' and 'Zack' also means 'quick', hence the name *Zacksort*.

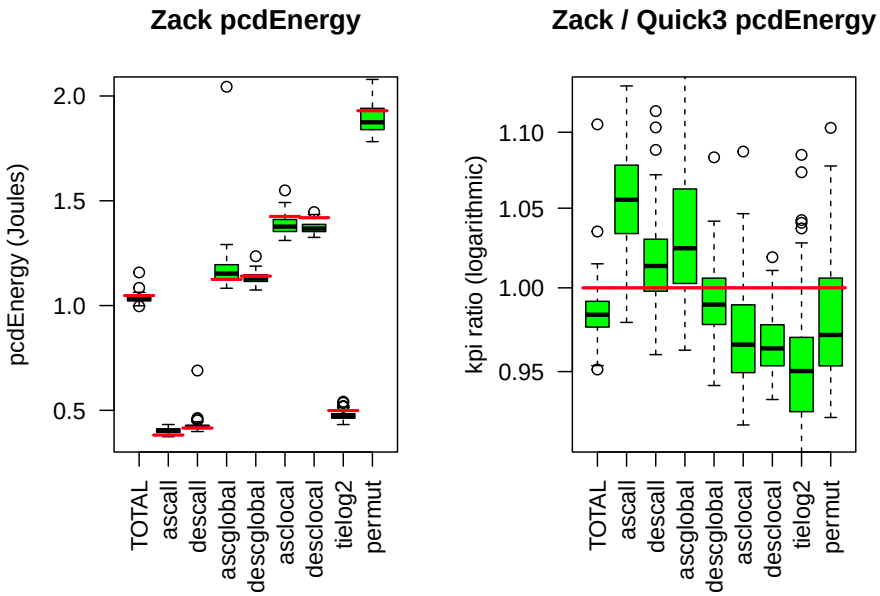

Figure 4.4: Zacksort compared to Quicksort3

Table 4.5: Zacksort / 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	0.99	0.99	-0.01	0e+00	0.0000
ascall	1	0.99	0.99	0.00	3e-04	0.0034
descall	1	0.97	0.98	-0.01	0e+00	0.0000
ascglobal	1	1.03	1.04	0.04	0e+00	0.0000
descglobal	1	1.03	1.03	0.03	0e+00	0.0000
asclocal	1	1.03	1.02	0.03	0e+00	0.0000
desclocal	1	1.02	1.02	0.03	0e+00	0.0000
tielog2	1	0.66	0.66	-0.25	0e+00	0.0000
permut	1	1.03	1.02	0.03	0e+00	0.0002

Compared to *Quicksort2*, *Zacksort* needs only 99% Energy, due to 66% Energy on strongly tied data. Remember, that Quicksort3 needed in TOTAL more Energy than Quicksort2.

4.2.5 Engineering Zucksort

```
innovation("Zucksort","A", "semi-flipping DIET-FLIP Quicksort")
dependency("Zucksort", "DIET")
dependency("Zucksort", "FLIP")
```

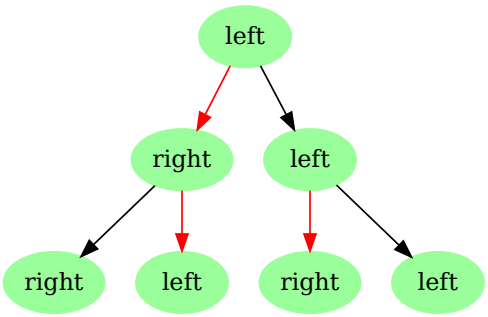

Figure 4.5: Zucksort semi-flipping symmetric recursion, red branches can have pivot-ties

Another way to combine DIET and FLIP is flipping the pivot-asymmetry only in that branch, that can contain pivot-ties, the other branch is not at risk for quadratic execution costs. Assume the left function puts pivot-ties left and the right function puts pivot-ties right, then the mutual-recursion is: the left function flips to calling the right function on the left branch and keeps calling the left function on the right branch; while the right function keeps calling the right function on the left branch and flips to calling the left function on the right branch.

4.2.6 POET method

```
innovation("POET","B", "Pre-Order Early Termination")
dependency("POET", "NORELORE")
```

So far *Zucksort* early terminates only on ties, not on presorted data. Testing in a loop for presorted data before starting to sort is not new, and is an extra-investment of comparisons which may or may not pay-off. I now present the *POET* method, which like the *DIET*-method employs the *NORELORE* principle as much as possible. The initial idea of POET was to put a second pre-loop between the DIET-loop and the MAIN-loop such that - without regret - each loop spills into the next and the next leverages the work done so far by the previous. Let's see: Once the DIET-loop terminates without early-termination, the last inspected element is *distinct* from the pivot, but still can be *presorted* relative to the previous element (or be the first element). I.E. it is possible to continue with a POET-loop with only one comparison overlap. The POET-loop either reaches the other side, confirming completely presorted data and results in early-termination, or it breaks and needs to be followed by the MAIN loop. So far the theory. In practice, there are a couple of difficulties:

- the initial SWAP for placing the pivot at the beginning of the data is incompatible with presorted data, hence must not be done
- without the SWAP the DIET loop needs to compare one more element
- reading the pivot before and doing this SWAP after the POET loop costs two large-distance operations
- not knowing the position of the pivot after partitioning does not allow to exclude it between the left and right partition, which (theoretically) gives a $\mathcal{O}(\infty)$ instead of a $\mathcal{O}(N^2)$ worst case

- reusing the progress of a break-ed POET loop is not trivial, theoretically one could identify the first element of the presorted sequence that belongs to the other partition by binary search and then enter a *(B)inary (I)dentified (A)symmetric (S)earch (BIAS)*-loop. A BIAS-loop could simply increment the pointer along the presorted sequence and search only on the other side for elements that need swapping, this is theoretically cheaper than the final MAIN-loop, which needs to compare on both sides.

In summary: the idea to put a POET-loop between the DIET-loop and the MAIN-loop is still to complicated, violates the *DONOTUBE* principle and does not pay-off. The POET method is actually much simpler: instead the POET-loop simply replaces the DIET-loop. This is possible, because an all-tied sequence formally is presorted, hence a POET-loop having affirmed complete presortedness enables early-termination for all-tied data. Different from the DIET-loop, the progress of the POET-loop cannot be reused by the MAIN-loop, hence, misinvested comparisons are lost and the MAIN-loop needs to start from scratch. This is less dramatic than it sounds. Yes, the DIET-method was provably optimal with at most one misinvested comparison per partitioning

(deterministically) and the POET-method does not have this property. But under random input the expected number of misinvested comparisons is smaller-equal one. And any non-random non-presorted data that lures the algorithm into big progress of the POET-loop must be highly structured, and hence is likely to be early terminated in its branches.

4.2.7 Engineering Ducksort

```
innovation("Ducksort","A", "semi-flipping POET-FLIP Zucksort")
dependency("Ducksort", "Zucksort")
dependency("Ducksort", "POET")
```

The resulting algorithm of replacing the *DIET*-method with the *B-level innovation – (P)re-(O)rder (E)arly (T)ermination (POET)*-method in *Zucksort* is called *Ducksort* and the differences are:

Zucksort	Ducksort
1. delegate to *Insertionsort* if small	1. delegate to Insertionsort if small
2. sample pivot and SWAP as sentinel	2. POET-loop potentially early terminates
3. DIET-loop potentially early terminates	3. sample pivot and SWAP as sentinel
4. otherwise MAIN-loop	4. otherwise MAIN-loop
5. SWAP pivot into position	5. SWAP pivot into position
6. recall left and right excluding pivot	6. recall left and right excluding pivot

The simplicity of Ducksort is almost poetically beautiful! A few lines of code and mirroring them is clearly superior to the famous *Quicksort3*: Ducksort early-terminates on more best-cases and still is faster on random input. Furthermore it can be block-tuned (*DucksortB*) which is difficult with Quicksort3. Note finally that Ducksort can further be tuned to also early-terminate on slightly disturbed presorted data, using the *limited Insertionsort* from *Pdqsort* (see Peters (2014);Peters (2015);Peters (2021b)). I have tried this, but did not find such ordinary tuning-trade-offs convincing (*NORELORE*).

Despite all the enthusiasm about the elegance of *Ducksort*, it should not be concealed that the practical benefit of the additional adaptivity is limited: theoretically, every branch in the recursion can recognize presorted data, but in practice this hardly ever happens: when the recursion has zoomed into a presorted area with partially presorted data, the formerly presorted data has long since been scrambled by the previous partitioning SWAPs. Truly effective processing of presorted data is the domain of Split&Merge Algorithms. An alternative to achieve $\mathcal{O}(N)$ for completely presorted data is to simply use a single *POET*-loop before recursion begins (named *ZucksortA* in the test-bed).

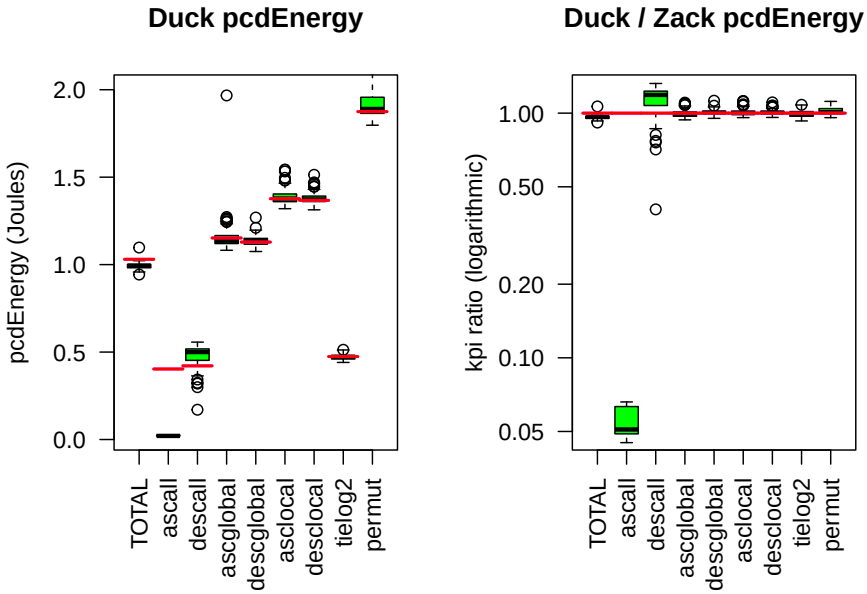

Figure 4.6: Ducksort compared to Zacksort

Table 4.7: ducksort / Zacksort (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	0.96	-0.04	0.0000	0.0000
ascall	1	0.05	0.05	-0.38	0.0000	0.0000
descall	1	1.20	1.19	0.08	0.0000	0.0000
ascglobal	1	0.99	0.99	-0.02	0.0000	0.0891
descglobal	1	0.99	1.00	0.01	0.0000	0.2645
asclocal	1	0.99	1.00	0.00	0.0000	0.8298
desclocal	1	0.99	1.00	0.01	0.0003	0.0865
tielog2	1	0.99	0.99	0.00	0.1482	0.2039
permut	1	1.00	1.01	0.02	0.6096	0.1602

Compared to *Zacksort*, *Ducksort* needs only 96% Energy due to only 5% Energy on presorted data.

4.2.8 Partial Z:cksort

```
innovation("Zackpart","F", "MECEP partial sorting between l and r")
innovation("Zuckpart","F", "MECEP partial sorting between l and r")
dependency("Zackpart", "Zacksort")
dependency("Zuckpart", "Zucksort")
innovation("Zackselect","F", "MECEP more informative than Quickselect")
innovation("Zuckselect","F", "MECEP more informative than Quickselect")
dependency("Zackselect", "Zackpart")
dependency("Zuckselect", "Zuckpart")
innovation("Zackpartleft","F", "MECEP partial sorting left of r")
innovation("Zuckpartleft","F", "MECEP partial sorting left of r")
dependency("Zackpartleft", "Zackpart")
dependency("Zuckpartleft", "Zuckpart")
innovation("Zackpartright","F", "MECEP partial sorting right of l")
innovation("Zuckpartright","F", "MECEP partial sorting right of l")
dependency("Zackpartright", "Zackpart")
dependency("Zuckpartright", "Zuckpart")
```

From *Quicksort1* Hoare (1961a);Hoare (1961c) derived the 'FIND' algorithm, later known as *Quickselect*. Quickselect is a special case of partial sorting (*Quickpart*). Let X be an unsorted set of N elements and Y the sorted version of this set. Quickpart (usually following the logic of *Quicksort2*) takes as input X and two positions l, r where $1 \leq l, r \leq N$ and returns a partially sorted set Z such that

$$\begin{array}{rcl} Z[i < l] & \leq & Y[l] \\ Z[l..r] & = & Y[l..r] \\ Z[i > r] & \geq & Y[r] \end{array}$$

Note the use of $\leq$ and $\geq$ because ties can go to both partitions. Let 'Z:ck' denote both, 'Zack' and 'Zuck', then designing *Z:ckpart* following the logic of *Z:cksort* gives similar algorithms but due to *MECEP* with stricter guarantees:

$$\begin{array}{rcl} Z[i < l] & < & Y[l] \\ Z[l..r] & = & Y[l..r] \\ Z[i > r] & > & Y[r] \end{array}$$

Different constraints on the parameters l and r give special case-algorithms:

prior art	greeNsort	constraints on l,r
Quickpart	Z:ckpart	(none)
Quickpartleft	Z:ckpartleft	l=1
Quickpartright	Z:ckpartright	r=N
Quicksort	Z:cksort	l=1, r=N
Quickselect	Z:ckselect	l=r

The difference between the prior art and the *greeNsort®* algorithms is best demonstrated by comparing *Quickselect* and *Z:ckselect*. The former ignores ties and the only meaningful return-value is the key-value at position $l = r$. The latter returns the positions of the leftmost and rightmost ties to that value, hence is more informative: we learn whether the data is tied at this position and how much it is tied. In Quickselect, ties to $Y[l = r]$ can be anywhere, Z:ckselect guarantees that all ties are in the position where they would be in the fully sorted set Y, hence Z:ckselect does somewhat more sorting work:

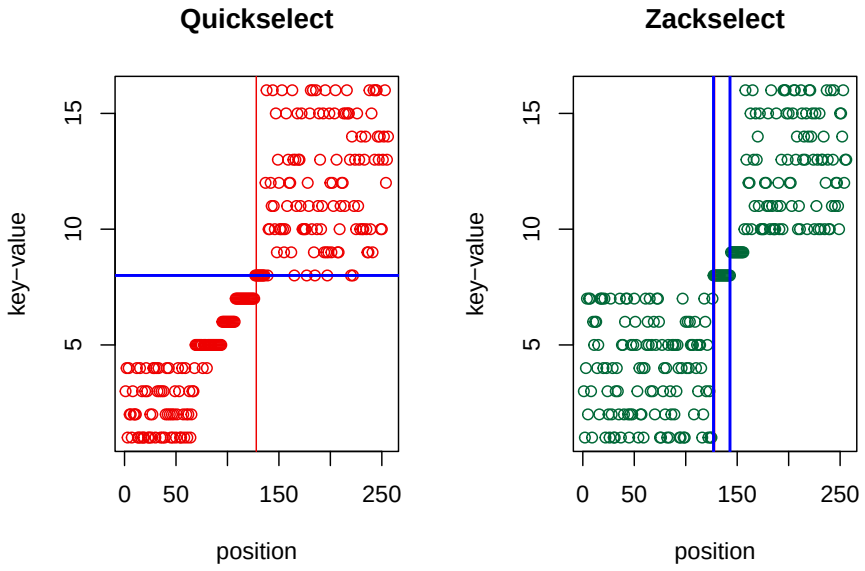

Figure 4.7: Quickselect versus Zackselect, return value(s) in blue

The cost of both is comparable and both have expected $\mathcal{O}(N)$.

4.2.9 Z:cksort implementation

When using random pivots, *Z:cksort* has expected $\mathcal{O}(N \log D)$ cost for D distinct keys due to ties. This probabilistic guarantee of $\mathcal{O}(N \log N)$ or better is lost when using deterministic pivots such as median-of-three. Replacing a random pivot by an assumption about random data (Sedgewick:1977a) is academic and dangerous:

> *The first principle was security [...] logically impossible for any source language program to cause the computer to run wild, either at compile time or at run time. [...] I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.* — Tony Hoare (The 1980 ACM Turing Award Lecture)

Yes, using random number generators in *Quicksorts* and *Z:cksorts* has a cost which can be saved, but a proper deterministic implementation must control recursion depth and fall back to an algorithm *with* $\mathcal{O}(N \log N)$ guarantees. Note that falling back to *Heapsort* as in *Introsort* comes with a performance hit[3]. When implementing Z:cksort with deterministic pivots, the *same* pivots should be selected in the left and right version of the algorithm (hence not perfectly symmetric): this guarantees that a misbehaved pivot will be rendered harmless at the next recursion level. Z:cksort is quite well-behaved when taking a pivot from the middle, however it remains attackable, hence serious code using deterministic pivots must implement a fallback. Note that the most efficient fallback is Z:cksort with random pivots: the expected execution cost is always $\mathcal{O}(N \log N)$ and *RUNtime* about factor 3 better than Heapsort, hence it is never rational to switch to Heapsort (unless probabilistic behavior is fundamentally unacceptable in the given context).

Like *Quicksort2*, *Z:cksort* can be implemented with sentinels[4] and block-tuned (see *ZacksortB*, *ZucksortB* and *DucksortB*) as predicted by Hoare (1962) and rediscovered by Edelkamp and Weiß (2016),Edelkamp and Weiss (2016).

Quicksort2 can be easily executed in parallel branches (*PQuicksort2*, *PQuicksort2B*), however, parallel partitioning is complicated. The same is true for *Z:cksort* (*PDucksort*, *PDucksortB*).

[3] about factor 3 slower, or worse in case of ties

[4] the *greeNsort®* implementations use the pivot as sentinel one one side, both-sided sentinels are also possible albeit with less elegant code

4.3 Quicksort conclusion

I have shown how the *Quicksort-Dilemma* can be resolved and how exactly that symmetric probabilistic algorithm can be written which Tony Hoare tried to design with *Quicksort1*. Tony Hoare was *almost* successful, but the challenges of the 'almost' persisted for 60 years. I have shown that the prior art took a wrong turn in the seventies where it condemned Asymmetric-partitioning loops, which then was followed by many subsequent failures. *Zacksort* and *Zucksort* are elegant solutions that combine efficiency with early termination on tied data, and *Ducksort* additionally achieves early termination on presorted data without increasing code-complexity. This use of symmetry seamlessly leads to partial sorting algorithms that consistently handle ties and return more useful information.

Figure 4.8: 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 4.9: 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 4.10: DucksortB / Quicksort2B (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	0.87	0.87	-0.13	0e+00	0.0000
ascall	1	0.03	0.04	-0.59	0e+00	0.0000
descall	1	1.05	1.06	0.04	0e+00	0.0000
ascglobal	1	0.98	0.99	-0.01	0e+00	0.0118
descglobal	1	0.98	0.99	-0.01	0e+00	0.0004
asclocal	1	0.99	1.00	0.00	0e+00	0.4608
desclocal	1	0.99	0.99	-0.01	0e+00	0.0019
tielog2	1	0.36	0.39	-0.41	0e+00	0.0000
permut	1	0.99	1.01	0.01	4e-04	0.0570

Ducksort saves 5% Energy compared to *Quicksort2*, comparing the block-tuned versions the savings are even 13% (due to saving 64% on strongly tied data).

To illustrate the height of the *greeNsort*® inventions after this entrenched tradition of mistaken historical development, here is the Innovation-lineage of *Ducksort*: all innovations on which *Ducksort* depends form a *(D)irected (A)cyclic (G)raph (DAG)*:

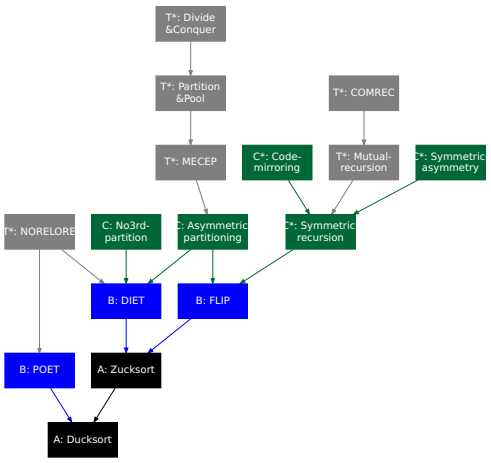

Figure 4.9: Innovation-lineage of Ducksort

Chapter 5

Split&Merge Algorithms

Quicksorts (and *Z:cksorts*) have the following limitations

- they are not stable[1]
- they have $\mathcal{O}(N^2)$ worst case
- their SWAPing limits to equally-sized elements
- their partitioning phase is difficult to parallelize

A *generic sorting algorithm* must deliver:

- stable positions with regard to ties
- a $\mathcal{O}(N \log N)$ worst case (deterministic)
- can be implemented with variable-sized elements
- can be implemented parallel in both, divide and conquer phases

Divide&Conquer generic-sorting can be done in the *Split&Merge* or in the *Partition&Pool* paradigm, I focus on Split&Merge algorithms here (generic Partition&Pool algorithms are considered in section *Partition&Pool Algorithms*).

```
innovation("Split&Merge","T*","Recursive split and merge paradigm")
dependency("Split&Merge","Divide&Conquer")
```

[1] stability can be achieved by a workaround: using the positions as extra key, but that costs extra memory and more comparisons

5.1 The Mergesort-Dilemma

Regarding binary *Mergesort* the prior art has basically four camps:

- balanced-Mergesort (using 100% buffer)
- natural-Mergesort (using not more than 50% buffer)
- Mergesort with block-managed memory (parsimoniously using $\mathcal{O}\left(\sqrt{(}N)\right)$ buffer)
- in-place-Mergesort (insisting on $\mathcal{O}(1)$ or not more than $\mathcal{O}(\log(N))$ buffer)

There is an inherent space-time trade-off: the less buffer required, the less speed achieved. There was lots of academic interest to find the perfect in-place-Mergesort (termed the "holy sorting grail" by Robert Sedgewick), however, in practice, in-place-Mergesorts were rarely used, because they were just too slow. Mergesort with block-managed memory requires a certain implementation complexity, hence is also not popular. For more information see the *Low-Memory* section. This leaves us with balanced-Mergesort versus natural-Mergesort.

In practice one finds many variations – often inefficient – of Mergesort. I consider the following two algorithms as prototypical for the most efficient algorithms of two camps:

- balanced-Mergesort: *Knuthsort* (see the next two sections)
- natural-Mergesort: *Timsort* by Peters (2002)

5.2 First algo (Bimesort)

```
innovation("Bimesort","T","Nocopy mergesort with bitonic merge")
dependency("Bimesort","Split&Merge")
dependency("Bimesort", "Symmetric-sorting")
```

We need to combine certain teaching of the prior art to obtain a fair reference algorithm that represents the knowledge of the prior art, which requires a bit of your attention and patience. The first step towards the reference algorithm is the definition of *Bimesort*.

In chapter 12 (Mergesort) of Sedgewick (1990) taught on page 164 a *Merge* algorithm that uses two sentinels. This has the advantage that the inner loop does not need any explicit loop checks (stop conditions), but has the disadvantage that a suitable sentinel value needs to exist, needs to be known and allocated memory needs to reserve extra space for two sentinels, which is not practical in many settings. Hence, Sedgewick (1990) on page 166 taught a *Mergesort* that uses a *Bitonic Merge* which does not need sentinels, but still needs only one loop check The book does not mention, that the sort is not stable as a consequence from the fact that the merge is not stable. In a later version, Sedgewick (1998) still teaches the unstable *Program 8.2 Abstract in-place-merge*, but added an exercise that mentions missing stability and asks students to fix the merge. So far all the mentioned code needed two moves per element and recursion level for copying data from memory A to buffer B and merging back to A. Sedgewick (1998) now teaches *Program 8.4 Mergesort with no copying*, an algorithm that alternates between merging from memory A to memory B one recursion level and merging from B to A on the next level, and hence saves the unnecessary copying. This is however not compatible with calling *Program 8.2 Abstract in-place-merge* with its two moves per element.

Sedgewick (page 346) mentions as a *"superoptimization"* the possibility to combine *nocopy-Mergesort* with a *bitonic merge strategy*: *"We implement routines for both merge and mergesort, one each for putting arrays in increasing order and in decreasing order"*. If one actually does this, even using truly stable merges for both orders, then the result is - again - an unstable algorithm, unless we replace the word "decreasing" in the above sentence with *Ascending from Right (AscRight)*. I don't know whether Sedgewick's superoptimization was stable or not (the code is not in the book), but the key message here is: what Sedgewick calls a "mindbending recursive argument switchery ... only recommended to experts", turns into a precise instruction as soon as we use the the terms *Ascending from Left (AscLeft)* and *AscRight*. Within the *greeNsort®* definition of *Symmetric-sorting* stability is easy, the resulting algorithm I call *Bimesort*. We show it here for its elegance, in a simplified version without register caching and without insertion-sort tuning. For this paper *Split&Merge* algorithms are implemented in C with pointers, the implementation assumes the data in A, an equally large buffer B and the data needs to be copied from A to B before sorting, although this is not needed for certain recursion depths. Note that all the complexity of duplicating the merge- and recursive sort-function comes with a severe disadvantage: the algorithm *must* move the data on each recursion level, even if the data is perfectly presorted.

```
void Bmerge_AscLeft(ValueT *z, ValueT *l, ValueT *m, ValueT *r){
  while(l<=m){
    if (LT(*r,*l)){
      *(z++) = *(r--);
    }else{
      *(z++) = *(l++);
    }
  }
  while(m<r){
    *(z++) = *(r--);
  }
}
void Bmerge_AscRight(ValueT *z, ValueT *l, ValueT *m, ValueT *r){
  while(m<r){
    if (LT(*r,*l)){
      *(z++) = *(l++);
    }else{
      *(z++) = *(r--);
    }
  }
  while(l<=m){
    *(z++) = *(l++);
  }
}
```

```
void Bsort_AscLeft(ValueT *a, ValueT *b, IndexT n){
    IndexT m;
  if (n>1){
    m = n/2;
    Bsort_AscLeft(b, a, m);
    Bsort_AscRight(b+m, a+m, n-m);
    Bmerge_AscLeft(a, b, b+m-1, b+n-1);
  }
}
void Bsort_AscRight(ValueT *a, ValueT *b, IndexT n){
  IndexT m;
  if (n>1){
    m = n/2;
    Bsort_AscRight(b, a, m);
    Bsort_AscLeft(b+m, a+m, n-m);
    Bmerge_AscRight(a, b, b+m-1, b+n-1);
  }
}
```

5.3 Reference algo (Knuthsort)

```
innovation("Knuthsort","T","Nocopy mergesort with Knuth's merge")
dependency("Knuthsort","Split&Merge")
```

All of Sedgwicks's effort about his bitonic merge-strategy ignored an earlier teaching of Knuth (1973), that you can have one loop check much easier: only that input sequence from which the last element has been taken for the merge can exhaust, hence only that one needs to be checked! Furthermore Katajainen and Träff (1997) has taught how to get away with even a half expected loop check. Katajainen's clever merge qualifies as 'tuning' because it costs an extra key-comparison at the beginning for identifying which of the two input sequences will exhaust first, only this one needs to be checked, about 50% in a balanced merge. But Knuth's clever merge can be seen as a basic algorithmic technique without such a trade-off. It can easily be combined with Sedgwicks's *Nocopy-Mergesort*. To honor Programming-Artist Donald Knuth, I call this *Knuthsort*. It is much simpler than *Bimesort* and serves as the main prior art reference for the *greeNsort*® algorithms. Here is the code in a simplified version without register caching and without insertion-sort tuning:

```
void Kmerge(ValueT *z, ValueT *l, ValueT *m, ValueT *r){
  ValueT *j=m+1;
  for (;;){
    if (LT(*j, *l)){
      *(z++) = *(j++);
      if (j > r)  // one check either here
        break;
    }else{
      *(z++) = *(l++);
      if (l > m)  // or one   check here
        break;
    }
  }
  while(l <= m)
    *(z++) = *(l++);
  while(j <= r)
    *(z++) = *(j++);
}
```

```
void Ksort(ValueT *a, ValueT *b, IndexT n){
  IndexT m;
  if (n>1){
    m = n/2;
    Ksort(b, a, m);
    Ksort(b+m, a+m, n-m);
    Kmerge(a, b, b+m-1, b+n-1);
  }
}
```

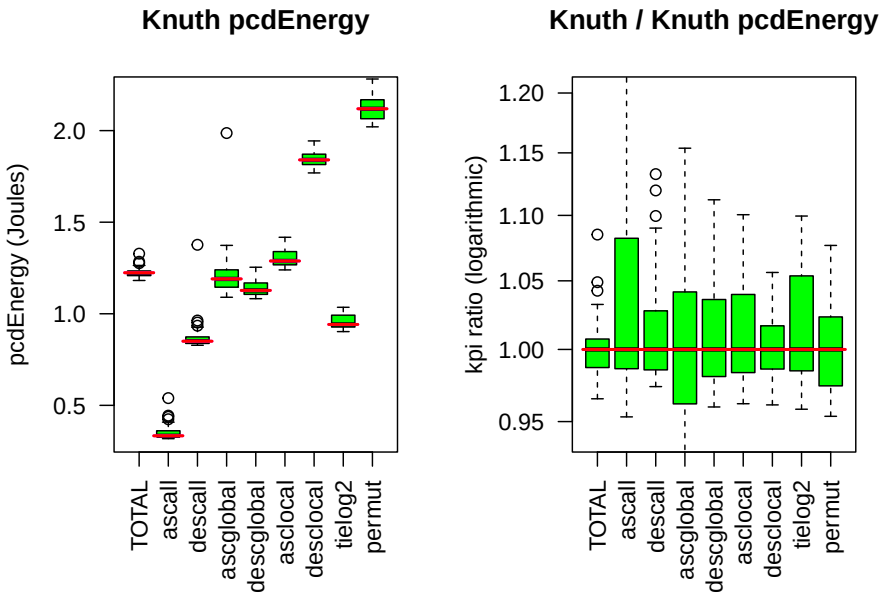

Figure 5.1: Knuthsort compared to Knuthsort. The left chart compares the absolute energy consumed for 8 input patterns and the TOTAL (of 5). The right chart shows the respective ratios to the median of the reference algorithm (here Knuthsort). The red lines show the reference medians.

Table 5.1: Knuthsort (medians of absolute measurements)

	%M	rT	pcdE	pcdF
TOTAL	2	0.0937272	1.2240148	2.4480297
ascall	2	0.0224867	0.3337648	0.6675297
descall	2	0.0635693	0.8502343	1.7004686
ascglobal	2	0.0896020	1.1903291	2.3806582
descglobal	2	0.0877766	1.1275486	2.2550973
asclocal	2	0.0976911	1.2881909	2.5763818
desclocal	2	0.1405737	1.8402222	3.6804445
tielog2	2	0.0712948	0.9417672	1.8835344
permut	2	0.1640641	2.1191219	4.2382439

5.4 2nd Reference (Timsort)

Peters (2002) *Timsort* is famous for its $\mathcal{O}(N)$ best-case for presorted data, but it comes with the following limitations:

- it uses an inefficient merge which needs two moves per element (compared to one move per element in *Knuthsort*)
- it is inherently serial
- it invests extra-operations ("galloping-mode") to speed-up best-cases (which violates the *T-level technique – (DO) (NO)t (TU)ne the (BE)st case (DONOTUBE)* principle)

Since the the python implementation is for lists only and cannot compete with *greeNsort®*, and since a fair re-implementation would have been too complicated, the test-bed uses as a reference the *C++ implementation of Timothy Van Slyke from 2018*, the most competitive implementation I could find.[2]. Here some data on Timsort, the magenta lines show the *eFootprint* medians[3] projected onto the *Energy* scale (calculated as $Energy_{Knuthsort} \cdot \%RAM_{Knuthsort}/\%RAM_{Timsort}$).

[2]Sebastian Wild (see Munro and Wild (2018)) has developed alternatives to Timsort with a drop-in replacement that are simpler and do an 'optimized' (more balanced) merging, but still based on an inefficient merge. The code for *Peeksort* and *Powersort* was not available at the time of running the simulations for this report, meanwhile I measured them, see Update: Powersort. Meanwhile Peters (2023a);Peters (2023b) seems to have improved the copy efficiency in his newest *Glidesort*, however that is Rust code and hence not measured here

[3]For some implementations Timsort needs less than 50% buffer if no heavy merging is needed, for perfectly presorted data the buffer memory can be zero (`%RAM=1`). In the *Timsort* code, *%RAM* is difficult to measure in a non-invasive way, therefore the test-bed assumes `%RAM=1.5`: the amount of memory that needs to be available to safely run Timsort

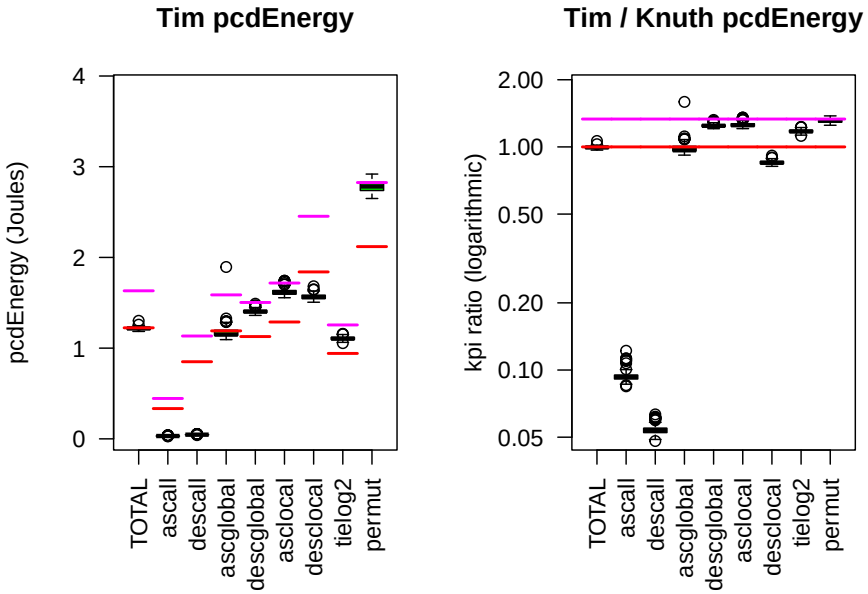

Figure 5.2: Timsort compared to Knuthsort. The left chart compares the absolute energy consumed for 8 input patterns and the TOTAL (of 5). The right chart shows the respective ratios to the median of the reference algorithm (here Knuthsort). The red/magenta lines show the Energy/eFootprint reference medians.

Table 5.2: Timsort / 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	0.75	0.98	0.99	0.75	-0.01	0.0000	0.0076	0
ascall	0.75	0.09	0.09	0.07	-0.30	0.0000	0.0000	0
descall	0.75	0.05	0.05	0.04	-0.80	0.0000	0.0000	0
ascglobal	0.75	1.01	0.97	0.72	-0.04	0.0786	0.0002	0
descglobal	0.75	1.24	1.24	0.93	0.27	0.0000	0.0000	0
asclocal	0.75	1.21	1.25	0.94	0.32	0.0000	0.0000	0
desclocal	0.75	0.83	0.85	0.64	-0.28	0.0000	0.0000	0
tielog2	0.75	1.21	1.17	0.88	0.16	0.0000	0.0000	0
permut	0.75	1.27	1.31	0.98	0.66	0.0000	0.0000	0

The `TOTAL` data show that *Timsort* is not much better than *Knuthsort* on the *RUNtime* and *pcdEnergy* KPIs: The improvements for presorted data are offset by deterioration for heavy sorting tasks. On the *pcdFootprint* KPI is is much better, because it requires only 75% memory.

5.5 Pre-Adpative (Omitsort)

```
innovation("Data-driven-location","B","data decides return memory")
innovation("Omitsort","A","skip merge and return pointer to data")
dependency("Omitsort", "Data-driven-location")
dependency("Omitsort","Split&Merge")
```

I now disclose how to modify *Knuthsort* to be adaptive for presorted data: differing from the prior art *greeNsort®* gives-up control over the location of the merged-data. If the two input sequences are not overlapping and merging is not needed, then it is possible to simply omit merging and the data stays where it was. The recursive function returns in which of the two memory areas the data resides. I call this method *Data-driven-location* and the resulting *Omitsort* has the following properties:

- like *Knuthsort* it uses 100% buffer (unlike *Timsort* which needs not more than 50%)
- unlike *Knuthsort* it does not require redundant upfront copying from data to buffer
- like *Knuthsort* and unlike *Timsort* it uses an efficient merge which needs only one move per element
- unlike *Knuthsort* and like *Timsort* for perfectly presorted branches no elements will be moved
- like *Timsort*, *Omitsort* has a $\mathcal{O}(N)$ best-case: due to not more than N non-overlap checks (and no moves)
- like *Knuthsort* and unlike *Timsort* a fully parallel implementation is possible

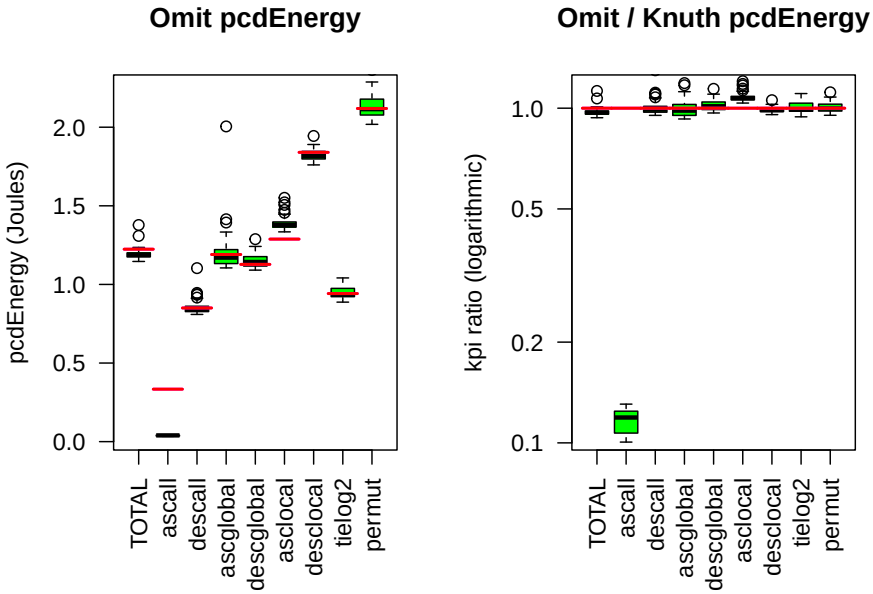

Figure 5.3: Omitsort compared to Knuthsort

Table 5.3: Omitsort / 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	1	0.97	0.97	0.97	-0.04	0.0000	0.0000	0.0000
ascall	1	0.11	0.12	0.12	-0.29	0.0000	0.0000	0.0000
descall	1	0.98	0.99	0.99	-0.01	0.0000	0.0015	0.0015
ascglobal	1	1.00	0.98	0.98	-0.02	0.0015	0.1706	0.1706
descglobal	1	1.00	1.01	1.01	0.02	0.4777	0.0374	0.0374
asclocal	1	1.07	1.07	1.07	0.09	0.0000	0.0000	0.0000
desclocal	1	0.99	0.99	0.99	-0.02	0.0001	0.0000	0.0000
tielog2	1	0.99	0.99	0.99	-0.01	0.0000	0.1050	0.1050
permut	1	1.00	1.00	1.00	0.00	0.8920	0.6938	0.6938

Compared to *Knuthsort*, *Omitsort* needs only about 97% *Energy*, for presorted data it is 12%.

5.6 Bi-adpative (Octosort)

```
innovation("Undirected-sorting","C","not requesting asc/desc order")
innovation("Data-driven-order","B","data decides asc/desc order")
innovation("Octosort","A","lazy directed undirected sort")
dependency("Data-driven-order", "Undirected-sorting")
dependency("Octosort", "Data-driven-location")
dependency("Octosort", "Data-driven-order")
```

Omitsort still has a limitation: it is adaptive to presorted (e.g. left-ascending) but not to reverse-sorted (e.g. left-descending) data. If the data is left-descending, then sorting incurs $\mathcal{O}(N \log N)$ moves (and depending on implementation details also comparisons). But if a reversed Omitsort had been used which sorts left-descending, the caller could reverse the final descending sequence with not more than N moves (and N comparisons for stability), which would still be a $\mathcal{O}(N)$ best-case. What to do without upfront knowledge that the data was descending from left?

I now disclose how to modify *Omitsort* to be adaptive for presorted and reverse-sorted data: I drop the prior art assumption that the sorting order needs to be specified in the API to the recursive function. It is possible to let the recursive function decide whether to sort or to reverse-sort and only lazily enforce the desired order. After the recursion is finished, the obtained order can be returned (undirected API) or the desired order can be forced (directed API). I call this *Data-driven-order* and the resulting *Octosort* has the following properties:

- like *Omitsort*, the presorted best-case costs less than N comparisons (and no moves)
- unlike *Omitsort*, the reverse-sorted best-case costs less than $2N$ comparisons and N moves
- like *Timsort*, the best-case is $\mathcal{O}(N)$ for presorted and reverse-sorted data

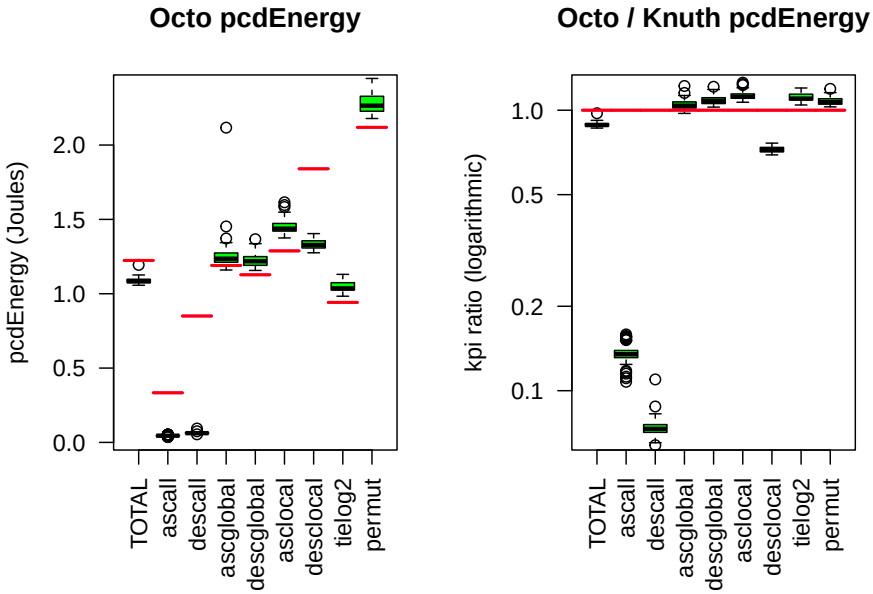

Figure 5.4: Octosort compared to Knuthsort

Table 5.4: Octosort / 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	1	0.88	0.89	0.89	-0.14	0	0	0
ascall	1	0.14	0.13	0.13	-0.29	0	0	0
descall	1	0.07	0.07	0.07	-0.79	0	0	0
ascglobal	1	1.04	1.04	1.04	0.05	0	0	0
descglobal	1	1.07	1.08	1.08	0.09	0	0	0
asclocal	1	1.10	1.12	1.12	0.15	0	0	0
desclocal	1	0.72	0.72	0.72	-0.51	0	0	0
tielog2	1	1.08	1.10	1.10	0.09	0	0	0
permut	1	1.07	1.07	1.07	0.15	0	0	0

Compared to *Knuthsort*, *Octosort* needs only about 90% *Energy*, for presorted and reverse-sorted data it is 14% and 7%.

5.7 Gapped-merging (GKnuthsort)

```
innovation("Gapped-setup","B","odd-even data-buffer layout")
dependency("Gapped-setup", "Ordinal-machine")
innovation("Gapped-merging","B","merge in odd-even data-buffer layout")
dependency("Gapped-merging", "Gapped-setup")
innovation("GKnuthsort","A","Knuthsort in odd-even data-buffer")
dependency("GKnuthsort", "Gapped-merging")
dependency("GKnuthsort", "Knuthsort")
```

Now I address a more fundamental problem of the prior art *Split&Merge* algorithms: they move the data always between two distant memory regions A and B. Even if A and B are contiguous neighbors, the minimum distance of the $\mathcal{O}(N \log N)$ moves is N. This differs dramatically from the distance-behavior of *Quicksorts*. The prior art knows that Quicksorts are *cache-friendly*, but it underappreciates that Quicksorts are *distance-friendly*: Quicksort recursion zooms into local memory such that the maximum distance is the current local size n of the problem (and not the global problem-size N).

Now I drop one prior art assumption behind those prior art Split&Merge algorithms: that data and buffer are separate memory regions. One way to make Split&Merge algorithms more distance-friendly is to define A and B as odd and even positions in contiguous memory. Doing so gives a local problem size $2n$. Adapting *Knuthsort* to *Gapped-merging* gives *GKnuthsort*:

Table 5.5: Sketch of GKnuthsort

position	1	2	3	4	5	6	7	8
setup	4	.	3	.	2	.	1	.
merge	.	3	.	4	.	1	.	2
merge	1	.	2	.	3	.	4	.

However, this approach has the following shortcomings

- elements to be sorted must have equal size
- it is not fast on current hardware

Speed might be improved by using odd and even blocks of data and buffer, say of size 1024. However, I will develop this idea into a more interesting direction.

5.8 Buffer-merging (TKnuthsort, Crocosort)

```
innovation("Buffer-merging","B","merging data and buffer")
innovation("Buffer-splitting","B","splitting data and buffer")
dependency("Buffer-merging","Split&Merge")
dependency("Buffer-splitting","Split&Merge")
dependency("Buffer-merging","Gapped-setup")
innovation("TKnuthsort","A","Knuthsort with buffer-merging (T-moves)")
dependency("TKnuthsort", "Buffer-merging")
dependency("TKnuthsort", "GKnuthsort")
innovation("Crocosort","A","Knuthsort with buffer-merging (R-moves)")
dependency("Crocosort", "Buffer-merging")
dependency("Crocosort", "GKnuthsort")
```

Now I drop a prior art assumption behind prior art *Split&Merge* algorithms: they focus on merging the data, but not the buffer. *greeNsort®* treats the buffer like the data as a first-class objective of merging. Merging must split and merge *data and buffer*, then bigger and bigger contiguous regions of data and of buffer are obtained as the merging proceeds.

A naive way of merging data *and* buffer is self-recursion which maintains an invariant data-buffer pattern, for example placing data left and buffer right within local memory. After a gapped setup, in order to maintain this pattern, data must first be transferred to one half of the memory and then merged to the other (*TKnuthsort*):

Table 5.6: Sketch of TKnuthsort

position	1	2	3	4	5	6	7	8
setup	4	.	3	.	2	.	1	.
transfer	.	.	4	3	.	.	2	1
merge	3	4	.	.	1	2	.	.
transfer	.	.	.	.	3	4	1	2
merge	1	2	3	4	.	.	.	.

This can be somewhat optimized to not transfer all data, but only that data which is in the target half for merging. With a little care to not ruin stability, *Crocosort* gets away with only 50% relocation:

Table 5.7: Sketch of Crocosort

position	1	2	3	4	5	6	7	8
setup	1	.	2	.	3	.	4	.
relocate	.	.	2	1	.	.	4	3
merge	1	2	.	.	3	4	.	.
relocate	.	.	.	.	3	4	1	2
merge	1	2	3	4	.	.	.	.

However, on current CPUs, also the extra 50% relocation moves are too expensive and are slower than simple *Knuthsort*. What to do?

5.9 Symmetric-language method

```
innovation("Symmetric-language","B","symmetric semi-in-place merging data and buffer")
dependency("Symmetric-language", "Buffer-merging")
dependency("Symmetric-language", "Symmetric-sorting")
dependency("Symmetric-language", "Symmetric-recursion")
dependency("Symmetric-language", "Footprint")
```

Now I disclose the solution to distance-reducing data- and buffer-merging: instead of self-recursion, *Symmetric-merging* uses *Symmetric-recursion* over *Buffer-asymmetry*, where the left branch sorts 'to left' (i.e. data left, buffer right) and the right branch sorts 'to right' (i.e. data right, buffer left). As a result, the two buffer regions border each other and form one contiguous buffer space without requiring any extra moves.

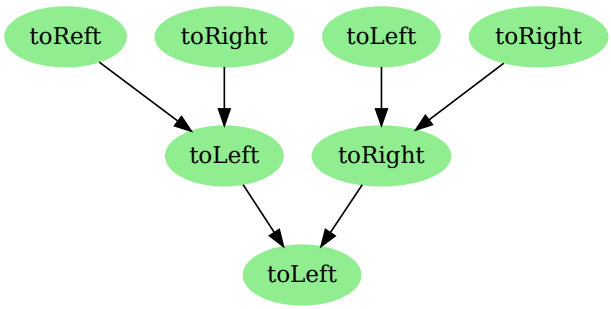

Figure 5.5: Symmetric-language

The following is a description for symmetric Divide&Conquer, i.e. Split&Merge as well as Partition&Pool:

Let LB denote a buffer-asymmetric chunk of memory with data left and buffer right and let BR denote a buffer-asymmetric chunk with data right and buffer left. Then a symmetric pair of left and right chunks $LBBR$ have a joint contiguous buffer memory BB in the middle and can be conquered either to the left into a bigger $LLBB$- chunk (which replicates the structure of the LB-chunk) or to the right to a bigger $BBRR$-chunk (which replicates the structure of the BR-chunk). Now consider a pair of such $LBBR$ pairs $LBBRLBBR$, and divide and conquer the left pair to the left and the right pair to the right to obtain $LLBBBBRR$ (which replicates the structure of $LBBR$ with a contiguous inner buffer area) and hence can be conquered again to the left or the right.

This recursive symmetric asymmetric data structure can be exploited with two flip-recursive merging functions `toLeft()` resp. `toRight()` that divide their $LLBB$ resp. $BBRR$ input into two smaller chunks LB and BR, call `toLeft()` on LB and `toRight()` on BR and then conquer them to $LLBB$ resp. $BBRR$. `toLeft()` reads and writes the data in $LBBR$ from right-to-left, `toRight()` reads and writes the data in $LBBR$ from left-to-right.

It is understood that the recursive functions toLeft() and toRight() are designed to not further divide if the input chunk is small enough; as a stopping criterion I contemplate a chunk that contains only a single element (and hence the chunk is sorted), or I contemplate a greater limit to the size of the chunk, at which the recursive function calls a different function to sort the elements in the chunk (for example tuning with *Insertionsort*). It is understood that by calling `toLeft()` or `toRight()` on a suitable top-level chunk the data in that chunk gets finally sorted and that data and buffer both are returned in a contiguous memory area, although during recursion many chunks and hence many gaps existed. It is understood that the real sorting work of $\mathcal{O}(N \cdot logN)$ comparing and moving is done in the merging (or partitioning phases), while the splitting (or pooling) phases do not require more than ON moves (and obviously no comparisons). Here is a simple example for merging four keys using four buffer elements:

Table 5.8: Sketch of symmetric merging, buffer is returned merged

position	1	2	3	4	5	6	7	8
setup to	L	B	B	R	L	B	B	R
setup do	4	.	.	3	2	.	.	1
merge to	L	L	B	B	B	B	R	R
merge do	3	4	.	.	.	.	1	2
merge to	L	L	L	L	B	B	B	B
merge do	1	2	3	4	.	.	.	.

Now I generalize: symmetric merging needs no more than 50% buffer space. Let I denote 'inner' and O denote 'outer' and write $LBBRLBBR$ as $OBBIIBB$. As before $OBBI$ is merged to the left and $IBBO$ is merged to the right, hence both are merged (read and written) from inner to outer. Note that the outer data ought have the target order of sorting, for example ascending. Reading the inner and outer data starts from the innermost positions of I and O and proceeds to the outermost positions; and writing starts in BB at position $|I|+|O|$ relative to the outermost position of O. The merge is completed once the inner data is exhausted, the rest of the outer data is already in-place. The minimum size of BB is hence that of I, hence required is $|BB| \geq |I|$. If the splitting in symmetric Split&Merge is done such that the size of I is smaller or equal to the size of O, hence required is $|I| \leq |O|$, it follows that a buffer size of I is sufficient. For a balanced split where $|I| = |O|$ this amounts to 50% buffer. To better represent the relative sizes we may write B instead of BB, and hence OBI (or IBO) for two chunks before merging to outer, where $|I| \leq |O|$ and $|I| = |B|$.

Table 5.9: Sketch of optimized symmetric merging, with only half buffer

position	1	2	3	4	5	6
setup to	L	B	R	L	B	R
setup do	4	.	3	2	.	1
merge to	L	L	B	B	R	R
merge do	3	4	.	.	1	2
merge to	L	L	L	L	B	B
merge do	1	2	3	4	.	.

Now recap the benefits of *Symmetric-merging*

- unlike *GKnuthsort* it operates on contiguous data
- like *Knuthsort* if is fully generic (stable, $\mathcal{O}(N \log N)$, allows variable size, allows parallelization)
- like *Knuthsort* (and Sedgewick's Nocopy-mergesort) there are no unneeded moves and can be tuned
- but it needs not more than 50% buffer (where *Knuthsort* needs 100%)
- unlike *Knuthsort* it zooms into local memory and minimizes distances (similar to *Quicksorts* and *Z:cksorts*)

5.10 Frogsort & Geckosort

```
innovation("Frogsort","B","symmetric-merging ½-adaptive presorting")
dependency("Frogsort","Symmetric-language")
dependency("Frogsort","Buffer-splitting")
innovation("Geckosort","B","symmetric-merging ¼-adaptive pre/rev")
dependency("Geckosort","Symmetric-language")
dependency("Geckosort","Buffer-splitting")
```

Now let's take a closer look at the *Symmetric-merging*. There are two algorithms to be distinguished, that I have named *Frogsort* and *Geckosort*, note that both names contain acronyms: *(F)lip (R)ecursive (O)rganized (G)aps (FROG)* and *(G)ap (E)xchange (C)an (K)eep (O)rder (GECKO)*. In Frogsort the sorting order in both branches is the same, for example *AscLeft*:

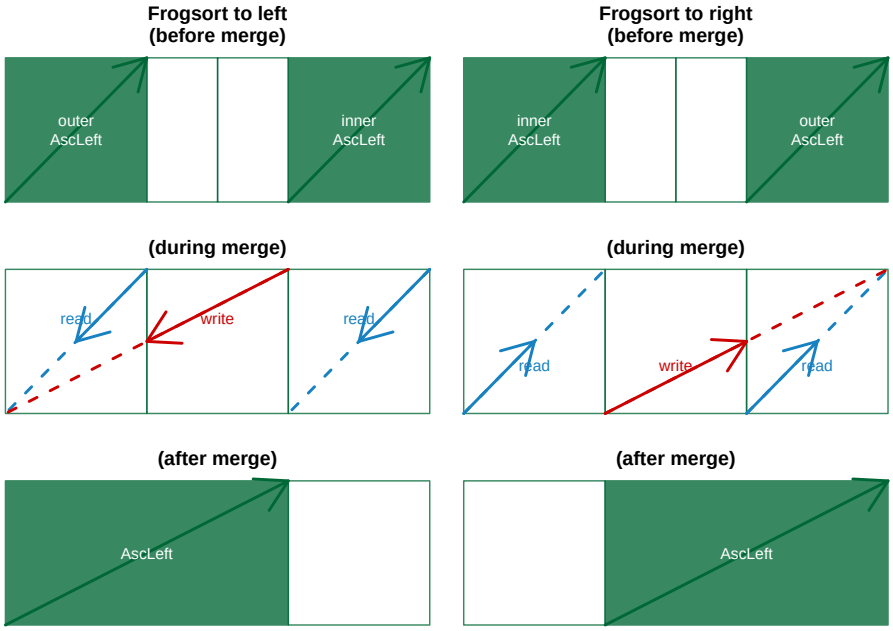

Figure 5.6: Frogsort merges to left and to right (here AscLeft)

Note that Frogsort is automatically adaptive to pre-sorted data: the *outer sequence* remains in place, only the inner sequence is compared and moved. Once the inner sequence is moved, merging stops. This means that the cost of merging reduces from N comparisons and moves to $N/2$ comparisons and moves. By investing an extra comparison to identify non-overlap, the $N/2$ comparisons can also be saved. Hence Frogsort has yet another asymmetry: *Order-asymmetry* regarding adaptivity. There is an alternative to Frogsort.

Geckosort uses in the right branch the flipped order of the left branch:

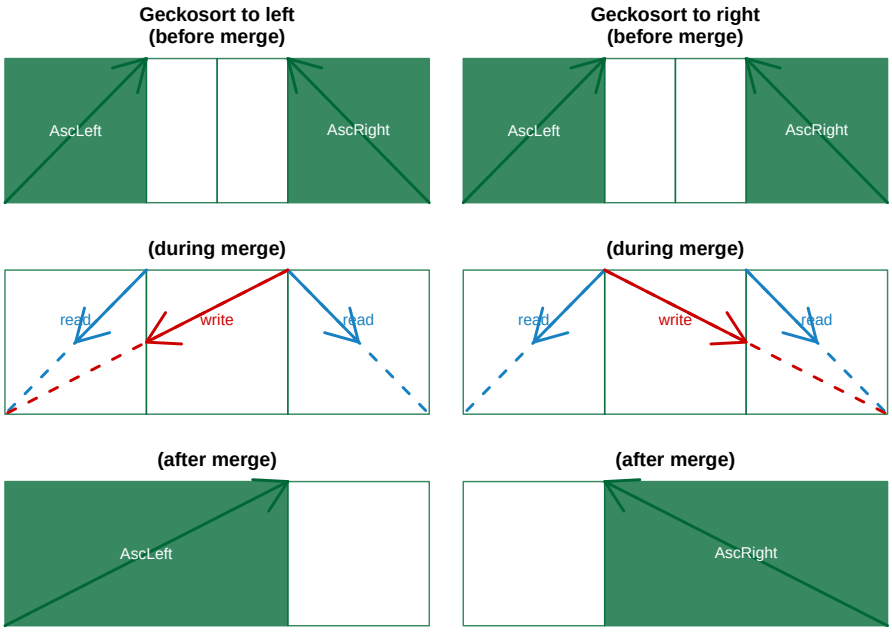

Figure 5.7: Geckosort merges to left and to right (here Ascending)

As an example here follows the beautifully symmetric code for *Geckosort1* in a simplified version without register caching and without insertion-sort tuning (full explanation in section *Engineering Frogsort1*):

```
void Geckomerge_toright_ascleft(
  ValueT *x
, ValueT *ll, ValueT *lr
, ValueT *rl, ValueT *rr
){
  for(;;){
    if (LT(*rl,*lr)){
      *(x++) = *rl++;
      if (rl>rr) break;
    }else{
      *(x++) = *lr--;
      if (ll>lr) return;
    }
  }
  while(ll<=lr){
    *(x++) = *lr--;
  }
  return;
}
void Geckomerge_toleft_ascright(
    ValueT *x
  , ValueT *ll, ValueT *lr
  , ValueT *rl, ValueT *rr
){
  for(;;){
    if (LT(*rl,*lr)){
      *(x--) = *rl++;
      if (rl>rr) return;
    }else{
      *(x--) = *lr--;
      if (ll>lr) break;
    }
  }
  while(rl<=rr){
    *(x--) = *rl++;
  }
  return;
}
```

```
void Geckoinit1_toleft_ascright(
  ValueT *x , ValueT *y
, IndexT n, IndexT l, IndexT L
){
  IndexT nr;
  if (n <= 2){
    if (n == 2){
      ValueT t;
      if (LE(x[l], x[l+1])){
        t = x[l+1]; y[L+1] = x[l]; y[L] = t; // no in-place overwrite
      }else{
        y[L+1] = x[l+1]; y[L] = x[l];
      }
    }else{
      y[L] = x[l];
    }
  }else{
    nr = n  / 2;
    Geckoinit1_toleft_ascright(x, y, n - nr, l    , L       );
    Geckoinit1_toright_ascleft(x, y, nr    , l+n-1, L+n+nr-1);
  }
}
void Geckoinit1_toright_ascleft(
  ValueT *x, ValueT *y
, IndexT n, IndexT r, IndexT R
){
  IndexT nl; ValueT t;
  if (n <= 2){
    if (n == 2){
      if (GT(x[r-1], x[r])){
        t = x[r-1]; y[R-1] = x[r]; y[R] = t; // no in-place overwrite
      }else{
        y[R-1] = x[r-1]; y[R] = x[r];
      }
    }else{
      y[R] = x[r];
    }
  }else{
    nl = n  / 2;
    Geckoinit1_toleft_ascright(x, y, nl    , r-n+1, R-n-nl+1);
    Geckoinit1_toright_ascleft(x, y, n - nl, r    , R       );
  }
}
```

```
void Geckosort1_toleft_ascright(
    ValueT *y
  , IndexT n
  , IndexT L
){
  if (n > 2){
    IndexT nr = n  / 2;
    Geckosort1_toleft_ascright (y, n - nr, L       );
    Geckosort1_toright_ascleft(y, nr    , L+n+nr-1);
    Geckomerge_toleft_ascright(y+L+n-1, y+L, y+(L+(n-nr)-1), y+(L+n), y+(L+n+nr-1));
  }
}
void Geckosort1_toright_ascleft(
    ValueT *y
  , IndexT n
  , IndexT R
){
  if (n > 2){
    IndexT nl = n  / 2;
    Geckosort1_toleft_ascright(y, nl    , R-n-nl+1);
    Geckosort1_toright_ascleft(y, n - nl, R       );
    Geckomerge_toright_ascleft(y+R-n+1, y+(R-n-nl+1), y+(R-n), y+(R-(n-nl)+1), y+R);
  }
}
```

Note that *Geckosort* is as stable as *Frogsort*. Note further, that Geckosort is more symmetric than Frogsort: one branch is adaptive to pre-sorting, the other branch is adaptive to reverse-sorted data. This is reflected in the naming: Frogs are asymmetric regarding locomotion: the back legs of the frog are stronger than its front legs, this helps jumping forward. The Gecko is more symmetric with equally strong back and front legs, which supports his preferred hunting technique of invisibly slow sneaking up on the prey. Note finally, that Geckosort is not fully symmetric, because ties are considered presorted in one branch, and must be reversed in the other. Let's recap the adaptivity implications of *Symmetric-merging*. *Frogsort* and *Geckosort* have four benefits over *Bimesort* and *Knuthsort*:

- they need not more (rather less) than 50% buffer compared to 100% in the prior art
- Symmetric-language reduces distances by zooming into local memory
- Symmetric-language needs only half the cost for presorted data (without any tuning)
- there is a choice between half-adaptivity for one order in Frogsort and quarter-adaptivity for both orders in Geckosort

These benefits come at minor costs and limitations:

- the code is more complex (albeit with symmetric redundancy that can be exploited in formal *Code-mirroring*)
- a $\mathcal{O}(N)$ setup-phase is needed to distribute data with buffer gaps (*Gapped-setup*)
- like *Bimesort* but unlike *Knuthsort* even for perfectly presorted data, half of the elements must be moved, hence the cost remains $\mathcal{O}(N \log N)$ and cannot be tuned to $\mathcal{O}(N)$ as was possible for Knuthsort with *Omitsort* and *Octosort*.

Now lets look at several ways to actually implement Frogsort (similar possibilities exist for Geckosort).

5.10.1 Engineering Frogsort0

```
innovation("Frogsort0","A","Frogsort on triplets or bigger chunks")
dependency("Frogsort0","Frogsort")
```

Frogsort0 Divides&Conquers *triplets* of memory cells: two cells contain two elements and a third empty buffer cell in the middle. The buffer in the middle allows to save one move for presorted data. The initial setup of triplets costs N moves and can be done in a simple loop, parallelization is possible up to the throughput of the memory-system. Note that for merging an odd number of triplets the splitting rule must assign the excess element to the outer branch, otherwise, if the first element is merged from the inner sequence, its target position is still occupied by the last element merged from the inner sequence. Odd numbers of elements can be handled in various ways:

- an extreme dummy-value can be used to complete the last triplet: after sorting it must be located in the extreme position bordering the buffer where it can be easily removed
- alternatively this can be resolved by *T-level technique - (COM)puting on the (REC)ursion (COMREC)*: at the outer (e.g. right) edge of the recursion tree a a dedicated function can handle incomplete triplets

Here is an example of the latter approach:

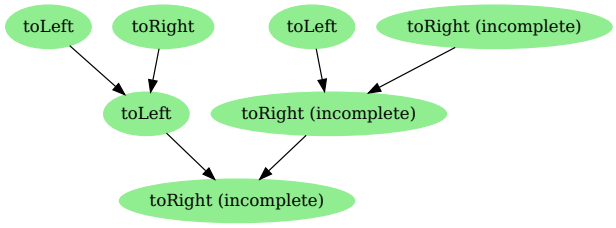

Figure 5.8: Frogsort zero with recursion for incomplete triplet

The recursion tree has $\mathcal{O}(N)$ calls, of which only $\mathcal{O}(logN)$ calls need to handle the special case, in other words: this has negligible cost.

Alternatively to triplets, bigger chunks can be used, for example composed of 16 elements left, 16 cells buffer and 16 elements right. This neatly integrates with tuning with *Insertionsort*.

For mixed input patterns (TOTAL) the test-bed measures for *Frogsort0* lower *eFootprint* (below the magenta line, resp.`r(pcdfootprint))` than the reference *Knuthsort* (and somewhat more *Energy*, around the red line, resp.`r(pcdenergy)`):

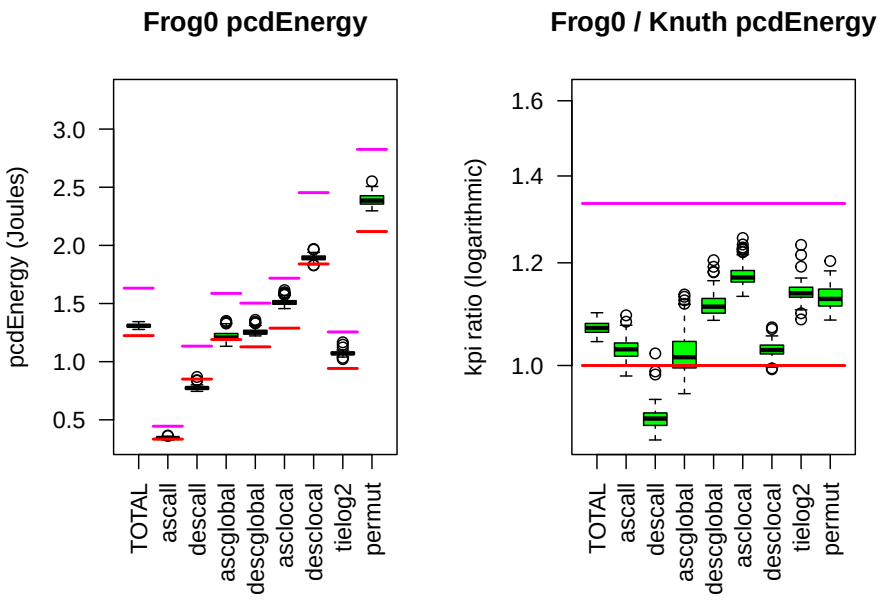

Figure 5.9: Frogsort0 compared to Knuthsort

Table 5.10: Frogsort0 / 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	0.75	1.08	1.07	0.80	0.08	0.0000	0.0000	0
ascall	0.75	1.06	1.03	0.77	0.01	0.3331	0.3047	0
descall	0.75	0.90	0.91	0.68	-0.08	0.0000	0.0000	0
ascglobal	0.75	1.07	1.01	0.76	0.02	0.0000	0.0032	0
descglobal	0.75	1.13	1.11	0.83	0.12	0.0000	0.0000	0
asclocal	0.75	1.19	1.17	0.88	0.22	0.0000	0.0000	0
desclocal	0.75	1.04	1.03	0.77	0.05	0.0000	0.0000	0
tielog2	0.75	1.15	1.14	0.85	0.13	0.0000	0.0000	0
permut	0.75	1.14	1.13	0.84	0.27	0.0000	0.0000	0

Compared to *Knuthsort*, *Frogsort0* needs only 80% *eFootprint* (but 7% more *Energy*).

5.10.2 Engineering Frogsort1

```
innovation("Frogsort1","A","balanced Frogsort on single elements")
dependency("Frogsort1","Frogsort")
```

Another approach to handle odd and even number of elements is *Frogsort1*: to divide&conquer single elements together with buffer cells according to a simple rule: the number of required buffer cells is always $\lfloor N/2 \rfloor$. Note that in case of odd N the smaller number of elements goes to the inner branch and the larger number to the outer branch. A convenient way to setup the data with the necessary buffer gaps is to run the recursion twice: first for the setup and then for the merging. In the setup phase there is no merging and the splitting moves no data until the leafs of the recursion tree are reached, where the input data is written to its target position 'to left' or 'to right' (and sorted in case the leaf contains more than one element). Once the data has been setup, the recursion is run again using the same splitting logic but this time with merging.

While this approach is elegant and efficient for single-threaded sorting, parallelizing the setup phase is harder and less efficient than in *Frogsort0*. Hybrid approaches are possible, where the static setup of Frogsort0 is combined with Frogsort1's splitting rule, however this involves modulo operations.

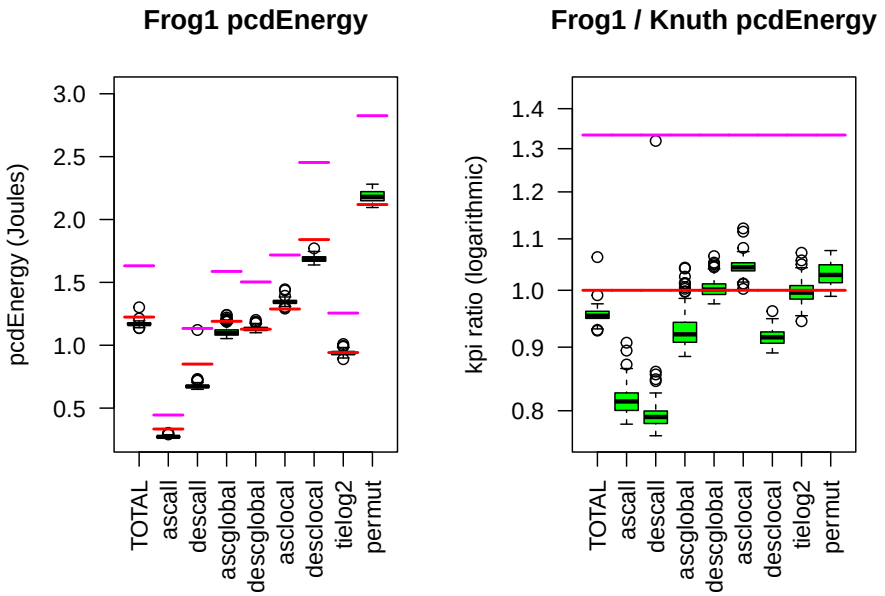

Figure 5.10: Frogsort1 compared to Knuthsort

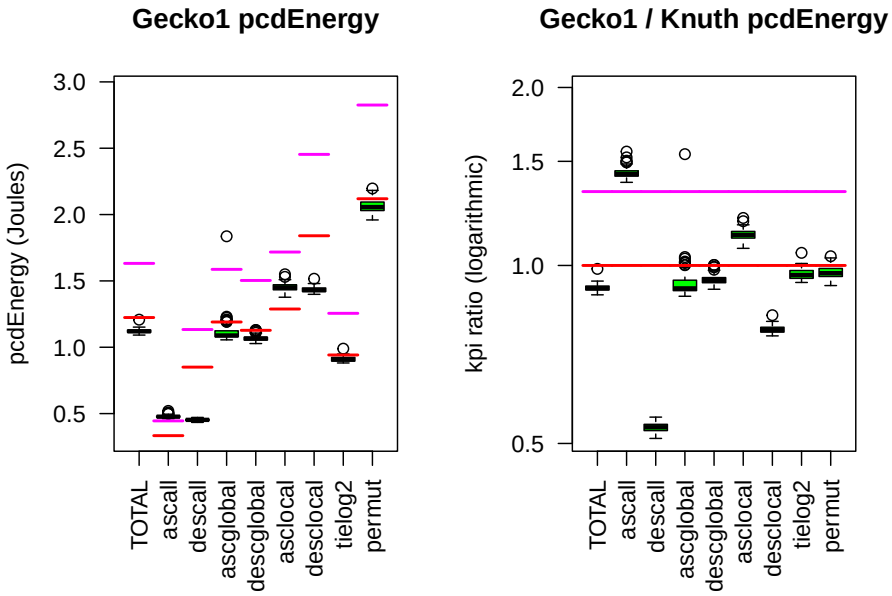

Figure 5.11: Geckosort1 compared to Knuthsort

Table 5.11: Frogsort1 / 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	0.75	0.96	0.95	0.72	-0.06	0.0000	0.0000	0
ascall	0.75	0.82	0.81	0.61	-0.06	0.0000	0.0000	0
descall	0.75	0.78	0.79	0.59	-0.18	0.0000	0.0000	0
ascglobal	0.75	0.96	0.92	0.69	-0.09	0.0000	0.0000	0
descglobal	0.75	1.02	1.00	0.75	0.00	0.7245	0.6339	0
asclocal	0.75	1.06	1.04	0.78	0.06	0.0000	0.0000	0
desclocal	0.75	0.93	0.92	0.69	-0.15	0.0000	0.0000	0
tielog2	0.75	1.00	1.00	0.75	0.00	0.0001	0.0013	0
permut	0.75	1.04	1.03	0.77	0.06	0.0000	0.0000	0

Table 5.12: Geckosort1 / 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	0.75	0.92	0.92	0.69	-0.10	0	0	0.0000
ascall	0.75	1.55	1.43	1.07	0.14	0	0	0.0217
descall	0.75	0.50	0.53	0.40	-0.40	0	0	0.0000
ascglobal	0.75	0.96	0.91	0.69	-0.10	0	0	0.0000
descglobal	0.75	0.96	0.95	0.71	-0.06	0	0	0.0000
asclocal	0.75	1.14	1.13	0.84	0.16	0	0	0.0000
desclocal	0.75	0.79	0.78	0.58	-0.41	0	0	0.0000
tielog2	0.75	0.98	0.96	0.72	-0.03	0	0	0.0000
permut	0.75	0.99	0.97	0.73	-0.06	0	0	0.0000

Compared to *Knuthsort*, *Frogsort1* needs only 96% of *Energy* and 72% of *eFootprint*, *Geckosort1* needs 92% *Energy* and 69% *eFootprint*.

5.10.3 Engineering Frogsort2

```
innovation("Frogsort2","A","imbalanced splitting Frogsort")
dependency("Frogsort2","Frogsort")
```

Now I drop yet another prior art assumption: that *balanced* divide&conquer is always to be preferred over *imbalanced* divide&conquer, due to fewer operations. I have intentionally stated above that *Symmetric-merging* needs *not more* than 50% buffer, it can be done with less: unbalanced symmetric Split&Merge, where $|B| = |I| < |O|$. $p_2 \leq 0.5$ specifies a fraction of buffer. *Frogsort2* uses a simple splitting rule: to sort 100% data with a fraction $p_2 = |B|/(|I| + |O|)$ of buffer, the outer branch handles a $1 - p_2$ fraction of the data with a $(1 - p_2) * p$ fraction of the buffer, the inner branch handles a p_2 fraction of data with a p^2 fraction of the buffer.

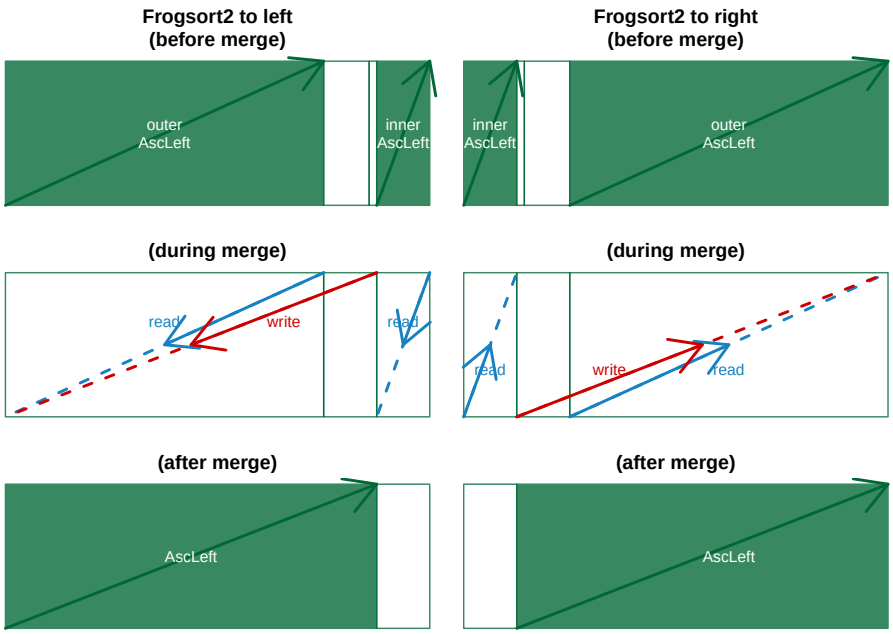

Figure 5.12: Frogsort2 (1/7 buffer) merges to left and to right (here AscLeft)

Frogsort1 is a special case of *Frogsort2* with $p_2 = 0.5$. Smaller p improves the *%RAM* measure. The prior art expects smaller p_2 (more imbalanced divide&conquer) to cause more operations and hence more *RUNtime*. The prior art expects an inverse-U-shaped function with an optimal *Footprint* measure p_2 below 50% buffer. To the surprise of the prior art, Frogsort2 not only delivers a better *Footprint*, it also is *faster* than Frogsort1 despite using less memory! The reason is – according to *greeNsort®* analyses – fewer branch-mispredictions due to the higher imbalance. In the test-bed by default Frogsort2 uses only $p_2 = 1/7$

(14.3%) buffer, which is the sweet spot regarding *speed* for sorting `doubles`. The sweet spot regarding *eFootprint* is below at about 7% buffer.

greeNsort® has experimented with implementations of *Knuthsort* and *Frogsort1* that reduce branch-misprediction by "nudging" compilers towards conditional moves, this was successful only for certain compiler versions. *Frogsort2* delivers similar speed-ups for *all* compiler versions, so seems more *resilient*.

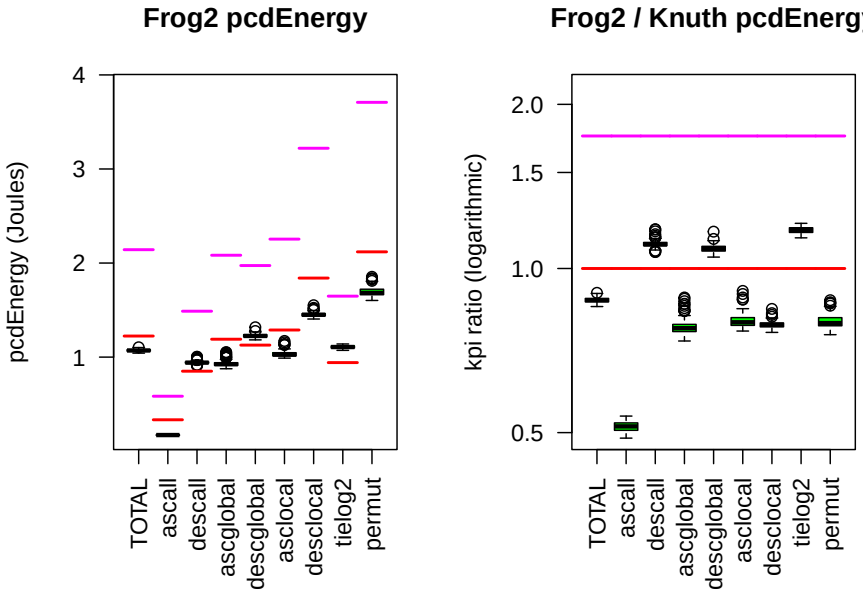

Figure 5.13: Frogsort2 compared to Knuthsort

Table 5.13: Frogsort2 / 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	0.86	0.87	0.50	-0.15	0	0	0
ascall	0.57	0.51	0.51	0.29	-0.16	0	0	0
descall	0.57	1.07	1.11	0.63	0.09	0	0	0
ascglobal	0.57	0.80	0.78	0.44	-0.27	0	0	0
descglobal	0.57	1.06	1.09	0.62	0.10	0	0	0
asclocal	0.57	0.79	0.80	0.46	-0.26	0	0	0
desclocal	0.57	0.79	0.79	0.45	-0.39	0	0	0
tielog2	0.57	1.13	1.18	0.67	0.17	0	0	0
permut	0.57	0.79	0.79	0.45	-0.44	0	0	0

Frogsort2 needs 86% *RUNtime*, 87% *Energy* and 50% *eFootprint* of *Knuthsort*.

5.10.4 Engineering Frogsort3

```
innovation("Buffer-sharing","T*","dedicate buffer to one branch at a time")
innovation("Share&Merge","C*","recursively share buffer until can splitting")
innovation("Frogsort3","A","imbalanced sharing Frogsort")
dependency("Frogsort3","Frogsort1")
dependency("Frogsort3","Share&Merge")
dependency("Share&Merge","Split&Merge")
dependency("Share&Merge","Buffer-sharing")
```

Symmetric-merging can be done with even less memory. So far, in the *Split&Merge* paradigm, the buffer memory was divided between the left and right branch of recursion. This allowed to create fully parallel implementations that execute all branches of the recursion tree in parallel. Now I introduce a new concept, the *Share&Merge* paradigm: *Buffer-sharing* shares buffer memory between both *Divide&Conquer* branches. Buffer is used alternating: it is serially passed down one branch, then down the other branch. Buffer is shared recursively until the amount of buffer is big enough relatively to recursive problem size to switch to *Buffer-splitting* and potentially process the remaining branches in parallel (Split&Merge).

The new *Frogsort3* does *Symmetric-merging* in the *Share&Merge* paradigm, and as soon as possible switches to *Frogsort1*. Note that *Frogsort2* features global parallelism and Frogsort3 more localized parallelism; which is better depends on features of the memory subsystem. Note also that Frogsort1 is a special case of Frogsort3 with 50% buffer. In the test-bed by default Frogsort3 uses only $p_3 = 1/8$ (12.5%) buffer and regarding *eFootprint* it still performs very well with as little as 5% buffer.

The prior art knows an implementation technique of Mergesort which uses 50% buffer to first sort one half of the data, then the other half and finally merges the two halves. *Frogsort3* can be seen as a recursive generalization of this technique applied to *Frogsort*, with the optimization that it additionally reduces distance.

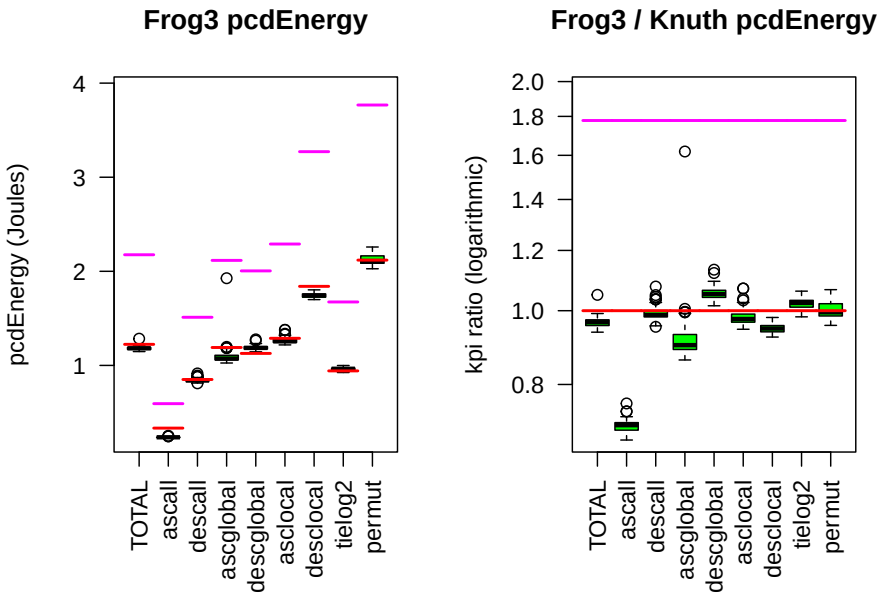

Figure 5.14: Frogsort3 compared to Knuthsort

Table 5.14: Frogsort3 / 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	0.56	0.97	0.97	0.54	-0.04	0.0000	0.0000	0
ascall	0.56	0.70	0.71	0.40	-0.10	0.0000	0.0000	0
descall	0.56	0.99	0.99	0.56	-0.01	0.0000	0.0000	0
ascglobal	0.56	0.95	0.90	0.51	-0.12	0.0000	0.0000	0
descglobal	0.56	1.06	1.05	0.59	0.06	0.0000	0.0000	0
asclocal	0.56	0.99	0.98	0.55	-0.03	0.0000	0.0000	0
desclocal	0.56	0.96	0.95	0.53	-0.10	0.0000	0.0000	0
tielog2	0.56	1.04	1.02	0.58	0.02	0.9904	0.2572	0
permut	0.56	1.01	1.00	0.56	-0.01	0.5394	0.8649	0

Compared to *Knuthsort*, *Frogsort3* needs 97% of *RUNtime*, 97% of *Energy* and only 54% of *eFootprint*.

5.10.5 Engineering Frogsort6

```
innovation("Frogsort6","A","generalized Frogsort")
dependency("Frogsort6","Frogsort1")
dependency("Frogsort6","Frogsort2")
dependency("Frogsort6","Frogsort3")
```

Frogsort2 and *Frogsort3* can be combined to to one generalized *Frogsort6* with two parameters p_2 and p_3 where $p_2 \geq p_3$ such that the algorithms switches from Frogsort3 to Frogsort2 as soon as the absolute buffer $b = p_3 \cdot N$ shared down the recursion tree reaches $b \geq p_2 \cdot n$ where n is the local problem size. Note the following special cases:

$algorithm$	p_2	p_3
$Frogsort1$	$= 0.5$	$= 0.5$
$Frogsort2$	< 0.5	$= p_2$
$Frogsort3$	< 0.5	$= 0.5$

Probably due to the larger code-size, *Frogsort6* performs slightly worse than the specialized *Frogsort1*, *Frogsort2* and *Frogsort3*.

```
innovation("Squidsort","B","lazy undirected symmetric sort")
dependency("Squidsort","Data-driven-order")
dependency("Squidsort","Frogsort")
```

It was already mentioned that *Symmetric-merging* cannot be tuned for $\mathcal{O}(N)$ best-case adaptivity because part of the data must be moved on each merge-call and hence the *Data-driven-location* method used in *Omitsort* is not available. However, the *Data-driven-order* method used in *Octosort* can be combined with Symmetric-language. The result are $\mathcal{O}(N \log N)$ algorithms that are *half-adaptive* to both, pre-sorted and reverse-sorted inputs, which is twice as adaptive than the simpler *Geckosort*, which is *quarter-adaptive* to both orders.

```
innovation("Squidsort1","A","lazy symmetric sort (50% buffer)")
dependency("Squidsort1","Squidsort")
dependency("Squidsort1","Frogsort1")
```

The *greeNsort®* test-bed implements *Squidsort1* as an bi-adaptive variant of *Frogsort1*.

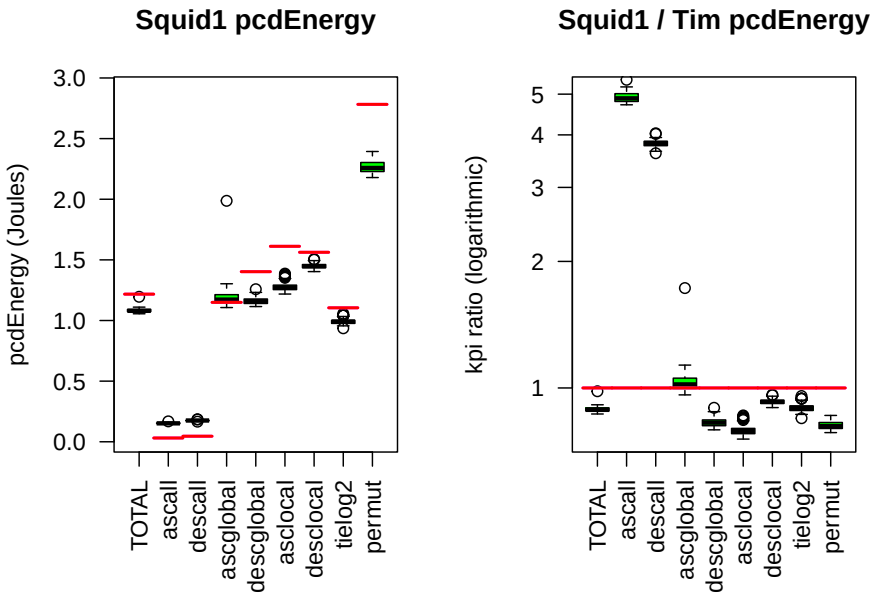

Figure 5.15: Squidsort1 compared to Timsort

Table 5.15: Squidsort1 / Timsort (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	1	0.90	0.89	0.89	-0.14	0.0000	0.0000	0.0000
ascall	1	5.02	4.89	4.89	0.12	0.0000	0.0000	0.0000
descall	1	3.76	3.83	3.83	0.13	0.0000	0.0000	0.0000
ascglobal	1	1.00	1.02	1.02	0.02	0.0027	0.0377	0.0377
descglobal	1	0.84	0.83	0.83	-0.24	0.0000	0.0000	0.0000
asclocal	1	0.82	0.79	0.79	-0.34	0.0000	0.0000	0.0000
desclocal	1	0.95	0.93	0.93	-0.11	0.0000	0.0000	0.0000
tielog2	1	0.88	0.90	0.90	-0.11	0.0000	0.0000	0.0000
permut	1	0.84	0.81	0.81	-0.52	0.0000	0.0000	0.0000

Compared to *Timsort*, *Frogsort1* needs about 90% *RUNtime*, *Energy* and *eFootprint*, for hard sorting work (`permut`) it is rather 83%. For *perfectly* presorted data Timsort needs less *Energy* by factor 4 to 5, but this advantage disappears quickly when very little noise disturbs perfect preorder. Also keep in mind that both algorithms are fast on presorted data, hence the absolute difference on presorted data is small, and the absolute difference on permuted data is 4 times bigger. Tuning the best case doesn't pay off according to the *DONOTUBE* principle. For an in-depth comparison see *greensort.org/results.html#deep-dive-deconstructing-timsort*.

5.11.2 Engineering Squidsort2

```
innovation("Squidsort2","A","lazy symmetric sort (<50% buffer)")
dependency("Squidsort2","Squidsort")
dependency("Squidsort2","Frogsort2")
```

The *greeNsort®* test-bed implements *Squidsort2* as an bi-adaptive variant of *Frogsort2*.

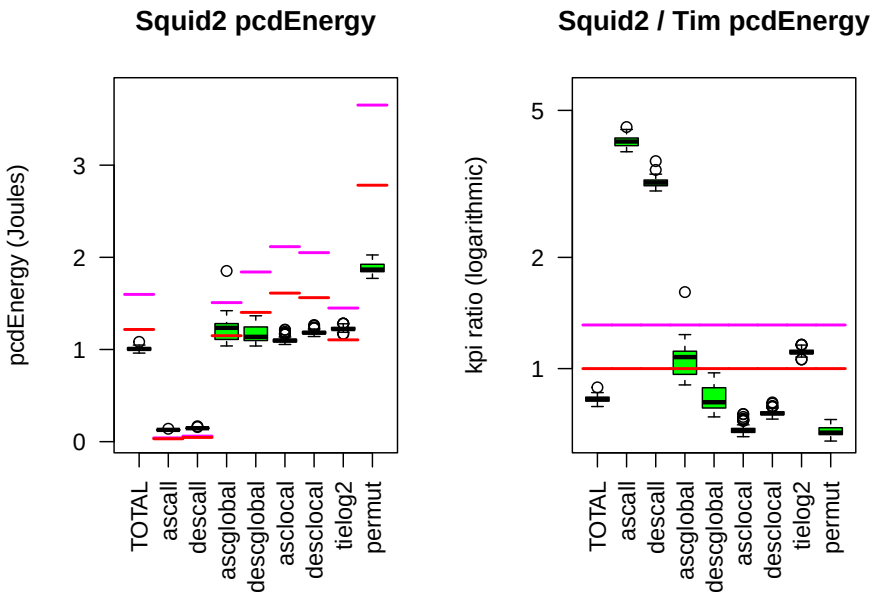

Figure 5.16: Squidsort2 compared to Timsort

Table 5.16: Squidsort2 / Timsort (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	0.76	0.83	0.83	0.63	-0.21	0.0000	0.0000	0
ascall	0.76	4.12	4.12	3.14	0.10	0.0000	0.0000	0
descall	0.76	3.07	3.19	2.43	0.10	0.0000	0.0000	0
ascglobal	0.76	1.03	1.07	0.82	0.09	0.1728	0.0015	0
descglobal	0.76	0.80	0.81	0.62	-0.27	0.0000	0.0000	0
asclocal	0.76	0.70	0.68	0.52	-0.52	0.0000	0.0000	0
desclocal	0.76	0.77	0.76	0.58	-0.38	0.0000	0.0000	0
tielog2	0.76	1.08	1.11	0.84	0.12	0.0000	0.0000	0
permut	0.76	0.69	0.67	0.51	-0.92	0.0000	0.0000	0

Compared to *Timsort*, *Frogsort2* needs about 83% *RUNtime* and *Energy* and 63% *eFootprint*, for hard sorting work (`permut`) it is rather 68% runTime and *Energy* and 52% *eFootprint*. For *perfectly* presorted data Timsort needs less *Energy* by factor 6, but this advantage disappears quickly when very little noise disturbs perfect preorder. Also keep in mind that both algorithms are fast on presorted data, hence the absolute difference on presorted data is small, and the absolute difference on permuted data is 9 times bigger. Tuning the best case doesn't pay off according to the *DONOTUBE* principle. For an in-depth comparison see *greensort.org/results.html#deep-dive-deconstructing-timsort*.

5.12 Update: Powersort

Compared to *Timsort*, *Peeksort* and *Powersort* are slightly more efficient on random data and slightly less efficient on presorted data. Overall they are not substantially better than Timsort and inferior to *Squidsorts* and *Frogsorts*. The following are the results of single measurements on AMD with `n=2^25`.

Let's look at *runTime* normalized by *Knuthsort* at randomly permuted data:

Table 5.17: runTime of Timsort alternatives normalized by Knuthsort with random data

	permut	asclocal	desclocal	ascglobal	descglobal	ascall	descall	ALL	ASC
Knuth	1.000	0.580	0.776	0.549	0.544	0.167	0.326	0.618	0.574
Omit	1.027	0.572	0.704	0.545	0.548	0.013	0.283	0.590	0.539
Octo	0.986	0.574	0.567	0.544	0.545	0.014	0.020	0.529	0.529
Gecko1	0.991	0.653	0.633	0.511	0.510	0.180	0.161	0.579	0.584
Frog1	0.981	0.589	0.690	0.510	0.526	0.103	0.215	0.574	0.546
Frog1A	0.964	0.582	0.681	0.498	0.537	0.076	0.234	0.567	0.530
Squid1	0.969	0.559	0.589	0.515	0.537	0.077	0.083	0.537	0.530
Frog2	0.717	0.433	0.589	0.400	0.547	0.061	0.352	0.477	0.403
Frog2A	0.721	0.420	0.597	0.411	0.549	0.042	0.362	0.478	0.399
Squid2	0.722	0.419	0.438	0.453	0.512	0.043	0.047	0.419	0.409
PFrog1	0.164	0.132	0.142	0.104	0.117	0.078	0.099	0.125	0.119
PFrog2	0.156	0.129	0.147	0.117	0.127	0.083	0.118	0.129	0.121
Tim	1.120	0.640	0.621	0.499	0.635	0.009	0.017	0.583	0.567
Peek	1.106	0.641	0.649	0.548	0.608	0.028	0.055	0.593	0.581
Power	1.085	0.647	0.644	0.557	0.588	0.029	0.034	0.584	0.579

There are two totals, `ASC` is the average cost over the `permut` and `asc*` datasets, `ALL` is the average cost over the `permut` (twice) and `asc*` and `desc*` datasets. By convention sorting is rather ascending, hence I consider `ASC` practically more relevant, `ALL` is included to provide a benchmark for bi-adaptive algorithms (*Octosort*, *Geckosort1*, *Squidsort1*, *Squidsort2*, *Timsort*, *Peeksort*, *Powersort*). While *Timsort*, *Peeksort* and *Powersort* excel at presorted data, they pay a high tuning price on randomly permuted data. In a mixed portfolio of tasks such as `ASC` they they play in the same league like a simple *Knuthsort*. The simple un-tuned *Frogsort2* algorithm is a clear speed-winner. Both parallel *Frogsorts* achieve a speedup of 8x on `ASC`: when parallelized, *Frogsort2* loses its speed

advantage over *Frogsort1*. Following their speed measurements, Munro and Wild (2018) have preferred *Powersort* over *Peeksort*, although top-down *Peeksort* could be parallelized, while bottom-up *Timsort* and *Powersort* algorithms are inherently serial.

Let's look at *Energy* normalized by *Knuthsort* at randomly permuted data:

Table 5.18: bcdEnergy of Timsort alternatives normalized by Knuthsort with random data

	permut	asclocal	desclocal	ascglobal	descglobal	ascall	descall	ALL	ASC
Knuth	1.000	0.479	0.643	0.461	0.459	0.145	0.285	0.559	0.521
Omit	1.029	0.465	0.598	0.458	0.450	0.010	0.250	0.536	0.490
Octo	0.813	0.469	0.476	0.459	0.459	0.010	0.015	0.439	0.438
Gecko1	0.803	0.540	0.524	0.427	0.430	0.165	0.145	0.480	0.484
Frog1	0.798	0.484	0.570	0.419	0.438	0.089	0.192	0.474	0.448
Frog1A	0.793	0.472	0.581	0.416	0.448	0.061	0.211	0.472	0.435
Squid1	0.793	0.466	0.482	0.426	0.443	0.063	0.067	0.442	0.437
Frog2	0.610	0.380	0.509	0.335	0.464	0.056	0.316	0.410	0.345
Frog2A	0.624	0.369	0.520	0.340	0.477	0.036	0.325	0.415	0.343
Squid2	0.633	0.381	0.399	0.390	0.446	0.036	0.040	0.370	0.360
PFrog1	0.268	0.209	0.229	0.173	0.204	0.118	0.167	0.205	0.192
PFrog2	0.260	0.215	0.245	0.154	0.206	0.095	0.189	0.203	0.181
Tim	0.946	0.539	0.537	0.408	0.532	0.006	0.017	0.491	0.475
Peek	0.909	0.534	0.544	0.464	0.512	0.019	0.046	0.492	0.482
Power	0.918	0.541	0.539	0.471	0.504	0.022	0.026	0.492	0.488

Regarding *Energy* the algorithms rank not very different. The differences are somewhat less pronounced.

Let's look at *eFootprint* normalized by *Knuthsort* at randomly permuted data:

Table 5.19: bcdFootprint of Timsort alternatives normalized by Knuthsort with random data

	permut	asclocal	desclocal	ascglobal	descglobal	ascall	descall	ALL	ASC
Knuth	1.000	0.479	0.643	0.461	0.459	0.145	0.285	0.559	0.521
Omit	1.029	0.465	0.598	0.458	0.450	0.010	0.250	0.536	0.490
Octo	0.813	0.469	0.476	0.459	0.459	0.010	0.015	0.439	0.438
Gecko1	0.602	0.405	0.393	0.321	0.323	0.124	0.109	0.360	0.363
Frog1	0.599	0.363	0.427	0.314	0.328	0.067	0.144	0.355	0.336
Frog1A	0.595	0.354	0.436	0.312	0.336	0.046	0.158	0.354	0.327
Squid1	0.594	0.350	0.361	0.320	0.332	0.047	0.050	0.331	0.328
Frog2	0.349	0.217	0.291	0.192	0.265	0.032	0.180	0.234	0.197
Frog2A	0.357	0.211	0.297	0.195	0.273	0.021	0.186	0.237	0.196
Squid2	0.362	0.217	0.228	0.223	0.255	0.021	0.023	0.211	0.206
PFrog1	0.201	0.157	0.172	0.130	0.153	0.088	0.125	0.153	0.144
PFrog2	0.156	0.129	0.147	0.093	0.123	0.057	0.114	0.122	0.109
Tim	0.709	0.404	0.403	0.306	0.399	0.004	0.013	0.368	0.356
Peek	0.682	0.401	0.408	0.348	0.384	0.014	0.034	0.369	0.361
Power	0.689	0.406	0.405	0.353	0.378	0.016	0.020	0.369	0.366

Regarding *eFootprints* the differences are more pronounced, *Timsort*, *Peeksort* and *Powersort* win over *Knuthsort*, but not over the *greeNsort®* algorithms. *Frogsort2* emerges as a clear winner, particularly if parallelized.

A final remark on *Glidesort* (Peters (2023a);Peters (2023a)) is justified: while *Octosort* and *Squidsort* lazily delay enforcing a particular order, *Glidesort* lazily delays sorting at all (in the hope to use partitioning instead of merging to benefit from ties). This cleverly makes *Glidesort* adaptive to presorting *and* ties, hats off. Like *Timsort Powersort* , *Glidesort* is implemented bottom-up, but we could do this recursively top-down and enhance *Octosort* and *Squidsort* to adapt to ties. However, it should be noted, that delaying sorting altogether is tuning that costs a trade-off: such an algorithm goes through recursion once to detect presorting, and without presorting, it goes through recursion a second time. This implies that unsorted chunks are not sorted when they touch the cache for the first time, rather they need to be touched a second time. This is the same phenomenon as in *Ducksort*, that making the algorithm adaptive to both, is no longer optimal for the case of single-adaptivity (*Zacksort* or *Zucksort*).

5.13 Algorithmic variations

In this section interesting variations of the *Symmetric-merging* algorithms are described:

- in-situ versus ex-situ context
- equally-sized versus size-varying elements
- serial versus parallel implementation

5.13.1 In-situ ex-situ

greeNsort® has implemented all algorithms *in-situ* and *ex-situ*. Assume the data resides in memory D. *ex-situ* allocates fresh memory A for data and B for buffer, copies data from D to A, sorts the data in A not modifying the original data in D (and finally copies back to D just for verification in the test-bed). *in-situ* uses the original memory D for sorting the data and only allocates the necessary buffer B. For the algorithms using 100% buffer this is straightforward. For those using a $p < 1$ buffer fraction, in-situ works as follows:

- calculate how many elements can be sorted in original memory $N_D = \lfloor N/(1+p) \rfloor$
- calculate how many elements must be sorted in buffer memory $N_B = N - N_D$
- calculate the buffer size $b = N_B * (1+p)$ and allocate
- do the gapped setup of N_B to B
- sort N_D in D and N_B in B
- merge N_B with N_D back to D
- deallocate B

Note that all *greeNsort®* measurements are conservatively done *in-situ*, where $p < 1$ gives an imbalanced final merge, hence according to prio-art this gives *Knuthsort* an advantage over *Frogsort* (however, not necessarily, due to branch-prediction).

Note further that it is tempting to join the setup-recursion and the merging-recursion in one recursion, however, the *greeNsort®* implementations don't do this, because this would actually increase the amount of memory required during the sort, for example from 150% to 250%.

5.13.2 Low-memory

Low-memory algorithms are not the focus of *greeNsort®*. *greeNsort®* suspects that in-place merging algorithms are academically overrated. However, due to their small *%RAM* needs, they have a potential for competitive *Footprints*. Therefore a couple of low-memory algorithms is compared, mostly taken from Andrej Astrelin's code[4], namely a simple slow in-place-mergesort (*IMergesort*, no buffer), his fast in-place *Grailsort* (no buffer) and his slightly faster *Sqrtsort* ($\mathcal{O}\left(\sqrt{N}\right)$ buffer). *greeNsort®* has complemented this with own implementation of two algorithms (*Walksort* and *Jumpsort*) that require $\mathcal{O}\left(\sqrt{N}\right)$ buffer and effective random-block-access. For me designing Walksort was pretty straightforward, hence it might not be very innovative, however it is not too complicated and faster than Astrelin's Sqrtsort. Jumpsort is a variation that uses relocation-moves for distance minimization. Because Walksort and Jumpsort have very similar performance on current hardware, I excluded Jumpsort from measurement here.

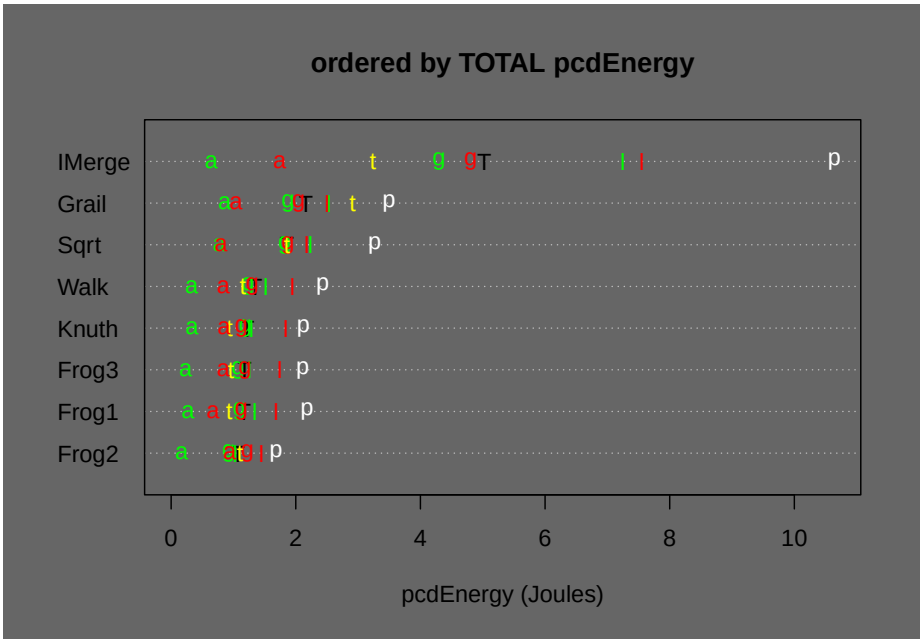

Figure 5.17: Medians of In-place-Mergesort alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

[4]https://github.com/Mrrl

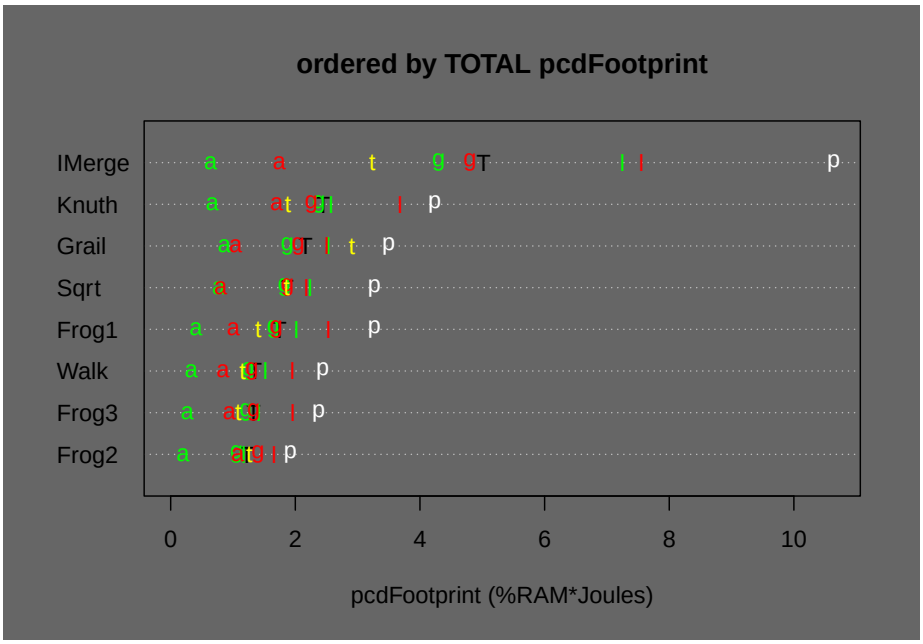

Figure 5.18: Medians of In-place-Mergesort alternatives ordered by TOTAL Footprint. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

Table 5.20: Walksort / Knurthsort (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	0.5	1.11	1.10	0.55	0.13	0.000	0e+00	0
ascall	0.5	1.04	0.99	0.50	0.00	0.391	1e-04	0
descall	0.5	0.95	0.99	0.49	-0.01	0.000	0e+00	0
ascglobal	0.5	1.09	1.04	0.52	0.04	0.000	0e+00	0
descglobal	0.5	1.16	1.15	0.58	0.17	0.000	0e+00	0
asclocal	0.5	1.22	1.18	0.59	0.23	0.000	0e+00	0
desclocal	0.5	1.07	1.06	0.53	0.11	0.000	0e+00	0
tielog2	0.5	1.24	1.23	0.62	0.21	0.000	0e+00	0
permut	0.5	1.17	1.15	0.58	0.31	0.000	0e+00	0

Table 5.21: Walksort / Sqrtsort (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	1	0.76	0.73	0.73	-0.51	0	0	0
ascall	1	0.46	0.43	0.43	-0.44	0	0	0
descall	1	1.14	1.05	1.05	0.04	0	0	0
ascglobal	1	0.73	0.68	0.68	-0.59	0	0	0
descglobal	1	0.73	0.69	0.69	-0.58	0	0	0
asclocal	1	0.71	0.68	0.68	-0.72	0	0	0
desclocal	1	0.92	0.90	0.90	-0.23	0	0	0
tielog2	1	0.65	0.62	0.62	-0.71	0	0	0
permut	1	0.77	0.74	0.75	-0.83	0	0	0

Compared to *Knuthsort*, *Walksort* needs only 10% more *Energy* and only 55% *eFootprint*. Compared to *Sqrtsort*, *Walksort* needs only about 75% *RUNtime*, *Energy* and *eFootprint*.

5.13.3 Stabilized Quicksorts

The prior art knows some workarounds that allow to use *Quicksorts* as *stable* sorting algorithms, also for *size-varying elements*. The most simple, popular and powerful way is *Indirect Quicksort* which can address both goals. *RQuicksort2* uses *Quicksort2* to sort *pointers* to doubles instead of sorting doubles directly, in case of tied keys the comparison function compares the pointers which always breaks the ties. Since such *Indirect Quicksort* has no ties anymore, *Z:cksort* would have no advantage. *RQuicksort2* is a variant that uses the block-tuning of Edelkamp and Weiß (2016),Edelkamp and Weiss (2016), but that actually slows down sorting dramatically. The take-away of RQuicksort2 is, that it needs $\mathcal{O}(N)$ extra memory for pointers and that it is slow, because the pointer indirection causes heavy random-access to the data.

Faster than sorting pointers *to* elements is sorting elements *and* position-indicating integers (*SQuicksort2*), this avoids random-access, but has two disadvantages:

- with SWAPing elements it can no longer sort *size-varying elements*
- standard code and APIs cannot handle the necessary SWAPs for a separate integer vector, hence either special code is needed, or special objects need to be composed that contain elements and integers

Again, block-tuning is (somewhat) harmful, hence I did not include RQuicksort2B and SQuicksort2B in the measurement.

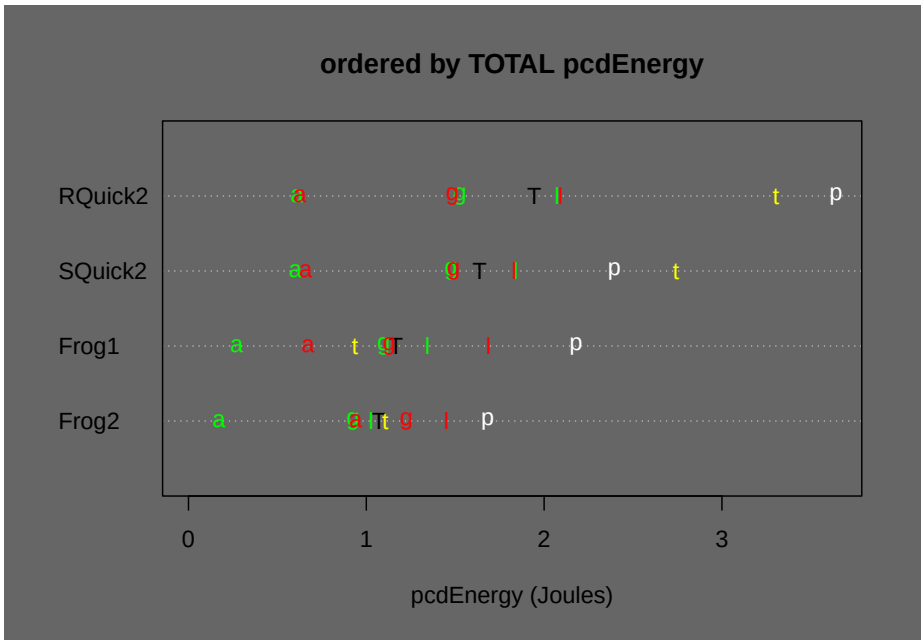

Figure 5.19: Medians of stabilizd Quicksorts and some stable alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

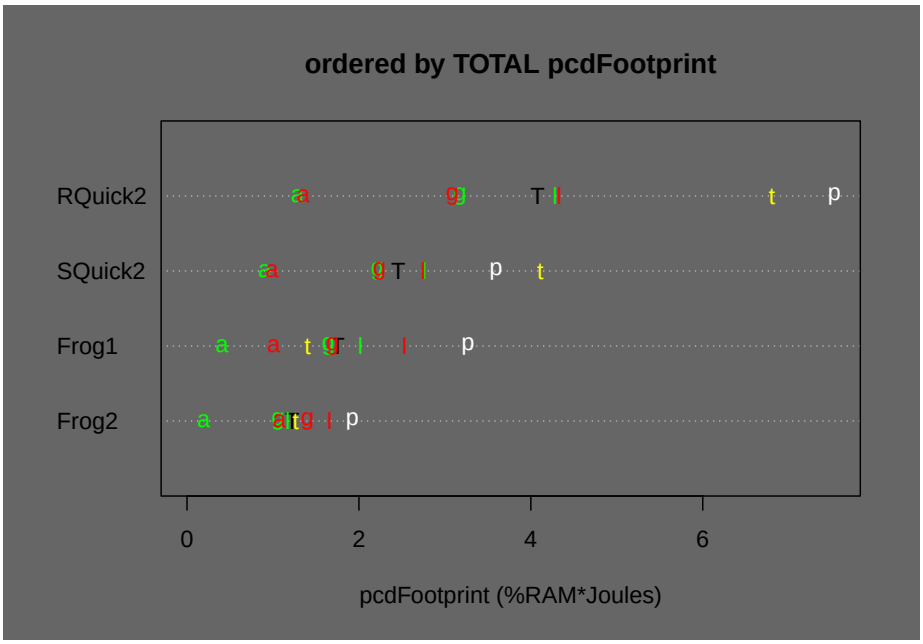

Figure 5.20: Medians of stabilizd Quicksorts and some stable alternatives ordered by TOTAL Footprint. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

Table 5.22: Frogsort2 / SQuicksort2 (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	0.76	0.60	0.65	0.50	-0.57	0	0	0
ascall	0.76	0.25	0.28	0.22	-0.43	0	0	0
descall	0.76	1.32	1.43	1.09	0.28	0	0	0
ascglobal	0.76	0.59	0.63	0.48	-0.55	0	0	0
descglobal	0.76	0.76	0.82	0.63	-0.27	0	0	0
asclocal	0.76	0.51	0.56	0.42	-0.82	0	0	0
desclocal	0.76	0.74	0.79	0.60	-0.38	0	0	0
tielog2	0.76	0.36	0.40	0.31	-1.63	0	0	0
permut	0.76	0.66	0.70	0.54	-0.71	0	0	0

Stabilized *Indirect Quicksort* cannot compete with *Frogsort*: *Frogsort2* needs only 60% *RUNtime* and 64% *Energy* compared to *SQuicksort2*.

5.13.4 Size-varying algos

```
innovation("DSDS","C","Direct Sorting Different Size")
innovation("DSNS","B","Direct Sorting Null-terminated Strings")
innovation("DSPE","B","Direct Sorting Pointered Elements")
dependency("DSNS","DSDS")
dependency("DSPE","DSDS")
innovation("VKnuthsort","A","Varying-size Knuthsort")
dependency("VKnuthsort","DSNS")
innovation("VFrogsort1","A","Varying-size Frogsort1")
dependency("VFrogsort1","DSNS")
dependency("VFrogsort1","Frogsort1")
```

As mentioned above, *Indirect Sorts* using pointers to elements are popular in the prior art for sorting strings as an example of sorting *size-varying elements*. *UKnuthsort* is an indirect stable *Knuthsort*, *UZacksort*, *UZacksortB* (block-tuned) are indirect *Zacksorts* that sort strings (not stable), *WQuicksort2* and *WQuicksort2B* (block-tuned) are stabilized indirect Quicksorts that sort strings and break ties by comparing pointers.

Such *Indirect Sorts* have the disadvantage to cause random access, they are obviously not efficient today's machines with their deep cache-hierarchies. *greeNsort®* explored a different approach to deal with memory distances that cause non-constant access costs: *C-level innovation – (D)irect (S)orting (D)ifferent (S)ize (DSDS)*.

One quite generic way to realize *DSDS* is *B-level innovation – (D)irect (S)orting (P)ointered (E)lements (DSPE)*. *DSPE* uses pointers to elements and does Split&Merge on pointers *and* elements: this avoids random access and reduces distance (is much more "cache-friendly" to use prior art-lingo). Note that splitting pointers gives balanced numbers of strings, not necessarily balanced size. So far so good. Note that the pointers cost extra memory. Is it possible to get rid of them?

Null-terminated strings have been cursed as "The Most Expensive One-byte Mistake" by Kamp (2011). Much of the critique is justified, but one aspect of null-terminated strings was a stroke of genius: searching for null-terminators in a series of strings gives a cheap balanced split by size (not necessarily a balanced number of strings). Splitting with null-terminators means searching for null-terminators from an desired split-position. If searching in one direction does not give a split (because search started within the last element of the first direction of search) then it must be searched in the other direction. This I call *B-level innovation – (D)irect (S)orting (N)ull-terminated (S)trings (DSNS)*. The *greeNsort®* test-bed implements *VKnuthsort*, a *DSNS Knuthsort*, and *VFrogsort1*, a *DSNS Frogsort1*. Let's see how they perform:

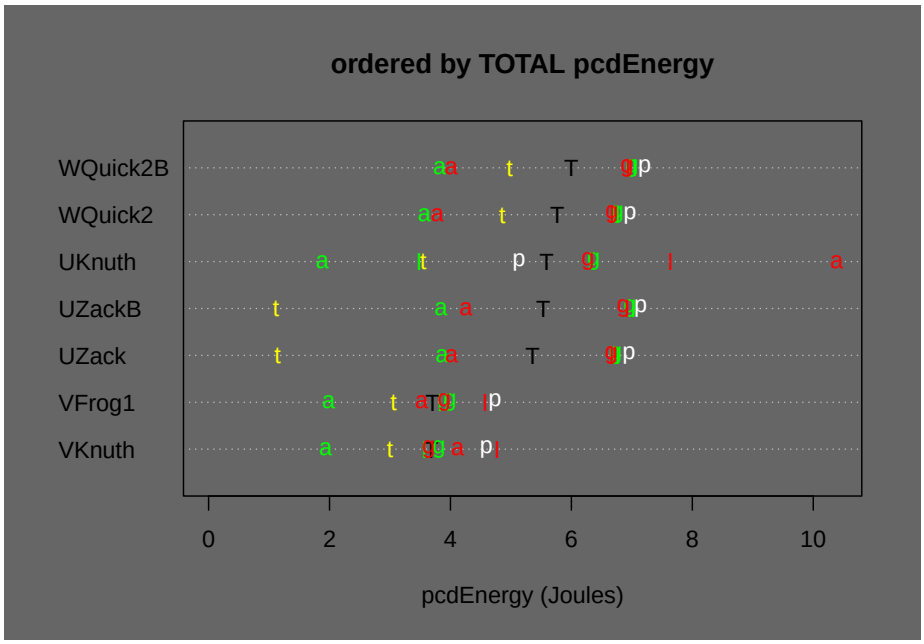

Figure 5.21: Medians of string Quicksorts and some stable alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

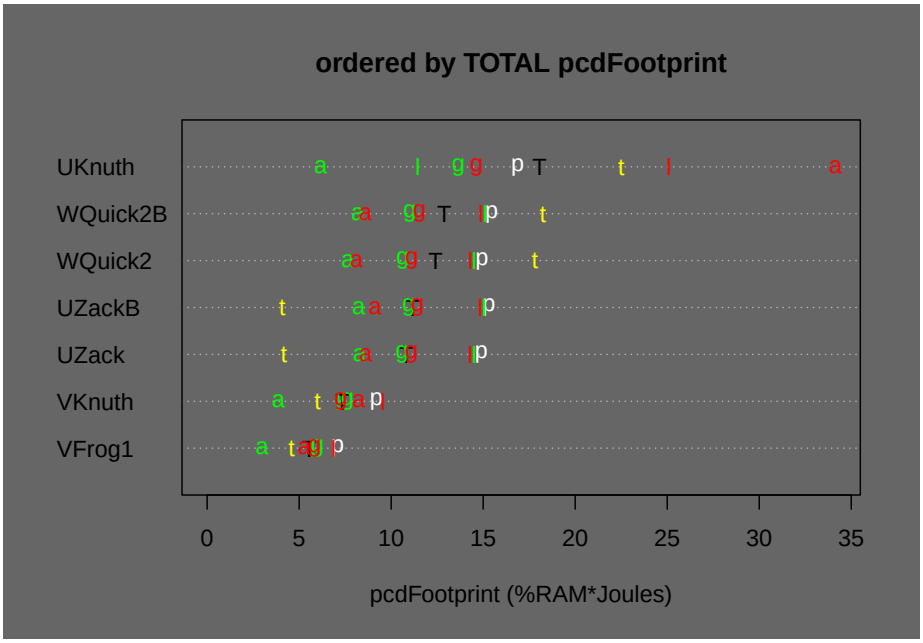

Figure 5.22: Medians of string Quicksorts and some stable alternatives ordered by TOTAL Footprint. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

- *UKnuthsort* needs less *Energy* than *WQuicksort2* but the latter has the better *eFootprint*
- when sacrificing stability, *UZacksort* needs less energy than *WQuicksort2*
- block-tuning does not pay off in *UZacksortB* and *WQuicksort2B*
- direct sorting (*VKnuthsort*, *VFrogsort1*) needs less *Energy* than indirect sorting (*UZacksort*, *UKnuthsort*)
- only for data with ties indirect unstable *UZacksort* needs less *Energy*, for sorting natural strings having many ties UZacksort might be the better choice if stability is not needed

- for short strings direct sorting has better *Footprint* than indirect sorting, for long strings Footprint of direct is worse
- *VFrogsort1* has better Footprint than *VKnuthsort*

A stable size-varying partition&pool sort promises to combine the speed and energy-efficiency of direct sorting with early-termination on ties. *DSNS* versions of *Kiwisort* and *Swansort* have not yet been implemented; since they need 100% buffer it is an open question whether this is better than *VFrogsort1*.

Table 5.23: VKnuthsort / UKnuthsort (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	0.59	0.66	0.66	0.41	-1.91	0	0	0
ascall	0.61	1.03	1.03	0.63	0.05	0	0	0
descall	0.61	0.40	0.40	0.24	-6.27	0	0	0
ascglobal	0.93	0.62	0.60	0.56	-2.56	0	0	0
descglobal	0.86	0.61	0.58	0.50	-2.63	0	0	0
asclocal	0.61	1.02	1.03	0.62	0.09	0	0	0
desclocal	0.61	0.63	0.63	0.38	-2.86	0	0	0
tielog2	0.32	0.85	0.85	0.27	-0.55	0	0	0
permut	0.61	0.87	0.90	0.54	-0.54	0	0	0

Table 5.24: VFrogsort1 / UKnuthsort (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	0.44	0.67	0.66	0.31	-1.89	0	0	0
ascall	0.46	1.12	1.06	0.48	0.10	0	0	0
descall	0.46	0.34	0.34	0.15	-6.87	0	0	0
ascglobal	0.70	0.65	0.63	0.44	-2.37	0	0	0
descglobal	0.64	0.65	0.62	0.40	-2.37	0	0	0
asclocal	0.46	1.11	1.10	0.50	0.35	0	0	0
desclocal	0.46	0.61	0.60	0.27	-3.06	0	0	0
tielog2	0.24	0.87	0.86	0.20	-0.49	0	0	0
permut	0.46	0.90	0.92	0.42	-0.40	0	0	0

Table 5.25: VFrogsort1 / WQuicksort2 (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	0.68	0.64	0.64	0.45	-2.06	0	0	0
ascall	0.70	0.59	0.56	0.39	-1.58	0	0	0
descall	0.70	0.94	0.93	0.65	-0.26	0	0	0
ascglobal	0.95	0.58	0.59	0.56	-2.76	0	0	0
descglobal	0.90	0.58	0.58	0.53	-2.77	0	0	0
asclocal	0.70	0.56	0.57	0.40	-2.94	0	0	0
desclocal	0.70	0.67	0.69	0.48	-2.10	0	0	0
tielog2	0.41	0.63	0.63	0.26	-1.80	0	0	0
permut	0.70	0.67	0.68	0.48	-2.23	0	0	0

Table 5.26: VFrogsort1 / UZacksort (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	0.68	0.69	0.69	0.51	-1.66	0	0	0
ascall	0.70	0.54	0.51	0.36	-1.88	0	0	0
descall	0.70	0.88	0.88	0.61	-0.50	0	0	0
ascglobal	0.95	0.58	0.59	0.57	-2.74	0	0	0
descglobal	0.90	0.59	0.59	0.53	-2.76	0	0	0
asclocal	0.70	0.56	0.57	0.40	-2.94	0	0	0
desclocal	0.70	0.67	0.69	0.48	-2.09	0	0	0
tielog2	0.41	2.68	2.68	1.10	1.92	0	0	0
permut	0.70	0.67	0.68	0.48	-2.22	0	0	0

For stable string-sorting, *VKnuthsort* needs only 66% of the *Energy* of *UKnuthsort*, and *VFrogsort1* needs only 75% of the *eFootprint* of *VKnuthsort* which cumulates to 31% of the *Footprint* of *UKnuthsort* and still 45% *Footprint* of *WQuicksort2*. If stability is not needed, *UZacksort* is more efficient than *WQuicksort2*, but *VFrogsort1* still needs only 70% of the *Energy* of UZacksort. Only for for strongly tied data UZacksort is by factor 2 or 3 more efficient.

5.14 Split&Merge conclusion

greeNsort® has analyzed the prior art on in-memory sorting in the *Split&Merge* paradigm. Practically relevant prior art algorithms in contiguous space are

- nocopy-merging with 100% buffer (*Knuthsort*: alternates merging between distant memory regions A and B)
- natural-merging with 50% buffer (*Timsort*: adaptive but copies out to B and merges back to A)

and both use $\mathcal{O}(N \log N)$ global distance moves. Algorithms without global moves are known, but little used due to their disadvantages: in-place-mergesorts are notoriously slow and cache-oblivious algorithms (*Funnelsort*[5]) are notoriously complicated, particularly for N that is not a power of two.

greeNsort® questioned and dropped several assumptions of the prior art and introduced new methods that allow new algorithms with better trade-offs:

- *Omitsort* combines nocopy-merging using 100% buffer with *Data-driven-location* and achieves a linear best-case for presorted data
- *Octosort* combines *Omitsort* further with *Data-driven-order* and is adaptive to pre-sorted and reverse-sorted data (like *Timsort*) but is efficient and can be fully parallel (unlike *Timsort*)
- a chain of seemingly inefficient innovations (*Gapped-merging*, naive *Buffer-merging*) is finally developed into a new efficient sorting paradigm: *Symmetric-merging* that operates on contiguous data, minimizes distance and needs not more than 50% buffer.
- *Frogsort0* and *Frogsort1* use 50% buffer and save 50% work on presorted data
- *Frogsort2 (p2)* and *Frogsort3 (p3)* split or share less than 50% buffer (much less) and are sometimes even faster (much faster)
- *Frogsort6 (p2,p3)* combines Frogsort1, Frogsort2 and Frogsort3
- *Geckosort* is a symmetric alternative to *Frogsort* that saves 25% work on pre-sorted and 25% on reverse-sorted data
- *Squidsort* combines *Symmetric-merging* with *Data-driven-order* (and is much more efficient than *Timsort*)
- *VKnuthsort* (100% buffer) and *VFrogsort1* (50% buffer) show that direct sorting of size-varying elements can be faster than the usual indirect pointer-sorting with Quicksorts (*WQuicksort2*, *UZacksort*) or Mergesorts (*UKnuthsort*)

[5] the simpler *Lazy-Funnelsort* drops the contiguous Van-Emde-Boas memory layout

The empirical results regarding *Energy* and *eFootprint* from the *greeNsort®* test-bed on the TOTAL mix of input patterns are:

- *Omitsort* using 100% buffer needs 97% Energy of *Knuthsort* (for presorted data it is 12%)
- *Octosort* using 100% buffer needs 90% Energy of *Knuthsort* (for presorted and reverse-sorted data it is 14% and 7%)
- *Frogsort1* using 50% buffer needs 96% of the Energy of *Knuthsort* and 72% of the eFootprint
- *Geckosort1* using 50% buffer needs 92% of the Energy of *Knuthsort* and 69% of the eFootprint
- *Frogsort2*(1/7) using 14% buffer needs 87% of the Energy of *Knuthsort* and 50% of the eFootprint
- *Frogsort3*(1/8) using 12.5% buffer needs 97% of the Energy of *Knuthsort* and 54% of the eFootprint
- *Squidsort1* using 50% buffer needs 90% of the Energy of *Timsort* and 90% of the eFootprint
- *Squidsort2*(1/7) using 14% buffer needs 83% of the Energy of *Knuthsort* and 63% of the eFootprint
- *Walksort* using $\mathcal{O}\left(\sqrt{(N)}\right)$ buffer needs only 73% Energy of Astrelin's *Sqrtsort*, which in turn needs 86% of his *Grailsort*, which in turn needs only 43% of a simple *IMergesort*.
- *VKnuthsort* using 100% buffer needs only 66% of the Energy of *UKnuthsorts* (and often much less Footprint, here 41%)
- *VFrogsort1* using 50% buffer needs only 66% of the Energy of *UKnuthsorts* (and often much less Footprint, here 31%)

Comparing *Split&Merge* with *Quicksorts*:

- unstable *Ducksort* using 0% buffer needs 95% of the Energy (and 83% eFootprint) of stable *Frogsort2* using 14% buffer
- stable *Frogsort2* using 14% buffer needs 64% of the Energy of the stabilized indirect *WQuicksort2s* using 64bit pointers
- the size-varying *VFrogsort1* splitting on null-terminators in strings and using 50% buffer needs 66% of the Energy of indirect sorts using 64bit pointers (and often less Footprint)

Given that *Geckosort1* performed better than *Frogsort1*: *Geckosort2*, *Geckosort3* and *Geckosort6* are promising but not yet implemented.

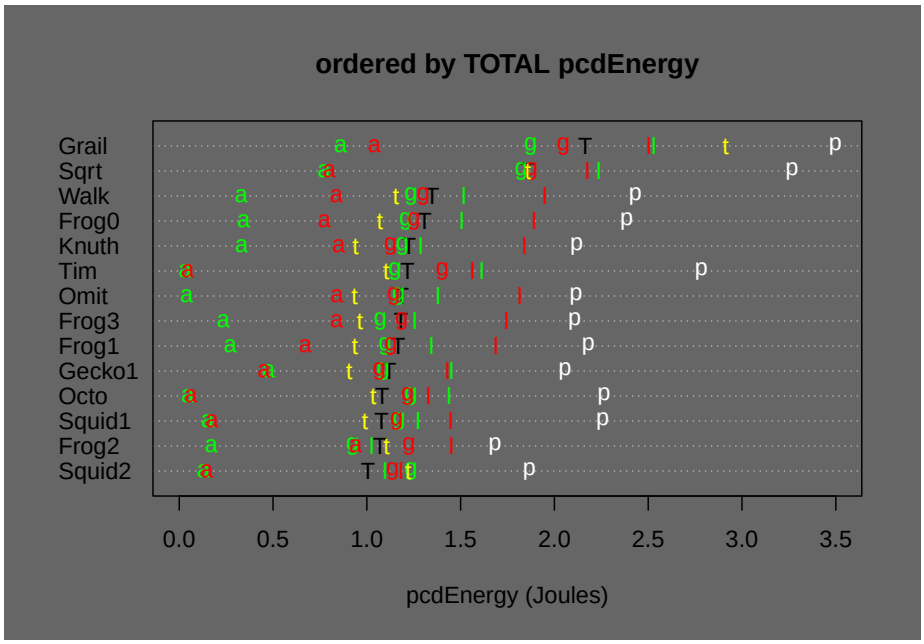

Figure 5.23: Medians of Mergesort alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

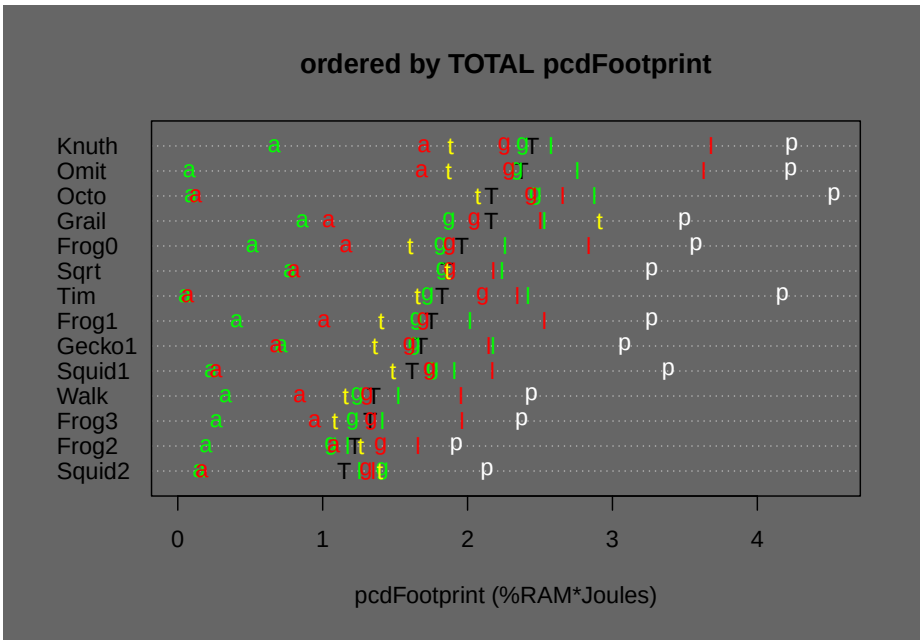

Figure 5.24: Medians of Mergesort alternatives ordered by TOTAL eFootprint. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

Table 5.27: 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

To illustrate the height of the *greeNsort®* Split&Merge inventions, I show the Innovation-lineage of *Squidsort2*: all innovations on which Squidsort2 depends form a *(D)irected (A)cyclic (G)raph (DAG)*:

```
if (allinno)
  depplot(parents("Squidsort2"))
```

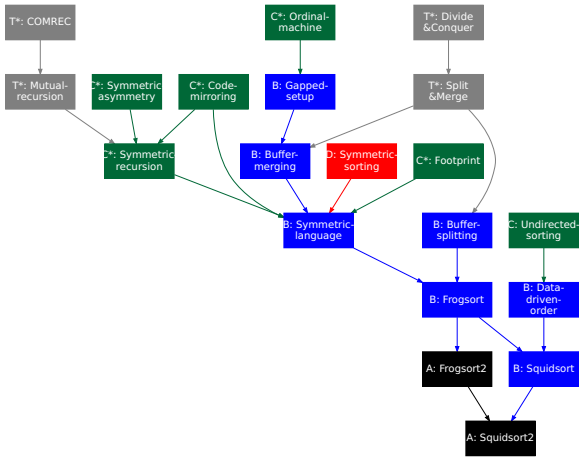

Figure 5.25: Innovation-lineage of Squidsort2

Chapter 6

Partition&Pool Algorithms

```
innovation("Partition&Pool","T*","Recursive partition and pool paradigm")
dependency("Partition&Pool","Divide&Conquer")
innovation("Wing-partitioning","B*","Stable partitioning without counting")
dependency("Wing-partitioning","Partition&Pool")
dependency("Wing-partitioning","Symmetric-sorting")
innovation("Kiwisort","A","Stable Partition&Pool sort 100% buffer")
dependency("Kiwisort","Wing-partitioning")
innovation("Buffer-partitioning","B*","partitioning data and buffer")
dependency("Buffer-partitioning","Partition&Pool")
dependency("Buffer-partitioning","Ordinal-machine")
innovation("Symmetric-partitioning","B*","symmetric partitioning data and buffer")
dependency("Symmetric-partitioning","Wing-partitioning")
dependency("Symmetric-partitioning","Buffer-partitioning")
dependency("Symmetric-partitioning","FLIP")
dependency("Symmetric-partitioning","DIET")
innovation("Gapped-wrapup","B*","collecting the gapped results")
dependency("Gapped-wrapup","Symmetric-partitioning")
innovation("Swansort","A","distance-reducing stable P&P sort 100% buffer")
dependency("Swansort","Gapped-wrapup")
dependency("Swansort","Symmetric-partitioning")
innovation("Storksort","A","distance-reducing stable P&P sort 50% buffer")
dependency("Storksort","Symmetric-partitioning")
dependency("Storksort","Footprint")
```

For generic sorting with $\mathcal{O}(N \log n)$ worse case, *Split&Merge* algorithms need not more than one pass over the data per recursion level. *Partition&Pool* usually need more passes. One reason is that determining a median for balanced partitioning is costly, even when using an approximate median of medians as in the BFPRT-select algorithm of Blum et al. (1973). Using one (or more) random

pivots avoids these costs, the price for this is a missing $\mathcal{O}(N \log n)$ worse case guarantee. Another reason for multiple passes in generic stable partitioning with 100% buffer is that two passes are needed, one for determining the size of the partitions and one for actually distributing the elements to the partitions in a stable manner. *Quicksort* avoids the need to know the beginning of the second partition by writing the second partition in reverse direction beginning with the outer right position. *Partition&Pool* is doubly clever by also avoiding the need for buffer, and the price for this is missing stability and missing generality (SWAPS require equal size of elements).

Writing two partitions from the left and right outer ends is also possible with 100% buffer, this allows size-varying elements but is still not stable. However, under the *greeNsort®* definition of *Symmetric-sorting* it is possible to make this stable by tracking the number of direction reversals. Once the recursion reaches a (completely tied) leaf, stability can be achieved by reversing the order of ties if and only if the number of reversals was odd. This method I call *Wing-partitioning*. The resulting sort when using 100% separate buffer memory (like in *Knuthsort*) and the *B-level innovation – (F)ast (L)oops (I)n (P)artitioning (FLIP)* and *B-level innovation – (D)istinct (I)dentification for (E)arly (T)ermination (DIET)* methods I call *Kiwisort*. *Kiwisort* features early termination on ties and a $\mathcal{O}(N \log n)$ probabilistic worse case, like *Z:cksort*.

The section on *Symmetric-merging* was already written to also cover *Symmetric-partitioning*. Instead of a separate buffer region, it is possible to also use contiguous data-buffer-memory and do Partition&Pool of data and 100% buffer. The two partitions are written in-place, one from the read direction (e.g. from left) and the other from the opposite direction (e.g. from right). This creates two partitions with 100% buffer in between, which can be partitioned recursively. Once the data is fully partitioned at the leafs of the recursion tree, there are gaps of buffer between the data. Hence a final pass over the data is needed to collect the data into contiguous memory, this method I call *Gapped-wrapup*. The resulting sort I call *Swansort*. Note that the in-situ implementation measured here requires a deterministic median split at the top level in order to properly distribute 50% of the data to the 100% freshly allocated temporary memory, this makes the in-situ implementation somewhat slower than the ex-situ implementation.

It is possible to modify *Swansort* to run with only 50% buffer. How? By re-introducing a counting pass! In the counting pass the algorithm determines the sizes of the two partitions. Once the partition size is known, the bigger partition is written in-place in the read-direction, the smaller partition is written from the opposite side. Since the smaller partition is $\leq 50\%$ it will always fit into the 50% buffer. The result looks similar to *Swansort* with buffer between two partitions, and can be easily partitioned recursively. Once the data is fully partitioned at the leafs of the recursion tree, there are not only gaps of buffer between the data, the sorted pieces are also not in sorting order. But each leaf of the recursion tree *knows* to which position in sorted order its data belongs and can send it to there. Writing the data directly into some result memory is not an

attractive option, because the required existence of this memory would actually turn this 150% memory algorithm to a 250% memory algorithm. However, the algorithm could send the data over a wire. There are two options how to do this. 1) in a serial execution, the data could simply be sent in serial order. 2) in a parallel execution, the data could be sent together with its designated position. The resulting algorithm I call *Storksort*. Storksort could make sense as a cloud-sorting service which receives data and sends it back sorted.

In conclusion: for sustainable stable binary sorting in contiguous memory, *Partition&Pool* is a less efficient paradigm than *Split&Merge*, with the important exception of early termination on ties. *Partition&Pool* is much more efficient, if one gives up the simplicity of binary sorting for k-ary sorting (which can better amortize the high cost per level) and/or the locality of contiguous memory for block-managed memory (which allows to skip a counting-pass). The *(I)n-place (P)arallel (S)uper (S)calar (S)ample(So)rt (IPS4o)* in block-managed-memory with $\sqrt{N}$ buffer by Axtmann et al. (2017) is an excellent k-ary algorithm, it has the following pros and cons:

- it has high implementation complexity
- it is not stable
- it requires random block-access
- it cannot sort variable size (except for indirect sorting which requires random access)
- the serial version is less more efficient than *Pdqsort* and *Ducksort*
- the parallel version is more efficient than *Pdqsort*, *Ducksort*
- IS4o is adaptive to presorted (and reverse-sorted), but IPS4o is much less adaptive

It would be interesting

- to see if *IPS4o* could benefit from *Z:cksort*'s *FLIP* method
- to stabilize *IPS4o* under the definition of *Symmetric-sorting* using *greeN-sort®* methods
- to see if *IPS4o* could benefit from the $\sqrt{N}$ block management of *Walksort* and *Jumpsort*

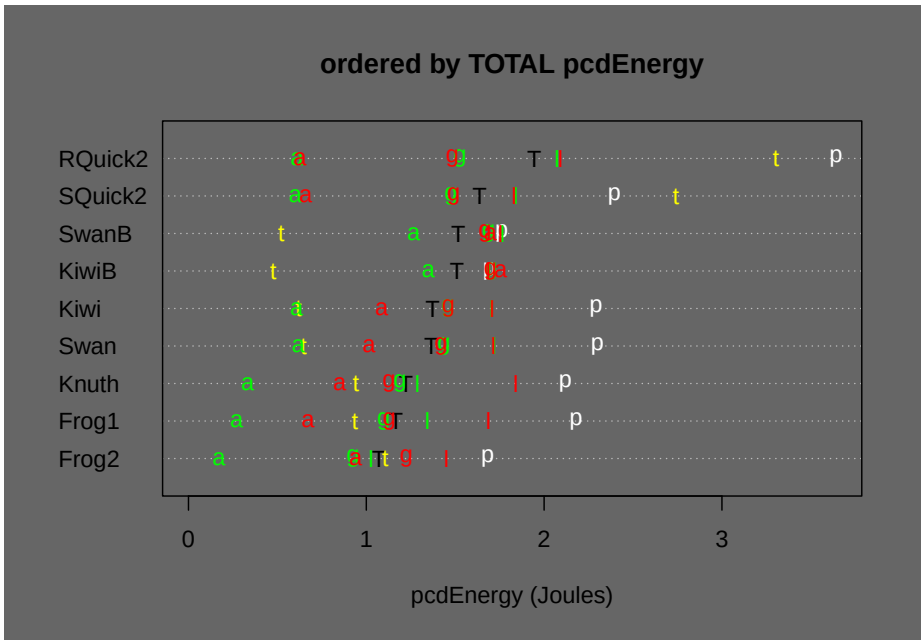

Figure 6.1: Medians of stable partitioning alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

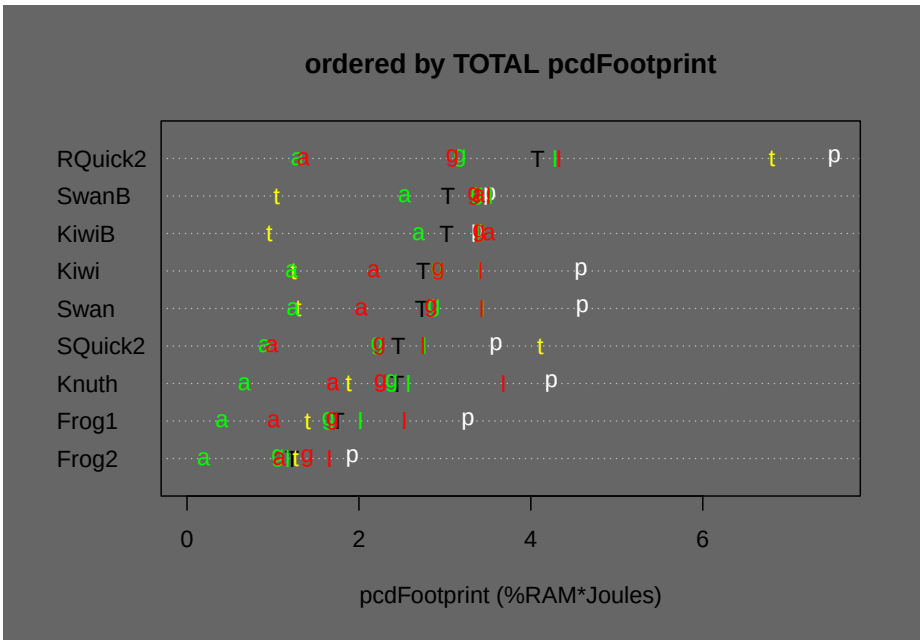

Figure 6.2: Medians of stable partitioning alternatives ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

Table 6.1: Swansort / 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	1	1.10	1.12	1.12	0.14	0	0	0
ascall	1	1.85	1.85	1.85	0.28	0	0	0
descall	1	1.15	1.19	1.19	0.16	0	0	0
ascglobal	1	1.20	1.21	1.21	0.25	0	0	0
descglobal	1	1.23	1.26	1.26	0.29	0	0	0
asclocal	1	1.35	1.33	1.33	0.43	0	0	0
desclocal	1	0.94	0.93	0.93	-0.12	0	0	0
tielog2	1	0.66	0.69	0.69	-0.29	0	0	0
permut	1	1.07	1.09	1.09	0.18	0	0	0

To illustrate the height of the *greeNsort®* partition&pool inventions, I show the innovation-lineage of *Storksort*: all innovations on which Storksort depends form a *(D)irected (A)cyclic (G)raph (DAG)*:

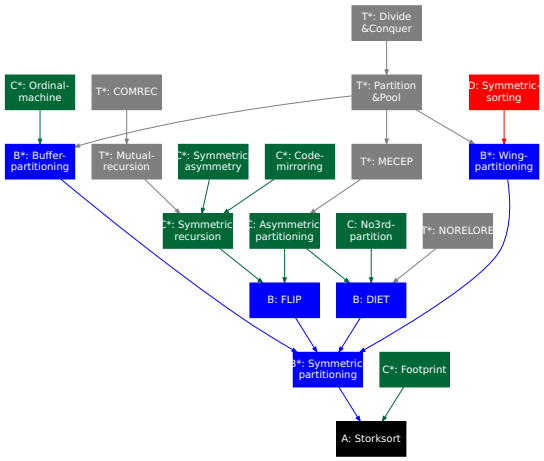

Figure 6.3: Innovation-lineage of Storksort

Chapter 7

Parallel sorting

7.1 Parallel merging

```
innovation("lean-parallel-merging","T","lean parallel merging")
innovation("PKnuthsort","E","parallel Knuthsort")
dependency("PKnuthsort","Knuthsort")
dependency("PKnuthsort","lean-parallel-merging")
```

The *greeNsort*® test-bed uses the POSIX pthread library. The APIs of the parallel sorting algorithms require specification of the number of allowed threads as a parameter. This number is split down the recursion tree. The first step to parallelizing a *Mergesort* is running the recursive branches in parallel.

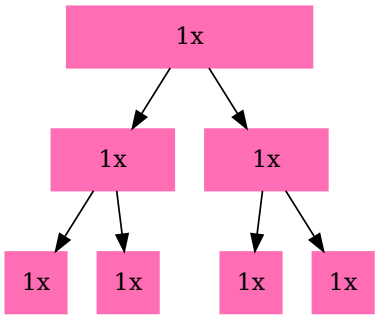

Figure 7.1: Number of threads per node in parallel splitting

But that is not enough: with parallel branching the maximum parallelism is only achieved at the leaves of the tree (4 in the example). Serial merging

prevents proper utilization of cores near the root of the tree. Merging can also be parallelized: by recursively searching for the median in the two sorted sequences to be merged (Cormen et al. (2009)[1]):

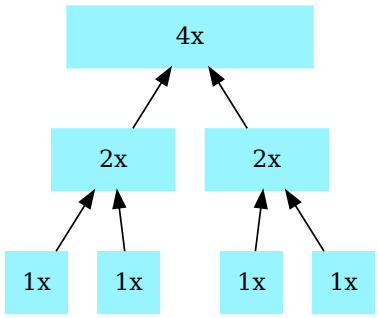

Figure 7.2: Number of threads per node in parallel splitting and parallel merging using recursive medians

This is still not optimal, because the parallelism of the merges is assumed to be a power of two. If there are 6 cores available, this implies that either only 4 are used or that 8 threads are used that compete for 6 cores. This implies context-switching and competing access into the cache-hierarchy. For the sake of *robustness greeNsort®* prefers to not make the assumption that using more threads than requested is harmless and strives for a parallel merge that will use exactly the requested number of cores by splitting into the appropriate number of percentiles.

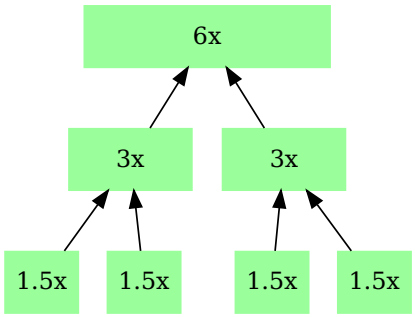

Figure 7.3: Number of threads per node in parallel splitting and parallel merging using percentiles

[1]Cormen et. al. describe an inefficient parallel merge which copies to buffer and merges back, however, marrying this with nocopy-merging of Sedgewick (1998) is straightforward

The theoretically most efficient solution is a recursive search procedure that operates not only on the two input vectors but also on a vector of requested percentiles. However, this requires extra dynamically allocated memory structures and and a certain implementation complexity. A much simpler and distance-friendly solution is to loop over the requested percentiles and search in turn for the next percentile in the remaining range. I call this *lean-parallel-merging*.

7.1.1 Parallel Knuthsort

Using *lean-parallel-merging* method, the reference algorithm *Knuthsort* has been parallelized into *PKnuthsort* and it scales reasonably with the number of available threads. With a single sorting process and 4 threads on the 4-core machine *RUNtime* goes down to 0.3 (theoretical optimum 0.25). Note this is *RUNtime*, not *CPUtime*, hence for 8 processes runTime must double (unlike CPUtime or *Energy*).

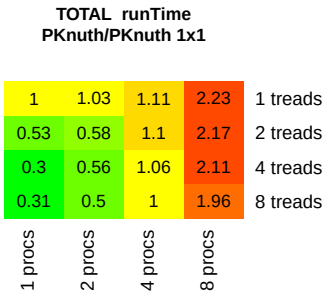

Figure 7.4: Parallel PKnuthsort runTimes relative to 1x1 (1 process 1 thread)

Energy savings by multi-threading are less impressive:

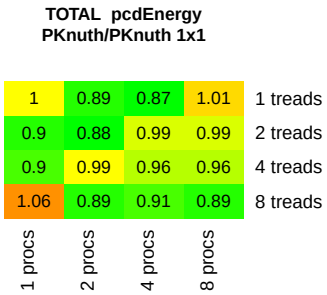

Figure 7.5: Parallel PKnuthsort pcdEnergy relative to 1x1 (1 process 1 thread)

7.1.2 Parallel Frogsorts

```
innovation("parallel-Symmetric-merging","B","iterative parallel merging")
dependency("parallel-Symmetric-merging","lean-parallel-merging")
innovation("PFrogsort0","E","Parallel Frogsort0")
innovation("PFrogsort1","E","Parallel Frogsort1")
innovation("PFrogsort2","E","Parallel Frogsort2")
innovation("PFrogsort3","E","Parallel Frogsort3")
dependency("PFrogsort0","parallel-Symmetric-merging")
dependency("PFrogsort0","Frogsort0")
dependency("PFrogsort1","parallel-Symmetric-merging")
dependency("PFrogsort1","Frogsort1")
dependency("PFrogsort2","parallel-Symmetric-merging")
dependency("PFrogsort2","Frogsort2")
dependency("PFrogsort3","parallel-Symmetric-merging")
dependency("PFrogsort3","Frogsort3")
```

Here I disclose how to parallelize *Frogsort*. Like with *Knuthsort* parallelizing the branches is trivial, except for some care to split the available threads proportional to the imbalanced branch-sizes in *Frogsort2* and *Frogsort3*. Parallelizing the *Symmetric-merging* is more tricky: it is not possible to parallelize the symmetric-merge at once, because part of the target memory is still occupied with the source data. Hence the parallel merging needs to be done iteratively for the amount of free buffer available. For the balanced *Frogsort0* and *Frogsort1* under random data this leads to merging 50% of the data, then 25% of the data, and so on. This carries a little more overhead and reduces the parallelism in the final iterations if one considers a minimum number of elements per thread. Still this scales quite well. I call this *parallel-Symmetric-merging*

Finally, a difference to Knuthsort is the parallelization of the setup phase. Frogsort0 uses a simple loop to setup its triplets, parallelizing is more successful than parallelizing the recursive setup of Frogsort1, Frogsort2, Frogsort3.

In the following *Frogsort Scaling* section parallel scenarios are compared to serial (1x1). In the following sections *Frogsort Speed*, *Frogsort Energy* and *Frogsort Footprint* each of the 16 scenarios is compared to *PKnuthsort*.

7.1.2.1 Frogsort Scaling

PFrogsort0 shows multi-threading behavior as good as *PKnuthsort* (and a little worse when running multiple processes with multiple threads):

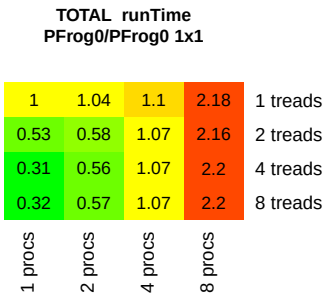

Figure 7.6: Parallel PFrogsort0 runTimes relative to 1x1 (1 process 1 thread)

PFrogsort1 multi-threading behavior is a bit worse (in one process) and a bit better when combined with multiple processes:

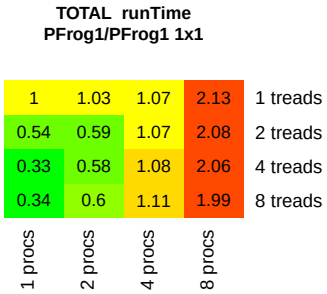

Figure 7.7: Parallel Frogsort1 runTimes relative to 1x1 (1 process 1 thread)

PFrogsort2 does not scale as good, but don't misinterpret this: the big advantage in serial executions diminishes when running *PFrogsort2* in parallel, but can still win over *PKnuthsort* (see the following sections):

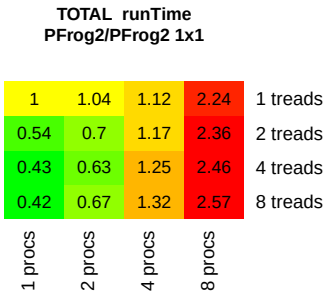

Figure 7.8: Parallel Frogsort1 runTimes relative to 1x1 (1 process 1 thread)

PFrogsort3 gives a similar picture:

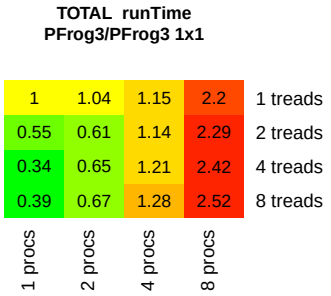

Figure 7.9: Parallel Frogsort1 runTimes relative to 1x1 (1 process 1 thread)

7.1.2.2 Frogsort Speed

PFrogsort0 is consistently faster than *PKnuthsort* if the number of threads does not exceed the number of physical cores:

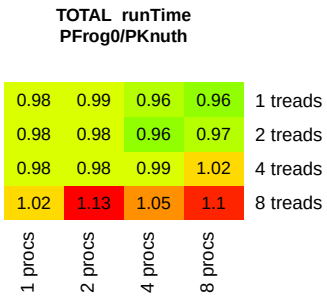

Figure 7.10: Parallel PFrogsort0 speed relative to Knuthsort

PFrogsort1 is somewhat slower than *PKnuthsort* particularly when multithreaded and without competing processes:

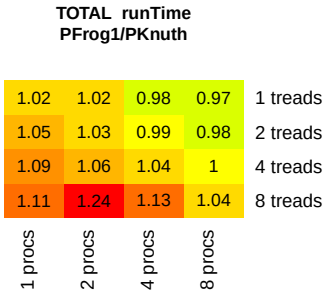

Figure 7.11: Parallel PFrogsort1 speed relative to Knuthsort

PFrogsort2 is faster than *PKnuthsort* except for high multi-threading and low parallel processes:

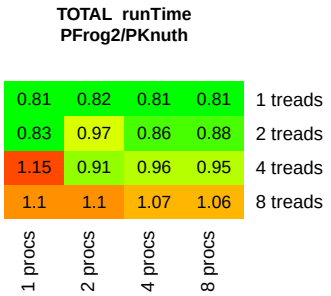

Figure 7.12: Parallel PFrogsort2 speed relative to Knuthsort

As a tribute to the low memory-requirement, *PFrogsort3* is consistently slower than *PKnuthsort*:

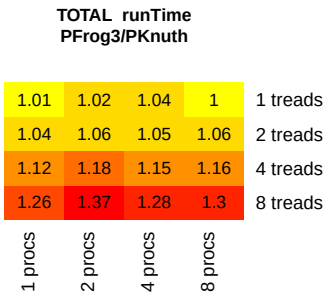

Figure 7.13: Parallel PFrogsort3 speed relative to Knuthsort

7.1.2.3 Frogsort Energy

PFrogsort0 is consistently more energy-efficient than *PKnuthsort* if the number of threads does not exceed the number of physical cores:

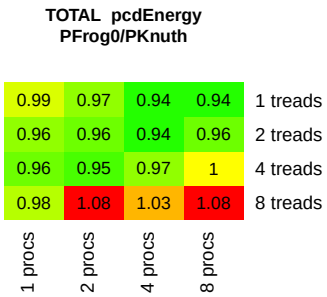

Figure 7.14: Parallel PFrogsort0 Energy relative to Knuthsort

PFrogsort1 needs the same amount of *Energy* than *PKnuthsort* – with less *%RAM* requirements – if the number of threads does not exceed the number of physical cores:

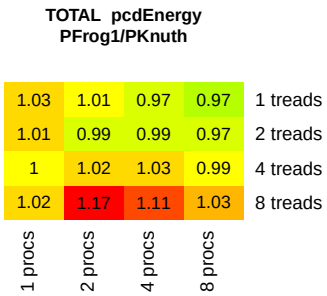

Figure 7.15: Parallel PFrogsort1 Energy relative to Knuthsort

With less memory requirements *PFrogsort2* is consistently more energy-efficient than *PKnuthsort* – if the number of threads does not exceed the number of physical cores:

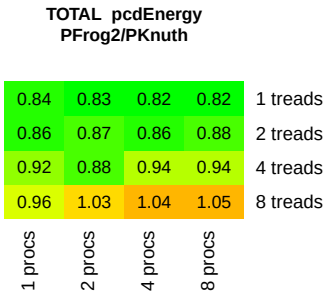

Figure 7.16: Parallel PFrogsort2 Energy relative to Knuthsort

PFrogsort3 tends to need more *Energy* than *PKnuthsort* (but less *%RAM*):

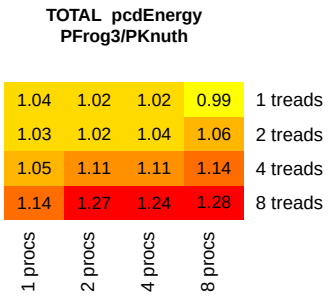

Figure 7.17: Parallel PFrogsort3 Energy relative to Knuthsort

7.1.2.4 Frogsort Footprint

PFrogsort0 and *PFrogsort1* have substantially lower *eFootprint* (about 75%) and *PFrogsort2* and *PFrogsort3* dramatically lower eFootprint (about 50%, particularly PFrogsort2).

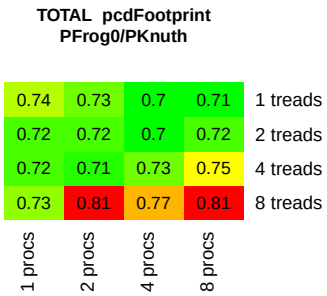

Figure 7.18: Parallel PFrogsort0 eFootprint relative to Knuthsort

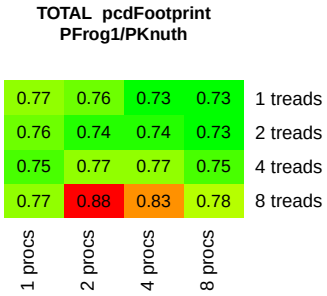

Figure 7.19: Parallel PFrogsort1 eFootprint relative to Knuthsort

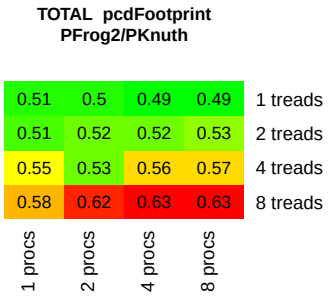

Figure 7.20: Parallel PFrogsort2 eFootprint relative to Knuthsort

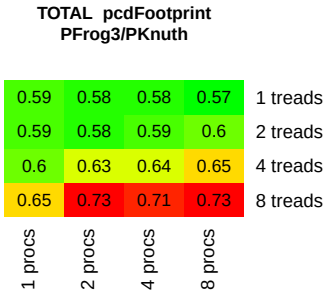

Figure 7.21: Parallel PFrogsort3 eFootprint relative to Knuthsort

In summary, needing less *%RAM*, the *Frogsorts* parallelize as well as *Mergesorts*, they have much lower *eFootprint* and *PFrogsort2* also needs less *Energy*. The energy (and speed) advantage of *PFrogsort2* reduces with more parallel execution, hence it is an open question what happens on CPUs with many more cores.

7.2 Parallel Quicksorts

7.2.1 Quicksort scaling

The parallel *Quicksorts* use multiple threads only for parallel branches, not for parallel partitioning, hence the expected speedup is lower. Here only the fastest versions with block-tuning are shown, *PQuicksort2B* and *PDucksortB*, the two have similar results and don't benefit from more than 2 threads:

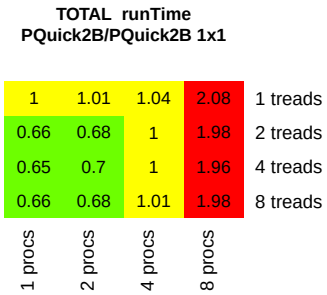

Figure 7.22: Parallel Quicksort2 runTimes relative to 1x1 (1 process 1 thread)

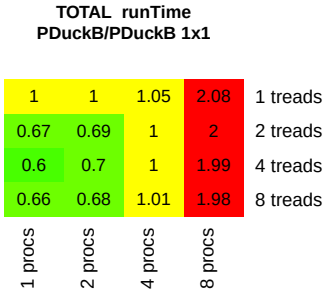

Figure 7.23: Parallel Ducksort runTimes relative to 1x1 (1 process 1 thread)

Note that regarding Energy not even two threads a clearly better than one:

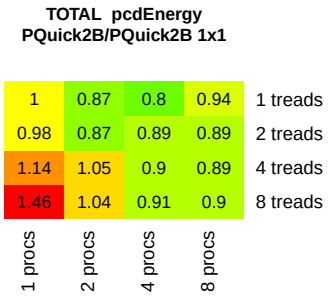

Figure 7.24: Parallel Quicksort2 Energy relative to 1x1 (1 process 1 thread)

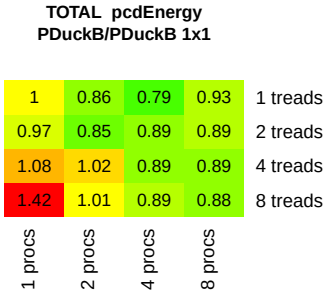

Figure 7.25: Parallel Ducksort Energy relative to 1x1 (1 process 1 thread)

7.2.2 Quicksort compared

PQuicksort2 is faster than *PKnuthsort*, except for high multi-threading and low parallel processes:

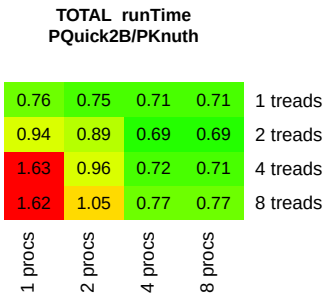

Figure 7.26: Parallel Quicksort2 speed relative to parallel Knuthsort)

PDucksortB does a bit better, but expectedly shows the same pattern:

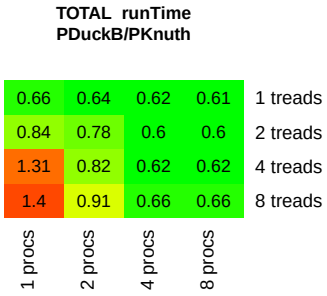

Figure 7.27: Parallel Ducksort (block-tuned) speed relative to parallel Knuthsort

Now comparing for Energy against *PKnuthsort*: if applicable and not multi-threaded too much *PQuicksort2B* saves 25% to 33%:

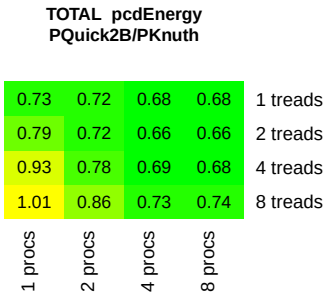

Figure 7.28: Parallel Quicksort2 Energy relative to parallel Knuthsort)

PDucksortB saves even more (35% to 40%):

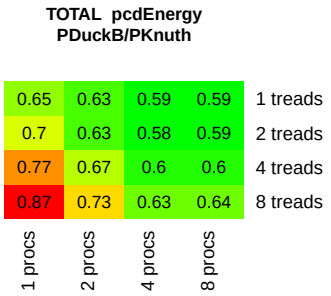

Figure 7.29: Parallel Ducksort Energy relative to parallel Knuthsort

Regarding *eFootprint* against *PKnuthsort*, the savings are dramatic, about 66%:

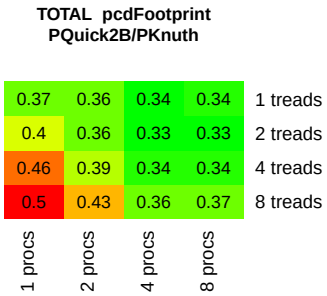

Figure 7.30: Parallel Quicksort2 eFootprint relative to parallel Knuthsort

PDucksort is even better:

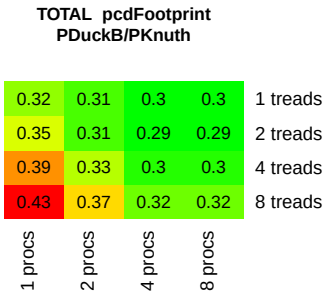

Figure 7.31: Parallel Ducksort eFootprint relative to parallel Knuthsort

In summary, partial parallelization of the *Quicksorts* gave some speed benefits, but neither achieves the speed of the parallel *Mergesorts*, nor achieves Energy benefits compared to serial execution. However, at 4 physical cores, regarding *Energy* and particularly regarding *eFootprint* the Quicksorts are still much better than the Mergesorts (provided they are applicable).

7.3 Update: AMD

Here I report some results with the 8-core AMD® Ryzen 7 pro 6850u (single runs with `n=2^27`), which gives quite similar results: *Frogsort0* scales as well as *Knuthsort*: both achieve speedup factor 8 with 16 hyper-threads. *Frogsort2* - after parallelizing some previously serial subtasks - achieves speedup factor 7.5 ex-situ and still factor 6.5 in-situ (results shown here are in-situ).

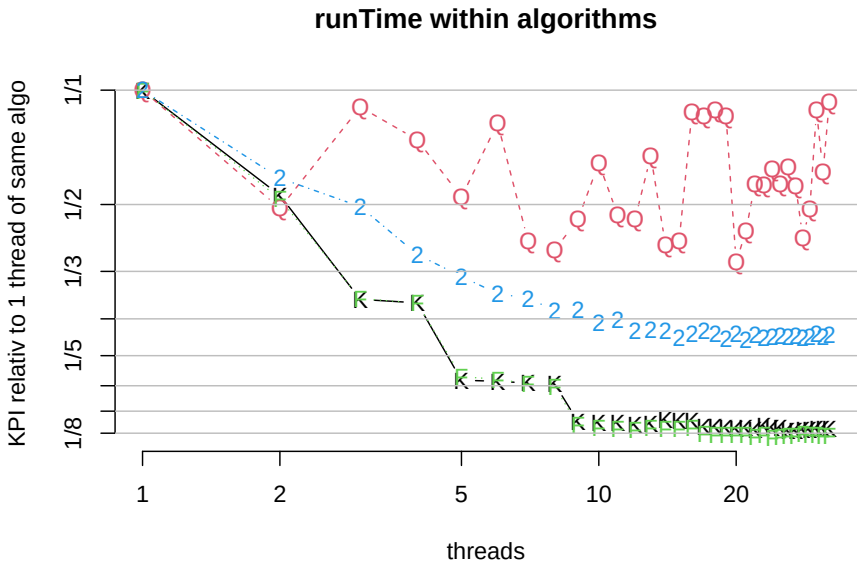

Figure 7.32: Scaling of parallel algorithms (runTime). (K)nuthsort (F)rosgsort0 Frogsort(2) (Q)uicksort2B

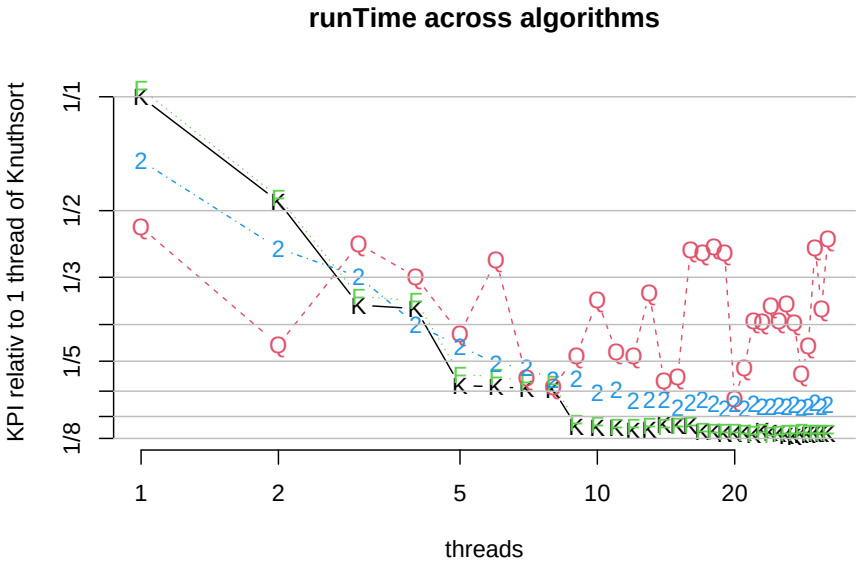

Figure 7.33: Comparison of parallel algorithms (runTime). (K)nuthsort (F)rosgsort0 Frogsort(2) (Q)uicksort2B

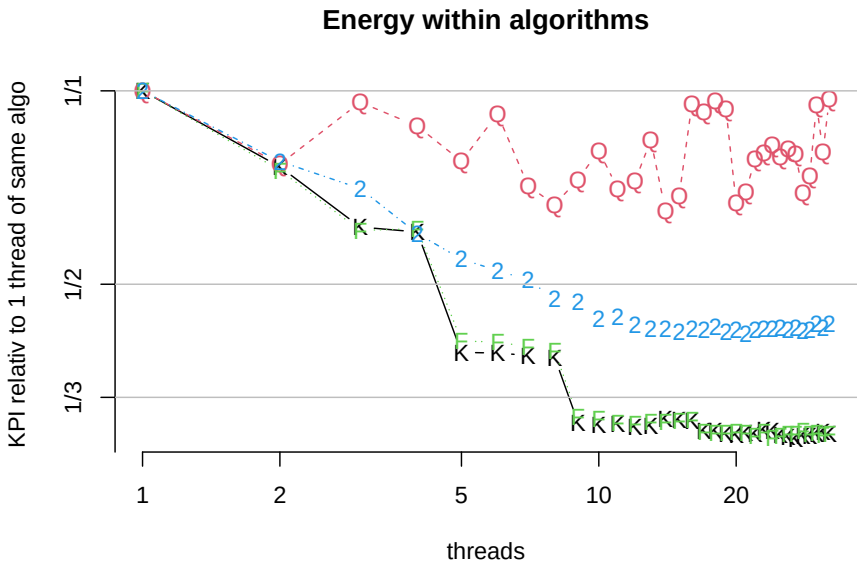

Figure 7.34: Scaling of parallel algorithms (bcdEnergy). (K)nuthsort (F)rosgsort0 Frogsort(2) (Q)uicksort2B

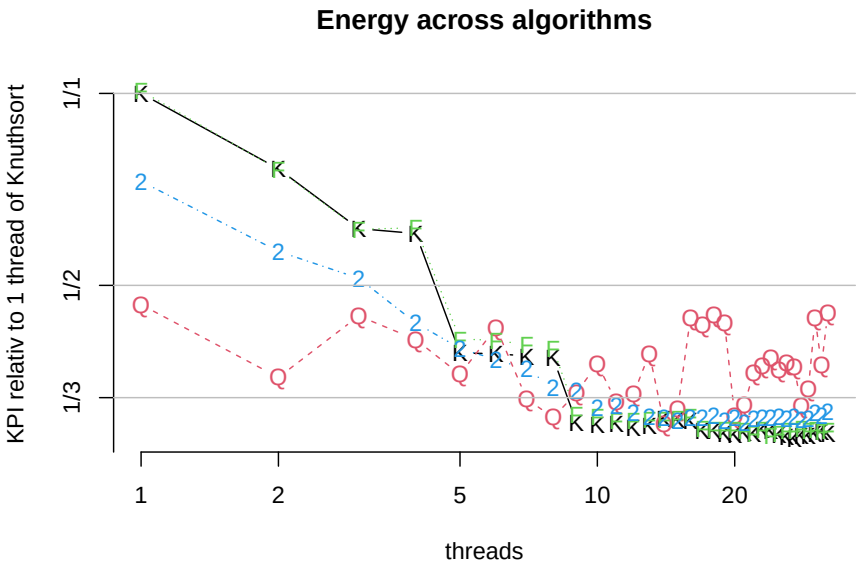

Figure 7.35: Comparison of parallel algorithms (bcdEnergy). (K)nuthsort (F)rosgsort0 Frogsort(2) (Q)uicksort2B

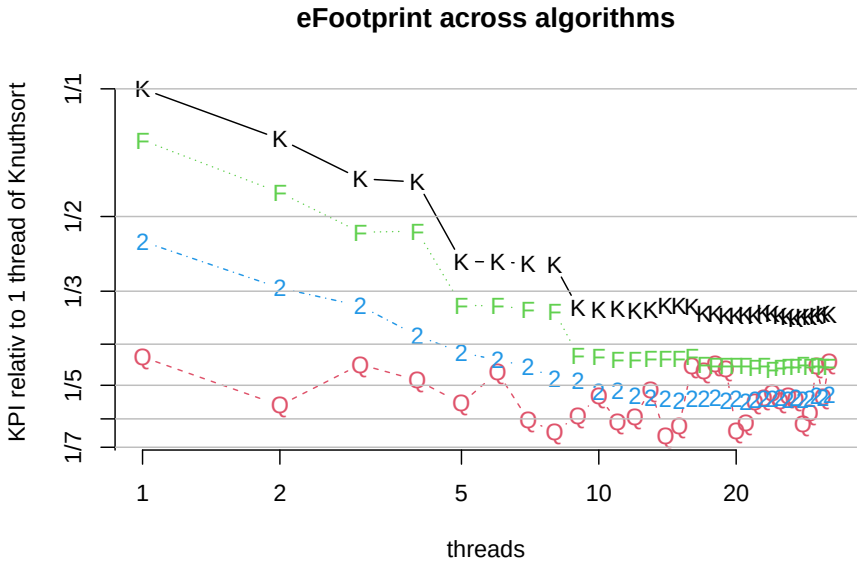

Figure 7.36: Comparison of parallel algorithms (bcdFootprint). (K)nuthsort (F)rosgsort0 Frogsort(2) (Q)uicksort2B

7.4 Parallel conclusion

Parallelization can save *Energy*, by running multiple concurrent tasks, by multi-threading tasks or by both.

The results have shown that the *greeNsort®* algorithms - despite their lower memory-requirements - can be multi-threaded as well as their prior art counterparts. *greeNsort® Split&Merge* algorithms have lower *eFootprint* than their prior art counterparts, and some even need less *Energy* or less *RUNtime*.

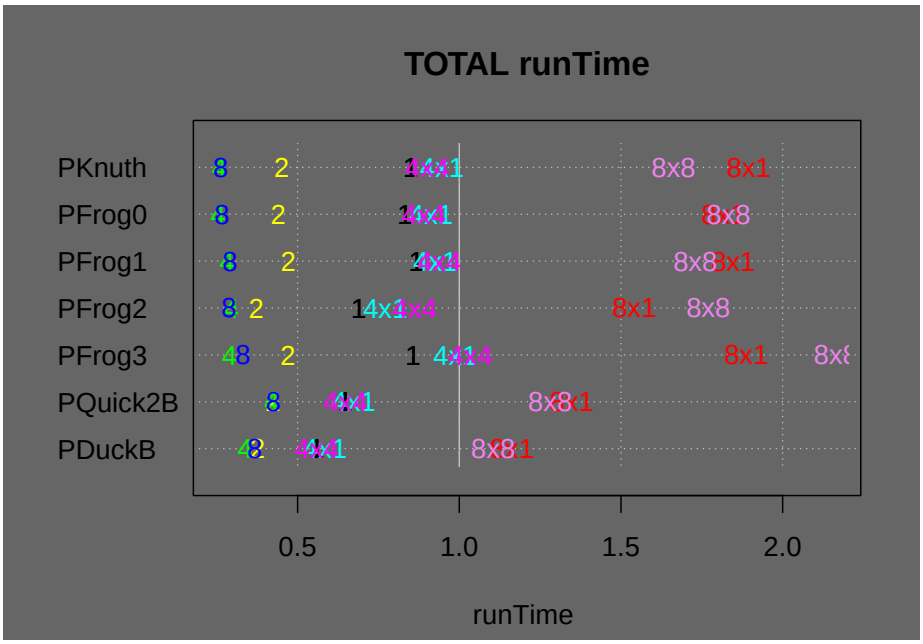

Figure 7.37: runTime summary of parallel algorithms

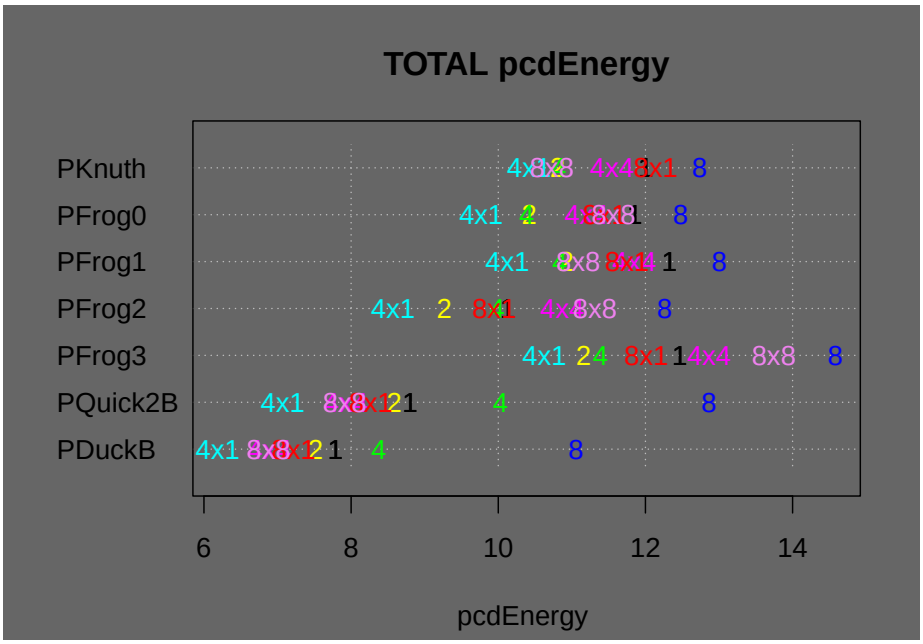

Figure 7.38: Energy summary of parallel algorithms

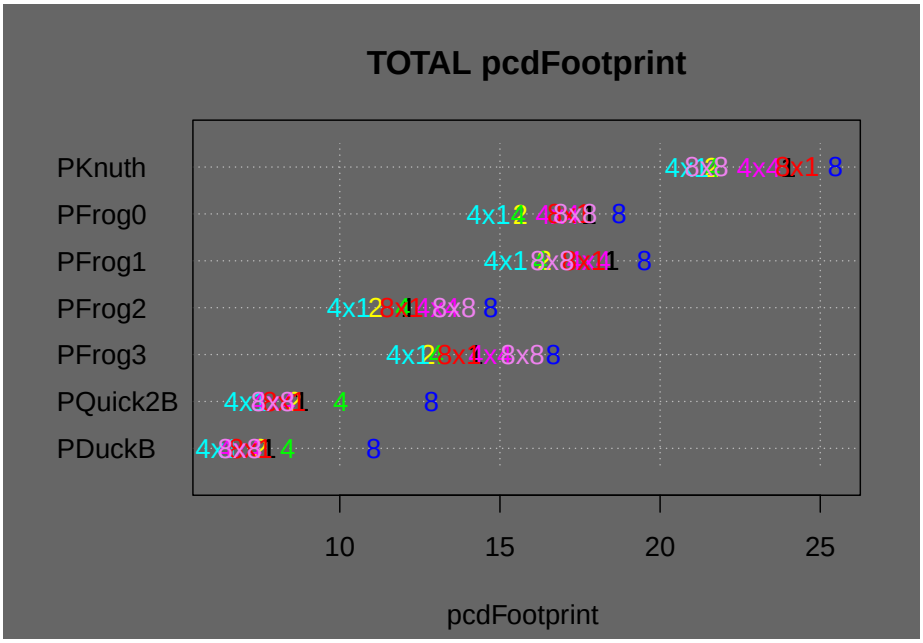

Figure 7.39: eFootprint summary of parallel algorithms

Chapter 8

Incremental sorting

```
innovation("Incremental-Zacksort","E","Amortized Incremental-Zacksort")
dependency("Incremental-Zacksort","Zacksort")
innovation("Zackheaps","E","Symmetric (tie-handling) Quickheaps")
dependency("Zackheaps","Incremental-Zacksort")
innovation("Incremental-Frogsort","E","Amortized Incremental-Frogsort")
dependency("Incremental-Frogsort","Frogsort")
innovation("Frogsteps","E","Symmetric stable dictionary")
dependency("Frogsteps","Incremental-Frogsort")
```

So far we have looked at basic batch-sorting, where the set of sorted elements is known once the sorting starts. *Incremental sorting* deals with expanding sorted sets as data arrives, potentially in single elements (*Online sorting*).

8.1 Incremental insertion

The most known Online sorting algorithm is *Insertionsort*, which by default does this: integrating a single element in a sorted set. Unfortunately, Insertionsort has $\mathcal{O}(N^2)$ cost, even if *binary search* is used to find the position in the sorted set at which to insert the new element (due to $\mathcal{O}(N)$ move cost per inserted element). No algorithm is known that can expand a contiguous sorted set by a single element at only $\mathcal{O}(\log N)$ cost. All $\mathcal{O}(N \log N)$ incremental sorting algorithms somehow relax the requirement, that a sorted set is maintained in contiguous space at any time (after each single element expansion). Examples of relaxing the requirement of contiguous sorting is *indexing* such as *binary trees*. Examples of relaxing the requirement that the set is sorted at any time is *amortized sorting* which somehow collects the changes and does the real work in batches.

An example of a combined strategy is *Librarysort* (Bender et al. (2006)), a variant of *Insertionsort* that keeps random gaps between the sorted elements (hence

not contiguous). With enough random gaps inserting a elements requires only local moves and hence $\mathcal{O}(1)$ insertion cost - like inserting a book in a book-shelf. Once the amount of gaps gets too low, physical rearrangement in a larger piece of memory costs $\mathcal{O}(N)$ which amortizes to $\mathcal{O}(N \log N)$. In spite of everyone's' experience with book-shelves this algorithms was discovered as late as 2006, and it has the underappreciated ability to sort elements of varying size.

A classic algorithm for real-time application is the maintenance of a heap, which integrates a new element at cost of only $\mathcal{O}(\log N)$, but sacrifices contiguous sorting (limited usefulness) and it sacrifices total speed due to heavy random access.

8.2 Incremental divide&conquer

A generic strategy to obtain a amortized incremental *Divide&Conquer* algorithm is to convert it into a data-structure on which the algorithm can be paused and resumed. Once that is available, it is possible to run it with interruptions and inject the new data at that position where the algorithm naturally would pick it up first. With this strategy each batch-algorithm hat its incremental dual.

For example the incremental dual of *Quicksort* is the *Incremental-Quicksort* of Navarro and Paredes (2010) with $\mathcal{O}(N \log N)$ amortized cost. By applying the *greeNsort®* innovations one obtains a *Incremental-Zacksort*[1] with $\mathcal{O}(N \log D)$ amortized cost, i.e. cheaper when duplicates reduce the number of distinct values such that $D < N$. Navarro and Paredes (2010) turned *Incremental-Quicksort* into a fully-fledged *priority queue* named *Quickheaps*. By combining with *Incremental-Zacksort* one obtains *Zackheaps* with efficient tie-handling[2].

Of course *Quickheaps* and *Zackheaps* inherit all weaknesses of *Quicksort* (see Split&Merge Algorithms). The incremental dual of *Frogsort* – *Incremental-Frogsort* – by contrast is *stable*, can handle elements of varying size and has deterministic $\mathcal{O}(N \log N)$ amortized cost[3]. Different from an *incremental Mergesort* that uses up to 100% buffer for maintaining a hierarchy of sorted runs, *Incremental-Frogsort* needs only 50% buffer. Different from *Librarysort* with its randomized gaps, *Incremental-Frogsort* has deterministic guarantees due to its *(F)lip (R)ecursive (O)rganized (G)aps (FROG)*.

Like *incremental Mergesort*, *Incremental-Frogsort* can be turned into a fully-fledged *sorted dictionary* named *Frogsteps*[4]. Note that the maximum memory requirement of *Frogsteps* is lower with 150% (compared to 200%), but the average memory requirement is not lower: when growing the structure to the next duplication, the traditional method must allocate 2x the current data size, *Frogsteps* must allocate 3x the current data size. Furthermore *Frogsteps* needs an

[1] not yet implemented
[2] not yet implemented
[3] implemented but so far not part of the test-bed
[4] not yet implemented

extra copy to migrate from the old to the grown memory, whereas the traditional method can *merge* into the grown memory, hence is always *nocopy*. But size duplication is a rare event in the lifetime of such a dictionary, during normal operation, *Frogsteps* benefits from reduced comparison and move-distance.

Such merge-based dictionaries do more work at insertion time than partition-based priority queues, and need less work for many queries, in other words: merge-based dictionaries have better read-amplification, partition-based priority queues have better write-amplification.

Such amortized data-structures and algorithms can further be developed into real-time indexes, such as *Lazy Search Trees* (Sandlund and Wild (2020)) or *LSM-Trees* (O'Neil et al. (1996)).

8.3 Incremental conclusion

The purpose of this section was to demonstrate that the *greeNsort®* innovations are orthogonal to a large body of existing literature and code which refers to binary *Divide&Conquer* sorting. Potentially all of these existing artefacts of the prior-art could be checked and adapted for the symmetric *greeNsort®* innovations. This could duplicate the number of these artefacts. Marketing on: *greeNsort®* is a contribution to exponential growth in scientific publications. Marketing off: realistically, *greeNsort®* is a contribution to correcting and optimizing, rather than duplicating human knowledge, most importantly when it comes to textbooks on data-structures and algorithms. But still: a fundamental an huge tidy-up project in human knowledge. Plus: I announce yet another plan for a new incremental sorting algorithm.

Chapter 9

Impact

In this section we give a perspective on the impact of the *greeNsort®* innovations on computational *Thinking*, *Research*, *Teaching*, *Libraries*, *APIs*, *IDEs*, *Languages*, *Compilers* and *Hardware* – to the degree that we already can imagine them today.

9.1 Thinking

I expect that the surprisingly successful development of the *greeNsort®* methods and algorithms with very few man years investment could somewhat change today's *computational thinking*. Liu et al. (2021) have already questioned the current scope as too narrow, I agree, but I think that their approach which adds *historical-thinking*, *data-centric-thinking* and *architectural thinking* still jumps too short. I plan a book chapter that will take a look at *recursive-thinking*, *symmetric-thinking*, *aesthetical-thinking*, *ethical-thinking* and *experimental-thinking* and their role in *problem-solving* for a *sustainable-society*.

9.2 Research

I expect short-term, mid-term and long-term impact on research:

- short-term: research on existing *greeNsort® algorithms* (like the publications following of the publication of *Dual-Pivot-Quicksort* of @Yaroslavskiy (2009), namely Wild (2012), Wild and Nebel (2012), Wild et al. (2013), Wild et al. (2015), Aumüller and Dietzfelbinger (2015), Martínez et al. (2015), Aumüller et al. (2016), Nebel et al. (2016), Wild et al. (2016), Wild (2016), Wild (2017), Wild (2018a), Wild (2018b), Martínez et al. (2019))
- mid-term: research applying *greeNsort® methods* and *principles* to further algorithmic problems

- long-term: research on *programming languages*, *compilers* and *CPUs* that support *greeNsort®* methods, particularly *Code-mirroring*

For example, the *B-level innovation – (D)istinct (I)dentification for (E)arly (T)ermination (DIET)* and *B-level innovation – (F)ast (L)oops (I)n (P)artitioning (FLIP)* methods have first been proven useful for solving *Quicksort-Dilemma* in *unstable binary* sorting, and then gave way to *Ducksort*. The section *Partial Z:cksort* has shown further implications for various closely related partial sorting algorithms, and section *Incremental sorting* has shown further implications for not so closely related priority queues. So far unstable sorting. Then DIET and FLIP could also be applied to *stable* binary *Partition&Pool* sorting, and I suspect it could play a role in *k-ary Partition&Pool* sorting, at least help to make *(I)n-place (P)arallel (S)uper (S)calar (S)ample(So)rt (IPS4o)* stable. From a more abstract perspective, the broadest implication of the *symmetric thinking* and *recursive thinking* behind *DIET* and *FLIP* is that they open the door to a new, unexplored part of an enlarged solution space for many algorithmic problems. In the statistical programming languages S and R there is the concept of *"computing on the language"*; this means using code to create code which is then executed; this increases the expressiveness of these interpreted languages but comes as a performance penalty. The *FLIP* method is an example of *computing on the recursion* which also increases expressiveness, but without such performance penalties. Clearly, if for each recursive recall we have to choose between *two* mirrored versions of code, this increases expressiveness by factor 2. In other words: *computing symmetry on the recursion* increases the solution space for algorithmic problems by at least factor 2. If we consider, that we actually do not mirror *all* code but *selected* code sections, the number of possible variations is higher than 2 (of which many are not useful, but some might). Of course, since all recursive algorithms can be coded without recursion, all of this enlarged solution space can theoretically be reached without *computing symmetry on the recursion*, but in practice, algorithms need to be written by humans who understand what they do, and hence *recursion* and *symmetry* are helpful *mental tools* for understanding problems and designing algorithms. Hence the concept of *computing symmetry on the recursion* is a powerful mental instrument that combines *recursive thinking*, *symmetric thinking* and *spatial thinking*. *Code mirroring* is at the junction between verbal and visual cognitive abilities, and that plays a role in problem solving and innovating. In summary, *computing symmetry on the recursion* is promising

- because of its potential to simplify (and visualize) complex algorithmic tasks
- because it is widely unexplored terrain
- because the inherent redundancy provides further optimization opportunities, see [*Languages*]

Note that this outlook has not even touched on the major share of the *greeNsort®* innovations: *Split&Merge* sorting. It also has not touched on *index-trees*. The

following quote shows a common belief that symmetric tree-traversal is not interesting:

> Binary tree is a tree structure, traversal is to make all nodes in the tree be visited only once, that is, arranged into a linear queue according to certain rules. A binary (sub) tree is a recursively defined structure, consisting of three parts: the root node (N), the left subtree (L), and the right subtree (R). Classify the traversal of the binary tree according to the access order of these three parts. There are 6 traversal schemes in total: NLR, LNR, LRN, NRL, RNL and LNR. Studying the traversal of binary trees is to study these 6 specific traversal schemes. Obviously, according to simple symmetry, the traversal of the left and right subtrees can be interchanged, that is, NLR and NRL, LNR and RNL, LRN and RLN Therefore, it is only necessary to study the three types of NLR, LNR and LRN, which are called "pre-order traversal", "medium-order traversal" and "post-order traversal".
>
> — TitanWolf - Binary tree traversal

Assuming upfront that mirrored versions of tree-operations can be ignored looks again like unexplored opportunities of a larger solution space.

9.3 Teaching

I expect short-term, mid-term and long-term impact on teaching:

- short-term: teaching existing *greeNsort® algorithms*, *methods* and *principles*. It is obvious, that textbooks on algorithms require updating their sections on *unstable sorting*, *partial sorting* and *selection* and *priority queues*. The same is true for sections on *stable sorting* and *incremental sorting*. Beyond that, textbooks need to update theirs sections on *recursion* and should teach the *greeNsort®* methods (such as *code-mirroring* and *computing-on-the-recursion*) and principles (such as *measuring sustainability* and *designing for simplicity*).
- mid-term: teaching the algorithms that result from research using *greeNsort®* methods and principles, see above
- long-term: teaching the results of research on *programming languages*, *compilers* and *CPUs* that support *greeNsort®* methods, see above and below

9.4 Libraries

The first motivation for *greeNsort®* was to provide more efficient sorting libraries in order to reduce CO2-emissions. Because of the bigger savings potential of

stable sorting the focus of *greeNsort*® is on alternatives to *Mergesort*, that save both, electricity and memory. Nonetheless, unstable *Quicksort* plays an important role in IT and most programming languages have an API for unstable in-place sorting, often using implementations inferior to *Pdqsort*, *Z:cksort* or *Ducksort*. *Z:cksorts* allow simple and efficient implementation which is superior to both *Quicksort2* and *Quicksort3*, notwithstanding further tuning. For example the C `sort` routine implements *3-way-Quicksort*. *Z:cksort* can replace it with the benefit of faster sorting untied data and still early terminating in tied data. The simplicity of *Z:cksort* facilitates verification of correctness of implementations.

Much more impact on sorting libraries is to be expected from the *greeNsort*® Split&Merge algorithms with their superior trade-off between memory and speed.

9.5 APIs

Each *Z:cksort* uses two binary comparison operators: `EQ` in the DIET loop and then `GT` (or `LT`) and `LE` (or `GE`). All of them can be derived from the `{-1|0|+1}` output of the ternary `cmd`-function which the C `sort` API takes as a parameter. However, using `cmp` is not efficient, because internally it always requires *two* comparisons, whatever the usage of its output is. Hence, further efficiency gains should be possible by creating a new API taking three binary comparison operators, or at least two of them. For those to whom this sounds overkill: C++ has introduced syntax for exactly this. I plan a book chapter on optimal design for sorting APIs.

9.6 IDEs

If *symmetric thinking* indeed gains importance in algorithm design, programmers will push for IDE support for mirroring code, for example 'copy' and 'paste mirrored'. However, that does not remove the burden to maintain the mirrored code, furthermore not all language element allow for mirroring, therefore more fundamental support for symmetry will be needed in *Languages*, *Compilers* and *Hardware*.

9.7 Languages

The mirroring of code in the *FLIP*-method comes with programming effort. Mirroring a piece of code is less work and less error-prone compared to writing new code, but still writing and maintaining it is work. However, it is work that can be automated by an interpreter, by a pre-processor or by a compiler. In other words: it is to be expected, that the *FLIP* method will have impact on future design of meta-programming or programming languages in order to facilitate programmer effort and minimize programming errors. To demonstrate

this, we give an implementation of *Zacksort* in R[1]. R is an advancement of the S language. It is related to scheme, a LISP dialect, and as such treats code as data and allows "computing on the language", in other words: meta-programming is built into the language. Note that this is just a demonstration of feasibility, since R as an interpreted language it too slow to be a serious implementation language for sorting algorithms. Having said this, let's do it. First we need the usual comparison functions:

```
EQ <- function(a,b)a==b
LE <- function(a,b)a<=b
GT <- function(a,b)a>b
LT <- function(a,b)a<b
GE <- function(a,b)a>=b
```

The `SELF`-function returns the code (parse tree) of the current function, and the `KEEP`-function protects code from being mirrored (and executes the code):

```
SELF <- function(){
  sys.function(sys.parent())
}
```

```
KEEP <- function(exp){
  eval.parent(substitute(exp))
}
```

The following `FLIP`-function returns a mirrored function of `f` (assuming `f` conforms with certain rules encoded in `FLIP`):

```
FLIP <- function (f)
{
  flip <- function(e) {
    if (is.call(e)) {
      if (is.symbol(e[[1]])) {
        if (e[[1]] != as.symbol("KEEP")) {
          for (i in seq_along(e)) {
            e[[i]] <- flip(e[[i]])
          }
        }
        e
      }
      else {
        e
      }
    }
    else if (is.symbol(e)) {
      s <- as.character(e)
      if (length(grep("_tieleft$",s))==1){
```

[1]The R Project for Statistical Computing

```
        fs <- sub("_tieleft","_tieright",s)
      }else if (length(grep("_tieright$",s))==1){
        fs <- sub("_tieright","_tieleft",s)
      }else{
        fs <- switch(s
          , l = "r"    , r = "l"
          , i = "j"    , j = "i"
          , f = "g"    , g = "f"
          , `+` = "-"  , `-` = "+"
          , `<` = ">"  , `>` = "<"
          , `<=` = ">=", `>=` = "<="
          , LT = "GT"  , GT = "LT"
          , LE = "GE"  , GE = "LE"
          , s
        )
      }
      as.symbol(fs)
    }
    else {
      e
    }
  }
  g <- f
  body(g) <- flip(body(g))
  return(g)
}
```

Here is a version of *Zocksort* simply calling itself via `SELF()` instead of the R-typical `Recall`-function:

```
zocksort <- function(x, l=1, r=length(x))
{
  if (l >= r)
    return
  j <- sample(r-l+1, 1)
  v <- x[j]
  x[c(l,j)] <- x[c(j,l)]              # SWAP pivot
  # DIET LOOP
  i <- l
  while (EQ(x[i <- i+1], v))
    if (i >= r)
      return(x)                       # early termination
  # MAIN LOOP
  i<-i-1; j<-r+1
  repeat{
    while ({j <- j-1; GT(x[j], v)})   # sentinel stop
```

```
      {NULL}
    while ({i <- i+1; LE(x[i], v)})
      if (i >= j)                    # explicit stop
        break
    if (i >= j)
      break
    x[c(i,j)] <- x[c(j,i)]           # SWAP
  }
  x[c(l,j)] <- x[c(j,l)]             # SWAP pivot back
  m <- j
  h <- SELF()                        # h(...) replaces Recall(...)
  n <- m - l; if (n>1) x[l:(m-1)] <- h(x[l:(m-1)], 1, n)
  n <- r - m; if (n>1) x[(m+1):r] <- h(x[(m+1):r], 1, n)
  return(x)
}
```

This is *Zacksort* calling a flipped version of itself:

```
zacksort <- function(x, l=1, r=length(x))
{
  KEEP(
    if (l >= r)
      return
  )
  j <- KEEP(sample(r-l+1, 1))
  v <- x[j]
  x[c(l,j)] <- x[c(j,l)]              # SWAP pivot
  # DIET LOOP
  i <- l
  while (EQ(x[i <- i+1], v))
    if (i >= r)
      return(x)                       # early termination
  # MAIN LOOP
  i<-i-1; j<-r+1
  repeat{
    while ({j <- j-1; GT(x[j], v)})   # sentinel stop
      {NULL}
    while ({i <- i+1; LE(x[i], v)})
      if (i >= j)                     # explicit stop
        break
    if (i >= j)
      break
    x[c(i,j)] <- x[c(j,i)]            # SWAP
  }
  x[c(l,j)] <- x[c(j,l)]              # SWAP pivot back
  m <- j
  h <- FLIP(SELF())
  KEEP({
    n <- m - l; if (n>1) x[l:(m-1)] <- h(x[l:(m-1)], 1, n)
    n <- r - m; if (n>1) x[(m+1):r] <- h(x[(m+1):r], 1, n)
  })
  return(x)
}
```

This is the *flipped Zacksort*:

```
> FLIP(zacksort)
function (x, l = 1, r = length(x))
{
    KEEP(if (l >= r)
        return)
    i <- KEEP(sample(r - l + 1, 1))
    v <- x[i]
    x[c(r, i)] <- x[c(i, r)]
    j <- r
    while (EQ(x[j <- j - 1], v)) if (j <= l)
        return(x)
    j <- j + 1
    i <- l - 1
    repeat {
        while ({
            i <- i + 1
            LT(x[i], v)
        }) {
        }
        while ({
            j <- j - 1
            GE(x[j], v)
        }) if (j <= i)
            break
        if (j <= i)
            break
        x[c(j, i)] <- x[c(i, j)]
    }
    x[c(r, i)] <- x[c(i, r)]
    m <- i
    h <- FLIP(SELF())
    KEEP({
        n <- m - l
        if (n > 1)
            x[l:(m - 1)] <- h(x[l:(m - 1)], 1, n)
        n <- r - m
        if (n > 1)
            x[(m + 1):r] <- h(x[(m + 1):r], 1, n)
    })
    return(x)
}
```

For comparison the double-flipped *Zacksort* (standardized without code comments):

```
> FLIP(FLIP(zacksort))
function (x, l = 1, r = length(x))
{
    KEEP(if (l >= r)
        return)
    j <- KEEP(sample(r - l + 1, 1))
    v <- x[j]
    x[c(l, j)] <- x[c(j, l)]
    i <- l
    while (EQ(x[i <- i + 1], v)) if (i >= r)
        return(x)
    i <- i - 1
    j <- r + 1
    repeat {
        while ({
            j <- j - 1
            GT(x[j], v)
        }) {
        }
        while ({
            i <- i + 1
            LE(x[i], v)
        }) if (i >= j)
            break
        if (i >= j)
            break
        x[c(i, j)] <- x[c(j, i)]
    }
    x[c(l, j)] <- x[c(j, l)]
    m <- j
    h <- FLIP(SELF())
    KEEP({
        n <- m - l
        if (n > 1)
            x[l:(m - 1)] <- h(x[l:(m - 1)], 1, n)
        n <- r - m
        if (n > 1)
            x[(m + 1):r] <- h(x[(m + 1):r], 1, n)
    })
    return(x)
}
```

and finally *Zucksort* which calls itself on the non-critical branch but its flipped-self on the critical branch:

```
zucksort <- function(x, l=1, r=length(x))
{
  KEEP(
    if (l >= r)
      return
  )
  j <- KEEP(sample(r-l+1, 1))
  v <- x[j]
  x[c(l,j)] <- x[c(j,l)]              # SWAP pivot
  # DIET LOOP
  i <- l
  while (EQ(x[i <- i+1], v))
    if (i >= r)
      return(x)                       # early termination
  # MAIN LOOP
  i<-i-1; j<-r+1
  repeat{
    while ({j <- j-1; GT(x[j], v)})   # sentinel stop
      {NULL}
    while ({i <- i+1; LE(x[i], v)})
      if (i >= j)                     # explicit stop
        break
    if (i >= j) break
    x[c(i,j)] <- x[c(j,i)]            # SWAP
  }
  x[c(l,j)] <- x[c(j,l)]              # SWAP pivot back
  m <- j
  g <- SELF()
  f <- FLIP(g)
  KEEP({
    n <- m - l; if (n>1) x[l:(m-1)] <- f(x[l:(m-1)], 1, n)
    n <- r - m; if (n>1) x[(m+1):r] <- g(x[(m+1):r], 1, n)
  })
  return(x)
}
```

We can formally verify the code-symmetry as true:

```
> # standardize Zacksort and Zucksort
> zacksort <- FLIP(FLIP(zacksort))
> zucksort <- FLIP(FLIP(zucksort))
> # single flip modifies the functions
> all.equal(FLIP(zacksort), zacksort)
[1] "target, current do not match when deparsed"
> all.equal(FLIP(zucksort), zucksort)
[1] "target, current do not match when deparsed"
> # double flip is idempotent
> all.equal(FLIP(FLIP(zacksort)), zacksort)
[1] TRUE
> all.equal(FLIP(FLIP(zucksort)), zucksort)
[1] TRUE
```

This section has shown that algorithms using the *FLIP*-method can be written without mirror-duplicating code-redundancy, in suitable languages.

9.8 Compilers

The above interpreted `FLIP` operation at runtime in each call of the R-zacksort is very inefficient. It is better to only do this only once at compile time. The seemingly least invasive thing to do is using meta-programming – for example the C-preprocessor – for creating mirrored versions of code, which then is compiled along the standard path, without any particular support for symmetry in the language, the compiler or the hardware. However, there might be reasons to consider symmetry in the language itself. For example many languages provide 'iterators'; these usually have a standard iteration direction and might not support efficient iteration in the reverse direction. C++, C# and Java have bi-directional iterators, Python and Rust have not, Mapcar in Lisp is unidirectional.

9.9 Hardware

Supporting symmetry in compilers but not in hardware still comes at a price: binary code duplication uses space in the instruction cache. If symmetric programming finds widespread use, then hardware support for symmetry – for example a FLIP instruction – might provide benefits. Co-design of language, compilers and hardware for symmetric programming is an interesting topic for future research.

9.10 Impact conclusion

Without knowing what will happen, I see good reasons to believe that – beyond saving time, energy and hardware – the *greeNsort®* innovations can have substantial impact on computational *Thinking*, *Research*, *Teaching*, *Libraries*, *APIs*, *IDEs*, *Languages*, *Compilers* and – hopefully – *Hardware*.

Chapter 10

Conclusion&Outlook

greeNsort® has rewound sorting research back to 1961 where Toni Hoare invented *Quicksort* and even further back to 1945 when John von Neumann described *Mergesort*. From there *greeNsort®* went on a journey through binary sorting in contiguous space. *greeNsort®* identified and challenged common core-beliefs of the prior art associated with these algorithms, introduced a web of innovations and suggested a classification for them.

greeNsort® introduced a method for measuring sustainability of algorithms that considers not only *RUNtime Energy* but also hardware *%RAM* requirements and combines both into the *Footprint* measure which allows a fair comparison of algorithms with different memory requirements.

greeNsort® introduced the *Ordinal-machine* model that motivates robust algorithms which reduce access- and move-distances. Distance-reduction is an underappreciated feature of *Quicksort*, and after resolving the *Quicksort-Dilemma* much of the *greeNsort®* project is about bringing distance-reduction to *Mergesort* and other *Divide&Conquer* algorithms which in the prior art move elements between two distant memory regions.

greeNsort® then introduced *Symmetric-asymmetry*, an important principle in many sciences, which surprisingly misses in computers science. Sorting and the von-Neumann machine is rife with asymmetries, namely *Access-asymmetry*, *Buffer-asymmetry*, *Order-asymmetry*, *Pivot-asymmetry* and *Tie-asymmetry*. Introducing symmetry into problem-solving with these asymmetries results in an enhanced solution space that contains many new methods and algorithms.

Donald Knuth's mathematical definition of sorting into *Ascending (Asc)* and *Descending (Desc)* addressed *Order-asymmetry* but ignored *Access-asymmetry* which arises as soon as sorting is done in a one-dimensional virtual memory that has *left* to *right* positions. *greeNsort®* introduces a new definition of *Symmetric-sorting* which differentiates *ascending* into *Ascending from Left (AscLeft)* and

Ascending from Right (AscRight). Using the example of *Bimesort* it was explained that unstable algorithms can be the consequence of confusing *right-ascending* with *left-descending.*

greeNsort® explained that many asymmetric problems can be resolved with *Symmetric-recursion*, for example the *Quicksort-Dilemma* – a result of *Access-asymmetry* and *Pivot-asymmetry.* Replacing two persistent prior art beliefs by *No3rd partition* and *Asymmetric-partitioning*, the solutions were elegant: combining the (symmetric) *B-level innovation – (F)ast (L)oops (I)n (P)artitioning (FLIP)* and the (lean) *B-level innovation – (D)istinct (I)dentification for (E)arly (T)ermination (DIET)* methods enabled with the *Z:cksorts* exactly those symmetric probabilistic algorithms that Tony Hoare tried to design with *Quicksort1*. Replacing the *DIET* with the *B-level innovation – (P)re-(O)rder (E)arly (T)ermination (POET)* method resulted in the even more adaptive *Ducksort.* This use of symmetry seamlessly led to *partial sorting* algorithms which consistently handle ties and return more useful information.

greeNsort® explained the space-time trade-off in *Mergesorts* and the *Mergesort-dilemma*: that the prior art knows either efficient *Knuthsort* needing 100% buffer or adaptive but inefficient like *Timsort* needing 50% buffer. *greeNsort®* explained that unlike *Bimesort*, *Knuthsort* can be made $\mathcal{O}(N)$ adaptive like *Timsort.* Introducing *Data-driven-location* enables *Omitsort* and combining this with core-belief *Undirected-sorting* and a new method *Data-driven-order* enables the bi-adaptive *Octosort.*

Then with a detour over new methods and algorithms (*Gapped-setup*, *Gapped-merging*, *GKnuthsort*, *Buffer-merging*, *TKnuthsort*, *Crocosort*) *greeNsort®* arrived at *Symmetric-merging* which allows distance reducing merging with only 50% buffer or less. Two basic algorithmic methods with different adaptivity were introduced (*Frogsort*, *Geckosort*) and several implementation variants explained (*Frogsort0* and *Frogsort1* with 50% buffer, *Frogsort2* with 14% buffer and even much faster than the competition. Alternative to *Buffer-merging* in the *Split&Merge* paradigm, *greeNsort®* suggested *Buffer-sharing* in the *Share&Merge* paradigm, which enables *Frogsort3* with even lower memory requirements (12.5% by default). The Frogsorts 1,2,3 can be joined in doubly parametrized *Frogsort6*.

greeNsort® then combined *Symmetric-merging* with the *Data-driven-order* method of *Omitsort* to obtain the bi-adaptive *Squidsort* algorithms, implemented *Squidsort1* and *Squidsort2* to show that they provide significant $\mathcal{O}(N \cdot \log N)$ adaptivity while – unlike *Timsort*, *Peeksort* and *Powersort* – keeping nocopy-efficiency in hard sorting tasks.

greeNsort® investigated some – possibly new – methods for merge-sorting with $\sqrt{N}$ buffer or less and found that $\sqrt{N}$ algorithms (particularly *Walksort*, *Jumpsort*) have better *Footprint* than prior art $N \log N$ algorithms, but not better than the simpler and more general *Frogsort2* with 14% buffer.

greeNsort® then turned to an under-appreciated generality of *Mergesorts*: that they can directly sort elements of varying size. Under the *C-level innovation*

– *(D)irect (S)orting (D)ifferent (S)ize (DSDS)* paradigm, two methods *B-level innovation – (D)irect (S)orting (N)ull-terminated (S)trings (DSNS)* and *B-level innovation – (D)irect (S)orting (P)ointered (E)lements (DSPE)* where presented, and the former used to implement *VKnuthsort* (100% buffer) and *VFrogsort1* (50% buffer) which use less *Energy* (and the latter less memory) than direct sorting methods such as *UKnuthsort*, *WQuicksort2* or *UZacksort*.

greeNsort® completed its exploration of new algorithms by diving into the *Partition&Pool* paradigm, where – as an exception – *Wing-partitioning* allows stable binary-partitioning with a single pass over the data per recursion level. *greeNsort®* implemented *Kiwisort* (using 100% distant buffer), introduced *Buffer-partitioning* and particularly *Symmetric-partitioning* with *Gapped-wrapup* to obtain *Swansort* (using 100% local buffer) and finally the travelling *Storksort* (using 50% local buffer) which can serve an exotic use-case (sorting over a wire).

greeNsort® compared a couple of parallel implementations to substantiate the claim that *Symmetric-merging* is as general as prior art merging. *greeNsort®* explained its preferred method for lean parallel merging (*lean-parallel-merging*) and how to parallelize *Symmetric-merging* (*parallel-Symmetric-merging*). After applying them to some of the above algorithms, it was shown that *PFrogsort0*, *PFrogsort1*, *PFrogsort2* and *PFrogsort3* parallelize almost as well as *PKnuthsort*.

greeNsort® finally glimpsed on the impact on Incremental sorting, priority-queues and sorted dictionaries. Using the *greeNsort®* classification, the height of the innovations can be quantified: Paradigmatic changes comprise of 12 new core beliefs (C-level innovations) and a new definition of sorting itself (D-level innovation). *greeNsort®* has introduced 20 new methods (B-level innovations) and 12 further methods (probably known T-level techniques) that allowed to design new algorithms. Here I reported 21 new algorithms, 9 extended algorithms and 8 further algorithms (A-, E- and F-level innovations).

Finally we looked on the *Impact* of *greeNsort®* on computational *Thinking*, *Research*, *Teaching*, *Libraries*, *APIs*, *IDEs*, *Languages*, *Compilers* and *Hardware*. Most important: the hand-coded *Symmetric-recursion* in the test-bed can also be generated with formal *Code-mirroring*, which leads to *Symmetric-languages*, mirroring *Symmetric-compilers* and supporting *Symmetric-hardware* instructions.

In summary: the aesthetic and ethical compass of the *greeNsort®* journey lead to *simplicity* and *sustainability*, also *robustness* and *resilience*. The focus was on *design* – for low *distance*. The contribution to research is the *revision of sorting* and *bringing symmetry to computer science*. The contribution to teaching is its *beauty of concepts* and its *poetry of code*. Welcome to the new era of *symmetric sustainable sorting* with better trade-offs between generality, simplicity, adaptivity and efficiency. Welcome also to a new field of research: investigating the implications of *Symmetric-asymmetry*, *Symmetric-recursion* and *Code-mirroring* on programming languages, compilers and hardware.

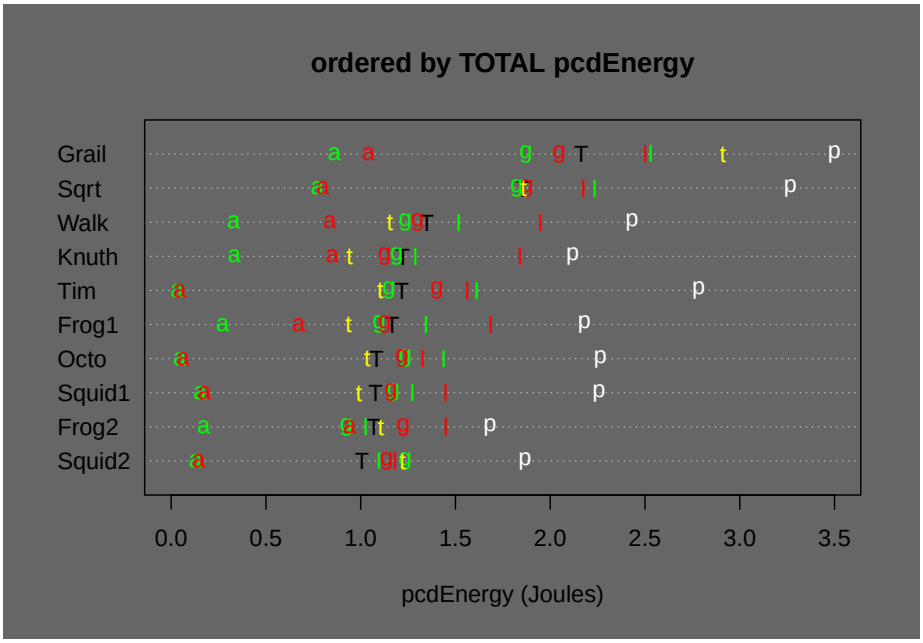

Figure 10.1: Medians of important algorithms ordered by TOTAL Energy. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

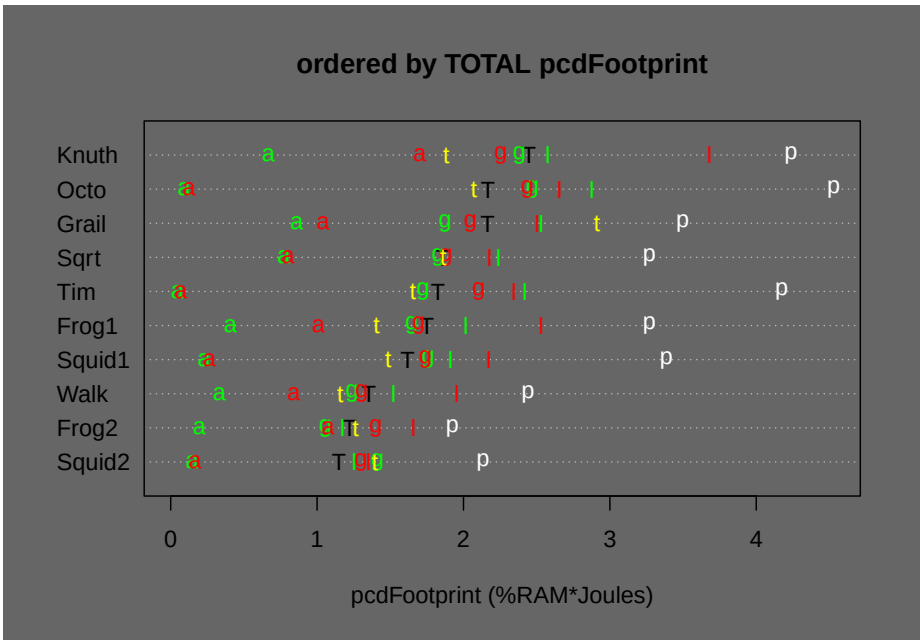

Figure 10.2: Medians of important algorithms ordered by TOTAL eFootprint. T=TOTAL, p=permut, t=tielog2; green: a=ascall, g=ascglobal, l=asclocal; red: a=descall, g=descglobal, l=desclocal

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* .

Terms&Tables

Definition&Convictions

Table 10.1: Definitions and Convictions

Symbol	Name	Memo
C*	Ordinal-machine	minimize monotonic cost of distance
	Footprint	measure variableCost x %fixedCost
	Symmetric-asymmetry	low-level-asymmetry, high-level-symmetry
D	Symmetric-sorting	'ascleft', 'ascright', 'descleft', 'descright'
C*	Symmetric-recursion	symmetric mutual recursion
	Code-mirroring	left-right mirroring of code
	Symmetric-compiler	compiling symmetric language
	Symmetric-hardware	instructions for mirroring code
C	No3rd-partition	Early Termination without 3rd partition
	Asymmetric-partitioning	Asymmetric-partitioning is good
	Undirected-sorting	not requesting asc/desc order
C*	Share&Merge	recursively share buffer until can splitting
C	DSDS	Direct Sorting Different Size

Bold techniques

Table 10.2: Bold techniques

Symbol	Name	Memo
B	Symmetric-language	symmetric semi-in-place merging data and buffer
	DIET	Distinct Identification for Early Termination
	FLIP	Fast Loops In Partitioning
	POET	Pre-Order Early Termination
	Data-driven-location	data decides return memory
	Data-driven-order	data decides asc/desc order
	Gapped-setup	odd-even data-buffer layout
	Gapped-merging	merge in odd-even data-buffer layout
	Buffer-merging	merging data and buffer
	Buffer-splitting	splitting data and buffer
	Frogsort	symmetric-merging ½-adaptive presorting
	Geckosort	symmetric-merging ¼-adaptive pre/rev
	Squidsort	lazy undirected symmetric sort
	DSNS	Direct Sorting Null-terminated Strings
	DSPE	Direct Sorting Pointered Elements
B*	Wing-partitioning	Stable partitioning without counting
	Buffer-partitioning	partitioning data and buffer
	Symmetric-partitioning	symmetric partitioning data and buffer
	Gapped-wrapup	collecting the gapped results
B	parallel-Symmetric-merging	iterative parallel merging

Algorithms

Table 10.3: Algorithms

Symbol	Name	Memo
A	Zocksort	self-recursive DIET Quicksort
	Zacksort	zig-zagging DIET-FLIP Quicksort
	Zucksort	semi-flipping DIET-FLIP Quicksort
	Ducksort	semi-flipping POET-FLIP Zucksort
	Omitsort	skip merge and return pointer to data
	Octosort	lazy directed undirected sort
	GKnuthsort	Knuthsort in odd-even data-buffer
	TKnuthsort	Knuthsort with buffer-merging (T-moves)
	Crocosort	Knuthsort with buffer-merging (R-moves)
	Frogsort0	Frogsort on triplets or bigger chunks
	Frogsort1	balanced Frogsort on single elements
	Frogsort2	imbalanced splitting Frogsort
	Frogsort3	imbalanced sharing Frogsort
	Frogsort6	generalized Frogsort
	Squidsort1	lazy symmetric sort (50% buffer)
	Squidsort2	lazy symmetric sort (<50% buffer)
	VKnuthsort	Varying-size Knuthsort
	VFrogsort1	Varying-size Frogsort1
	Kiwisort	Stable Partition&Pool sort 100% buffer
	Swansort	distance-reducing stable P&P sort 100% buffer
	Storksort	distance-reducing stable P&P sort 50% buffer

Extended&Further

Table 10.4: Extended and Further algorithms

Symbol	Name	Memo
F	Zackpart	MECEP partial sorting between l and r
	Zuckpart	MECEP partial sorting between l and r
	Zackselect	MECEP more informative than Quickselect
	Zuckselect	MECEP more informative than Quickselect
	Zackpartleft	MECEP partial sorting left of r
	Zuckpartleft	MECEP partial sorting left of r
	Zackpartright	MECEP partial sorting right of l
	Zuckpartright	MECEP partial sorting right of l
E	PKnuthsort	parallel Knuthsort
	PFrogsort0	Parallel Frogsort0
	PFrogsort1	Parallel Frogsort1
	PFrogsort2	Parallel Frogsort2
	PFrogsort3	Parallel Frogsort3
	Incremental-Zacksort	Amortized Incremental-Zacksort
	Zackheaps	Symmetric (tie-handling) Quickheaps
	Incremental-Frogsort	Amortized Incremental-Frogsort
	Frogsteps	Symmetric stable dictionary

Abbreviations

Asc Ascending

AscLeft Ascending from Left

AscRight Ascending from Right

AscPoor Ascending unstable

COMREC T-level technique - (COM)puting on the (REC)ursion

DAG (D)irected (A)cyclic (G)raph

Desc Descending

DescLeft Descending from Left

DescPoor Descending unstable

DescRight Descending from Right

DIET B-level innovation – (D)istinct (I)dentification for (E)arly (T)ermination

BIAS (B)inary (I)dentified (A)symmetric (S)earch

DONOTUBE T-level technique – (DO) (NO)t (TU)ne the (BE)st case

DSDS C-level innovation – (D)irect (S)orting (D)ifferent (S)ize

DSNS B-level innovation – (D)irect (S)orting (N)ull-terminated (S)trings

DSPE B-level innovation – (D)irect (S)orting (P)ointered (E)lements

EQ (EQ)ual

FLIP B-level innovation – (F)ast (L)oops (I)n (P)artitioning

FROG (F)lip (R)ecursive (O)rganized (G)aps

GECKO (G)ap (E)xchange (C)an (K)eep (O)rder

GE (G)reater (E)qual

GT (G)reater (T)han

IFS (Incremental) (F)rog (S)ort

IPS4o (I)n-place (P)arallel (S)uper (S)calar (S)ample(So)rt

IQS (Incremental) (Q)uick (S)ort

IZS (Incremental) (Z):ck (S)ort

KPI (K)ey (P)erformance (I)ndicator

LE (L)ower (E)qual

LT (L)ower (T)han

MECE (M)utually (E)xclusive and (C)ollectively (E)xaustive

MECEP T-level technique - (M)utually (E)xclusive and (C)ollectively (E)xaustive (P))artitioning

NE (N)ot (E)qual

NORELORE T-level technique – (NO)-(RE)gret (LO)op-(RE)use

PHOSITA (P)erson (H)aving (O)rdinary (S)kill (I)n (T)he (A)rt

POET B-level innovation – (P)re-(O)rder (E)arly (T)ermination

RAM (R)andom (A)ccess (M)emory

RAPL (R)unning (A)verage (P)ower (L)imit, Energy measurement of Intel

Glossary

%RAM Ratio of total memory relative to data size. $(data + buffer)/data$, see *Sustainable measurement* and *Methods&Measurement*

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, see *Asymmetries*

Asymmetric-partitioning C-level innovation – Asymmetric partitioning around a pivot is good, see *DIET method*

Bimesort T-level innovation – stable Bitonic Nocopy-Mergesort, see *First algo (Bimesort)*

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), see *Asymmetries*

Buffer-merging B-level innovation – merging data and buffer, see *Buffer-merging (TKnuthsort, Crocosort)*

Buffer-partitioning B-level innovation – partitioning data and buffer, see *Partition&Pool Algorithms*

Buffer-sharing T-level technique – dedicate buffer to one branch at a time (serially), see *Engineering Frogsort3*

Buffer-splitting T-level technique – split buffer into multiple parts that can be used parallel, see *Engineering Frogsort3*

cFootprint a *[Footprint measure]* using CPUtime for variableCost, $CPUtime\cdot$, see *Sustainable measurement* and *Methods&Measurement*

Chicksort A-level innovation – a Quicksort not stopping pointers on pivot-ties but still robust, instead of moving one pointer across all ties both pointers are moved alternating by one element, see *Symmetry*

Code-mirroring C-level innovation – left-right mirroring of redundant code sections, see *Mirroring code*

Crocosort A-level innovation – *Knuthsort* with buffer-merging (and Relocation moves), see *Buffer-merging (TKnuthsort, Crocosort)*

CPUtime measurement of CPU-time (summed over all parallel threads and processes), see *Sustainable measurement* and *Methods&Measurement*

RUNtime measurement of elapsed time (max-clock minus min-clock of all parallel threads and processes), see *Sustainable measurement* and *Methods&Measurement*

Data-driven-location B-level innovation – the data decides which memory is returned, see *Pre-Adpative (Omitsort)*

Data-driven-order B-level innovation – the data decides whether sorted or rev-sorted and returns the order, see *Bi-adpative (Octosort)*

Divide&Conquer T-level technique – recursive divide and conquer paradigm, see *Further techniques*

Ducksort A-level innovation – an adaptive *Zucksort* using the *POET-FLIP* methods, see *Engineering Ducksort*

DucksortB E-level innovation – branchless block-tuned version of Ducksort, see *Z:cksort implementation*

eFootprint a *[Footprint measure]* using an Energy measurement for variableCost, *Energy·*, see *pcdFootprint*, *Sustainable measurement* and *Methods&Measurement*

Energy a measurement of execution variableCost, here using RAPL, see *pcdEnergy*, *Sustainable measurement* and *Methods&Measurement*

Footprint C-level innovation – measure *variableCost·* for evaluating algorithms, see *tFootprint*, see *cFootprint*, see *eFootprint*, *Sustainable measurement* and *Methods&Measurement*

Frogsort B-level innovation – symmetric-merging ½-adaptive to presorting, see *Frogsort & Geckosort*

Frogsort0 A-level innovation – Frogsort on triplets or bigger chunks, see *Engineering Frogsort0*

Frogsort1 A-level innovation – balanced Frogsort on single elements, see *Engineering Frogsort1*

Frogsort1A F-level innovation – *Frogsort1* with non-overlap tuning for presorted data, for a bi-adaptive algorithm see *Squidsort1*

Frogsort2 A-level innovation – imbalanced splitting Frogsort on single elements, see *Engineering Frogsort2*

Frogsort2A F-level innovation – *Frogsort2* with non-overlap tuning for presorted data, for a bi-adaptive algorithm see *Squidsort2*

Frogsort3 A-level innovation – imbalanced sharing Frogsort on single elements, see *Engineering Frogsort3*

Frogsort6 A-level innovation – generalized Frogsort on single elements, see *Engineering Frogsort6*

Frogsteps E-level innovation – a dictionary based on [Incremental Frogsort], see *Incremental sorting*

Funnelsort a cache-oblivious Mergesort algorithm using the Van-Emde-Boas memory-layout (Prokop:1999, Frigo et al. (1999)), difficult for $N \neq 2^k$, for a simplified Version see *Lazy-Funnelsort*

Gapped-merging B-level innovation – merge in common odd-even data-buffer layout, see *[Gapped merging (GKnuthsort)]*

Gapped-setup B-level innovation – initial common odd-even data-buffer layout, see *[Gapped merging (GKnuthsort)]*

Gapped-wrapup B-level innovation – collecting the gapped results, see *Partition&Pool Algorithms*

Geckosort B-level innovation – symmetric-merging ¼-adaptive to pre-/rev-sorting, see *Frogsort & Geckosort*

Geckosort0 A-level innovation – *Geckosort* version of *Frogsort0*, see the testbed

Geckosort1 A-level innovation – *Geckosort* version of *Frogsort1*, see *Frogsort & Geckosort*

Geckosort2 A-level innovation – *Geckosort* version of *Frogsort2*

Geckosort3 A-level innovation – *Geckosort* version of *Frogsort3*

Geckosort6 A-level innovation – *Geckosort* version of *Frogsort6*

GKnuthsort A-level innovation – *Knuthsort* alternating between odd-even data-buffer positions, see *[Gapped merging (GKnuthsort)]*

Glidesort T-level technique – Peters (2023a);Peters (2023b), a natural-Mergesort tuned for adaptivity for presorted data and for duplicates by lazily delaying sorting of unsorted sequences in the hope to sort huge chunks of unsorted data with a partitioning sort, see also *Peeksort*

Grailsort T-level technique – an in-place mergesort invented by Huang and Langston (1992) and realized by Astrelin (2013), see *Low-memory*

Jumpsort A-level innovation – distance reducing version of *Walksort* using extra relocation moves, see *Low-memory*

IMergesort T-level technique – simple in-place-Mergesort as implemented by Astrelin, see *Low-memory*

In-place Parallel Super Scalar Samplesort In-place Parallel Super Scalar Samplesort *IPS4o* (Sanders:2004, Axtmann:2017)

Incremental-Frogsort E-level innovation – a lazy incremental Frogsort with not more than 50% buffer and amortized $\mathcal{O}(N \log N)$ cost, see *Incremental sorting*

Incremental-Quicksort a lazy incremental Quicksort by Navarro and Paredes (2010) with expected amortized $\mathcal{O}(N \log N)$ cost, see *Incremental sorting*

Incremental-Zacksort E-level innovation – a lazy incremental Zacksort with expected amortized $\mathcal{O}(N \log D)$ cost, see *Incremental sorting*

Introsort Introspective sort by Musser (1997) which combines a *Quicksort* with a fallback to *Heapsort* if the recursion-depth gets too deep, see *Z:cksort implementation*

Insertionsort a simple (incremental) sort that builds a sorted set by iteratively integrating single elements, it's likely very old, Knuth (1973) mentions that John Mauchly mentioned an optimization using [binary search] in 1946, see *Incremental sorting*

Heapsort Unstable in-place sort by Williams (1964), $\mathcal{O}(N \log N)$ worst-case, but not stable and not fast, due to wild random access, it first build a heap in-place, than removes one minimum (or maximum) value after the other, again in-place, see Z:cksort implementation

Kiwisort A-level innovation – a stable *wing-partitioning* sort with 100% distant buffer, see *Partition&Pool Algorithms*

Knuthsort T-level technique – Nocopy mergesort (Sedgewick (1998)) with Knuth's merge (Knuth (1973)), see *Reference algo (Knuthsort)*

Lazy-Funnelsort a simplified version of *Funnelsort* (Vinther:2003, Brodal et al. (2007)) which sacrifices the contiguous Van-Emde-Boas scheme for block-managed memory

Librarysort a non-contiguous lazy incremental Insertionsort (Bender et al. (2006)) using random-gaps with amortized $\mathcal{O}(N \log N)$ cost, see *Incremental sorting*

Mergesort class of (prior art) sorting algorithms based on merging, invented by von Neumann, see *The Mergesort-Dilemma*

Mergesort-dilemma either simple, efficient and parallel for non-sorted data (DivideConquer, i.e. Bimesort) or serial bi-adaptive to presorted data with halved buffer requirement (natural-Mergesort, i.e. Timsort), see *The Mergesort-Dilemma*

Mutual-recursion T-level technique – recursion with two mutually calling functions, is more expressive than self-recursion, see *Recursion model*

No3rd partition C-level innovation – Early Termination does not need a 3rd partition, not even two, see *DIET method*

Octosort A-level innovation – lazy bi-adaptive *undirected-sorting* variant of *Knuthsort* that return its order, see *[Bi-adaptive (Octosort)]*

Omitsort A-level innovation – lazy adaptive variant of *Knuthsort* that uses *[Data-driven location]*, see *[Pre-Adaptive (Omitsort)]*

Order-asymmetry the fact that that 'order' is asymmetric and reaches from 'low' to 'high' 'keys', see *Asymmetries*

Ordinal-machine C-level innovation – assume a monotonic cost of distance, hence minimize access- and move-distance, see *[Ordinal Machine]*

pcdEnergy the version of *RAPL* Energy used for comparison, see *Sustainable measurement* and *Methods&Measurement*

pcdFootprint the version of *RAPL eFootprint* used for comparison, see *Sustainable measurement* and *Methods&Measurement*

lean-parallel-merging T-level technique – lean parallel merging for an exact number of threads, see *Parallel merging*

Partition&Pool T-level technique – Recursive partition and pool paradigm, see *Partition&Pool Algorithms*

PDucksort Branch-Parallel *Ducksort* (without parallel partitioning), see *Z:cksort implementation*

PDucksortB Branch-Parallel block-tuned *DucksortB* (without parallel partitioning), see *Z:cksort implementation*

Peeksort T-level technique – Munro and Wild (2018) Peeksort, a natural-Mergesort tuned for adaptivity, but inefficient for real sorting tasks, see *The Mergesort-Dilemma* and *2nd Reference (Timsort)*

PFrogsort0 E-level innovation – Parallel *Frogsort0*, see *Parallel sorting*

PFrogsort1 E-level innovation – Parallel *Frogsort1*, see *Parallel sorting*

PFrogsort2 E-level innovation – Parallel *Frogsort2*, see *Parallel sorting*

PFrogsort3 E-level innovation – Parallel *Frogsort3*, see *Parallel sorting*

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, see *Asymmetries*

PKnuthsort E-level innovation – parallel *Knuthsort*, see *Parallel sorting*

Powersort T-level technique – Munro and Wild (2018) Powersort, a natural-Mergesort tuned for adaptivity, but inefficient for real sorting tasks, see *The Mergesort-Dilemma* and *2nd Reference (Timsort)*

PQuicksort2 Branch-Parallel *Quicksort2* (without parallel partitioning), see *Z:cksort implementation*

PQuicksort2B Branch-Parallel block-tuned *Quicksort2B* (without parallel partitioning), see *Z:cksort implementation*

parallel-Symmetric-merging B-level innovation – iterative parallel *Symmetric-Merging*, see *Parallel merging*

Quickheaps priority queue by Navarro and Paredes (2010) based on *[Incremental Quicksort]*, see *Incremental sorting*

Quickpart partial sorting between l and r using *Quicksort2* methods, see *Partial Z:cksort*

Quickpartleft partial sorting left of r using *Quicksort2* methods, see *Partial Z:cksort*

Quickpartright partial sorting right of l using *Quicksort2* methods, see *Partial Z:cksort*

Quickselect T-level technique – searching for position in sorted set using partial sorting with *Quicksort2* methods invented by Hoare (1961a);Hoare (1961c) under the name 'FIND', see *Partial Z:cksort*

Quicksort T-level technique – class of (prior art) algorithms invented by Tony Hoare, most notably see *Quicksort1*, *Quicksort2* and *Quicksort3*, see *The Quicksort-Dilemma*

Quicksort1 classic loop-symmetric Quicksort by Hoare (1961b), Hoare (1961a), see *The Quicksort-Dilemma*

Quicksort2 classic loop-symmetric Quicksort stopping at pivot-ties following Singleton (1969), see *The Quicksort-Dilemma*

Quicksort2B T-level technique – branchless version of Quicksort2 following Edelkamp and Weiß (2016),Edelkamp and Weiss (2016) (but without rudimentary early-termination on ties), see *The Quicksort-Dilemma*

Quicksort3 T-level technique – classic loop-symmetric Quicksort collecting ties in third partition following Bentley and McIlroy (1993), see *The Quicksort-Dilemma*

Quicksort-Dilemma either efficient for non-tied data (Quicksort2) or early termination on ties (Quicksort3), resolved by Z:cksort, see *The Quicksort-Dilemma*

RQuicksort2 stabilized indirect Quicksort2 using pointers, see *Stabilized Quicksorts*

RQuicksort2 block-tuned version of RQuicksort2, see *Stabilized Quicksorts*

Share&Merge C-level innovation – recursively share buffer until can splitting, see *Engineering Frogsort3*

Split&Merge T-level technique – Recursive split and merge paradigm, see *Split&Merge Algorithms*

Sqrtsort T-level technique – a square-root buffer mergesort derived from *Grailsort* and realized by Astrelin (2014), see *Low-memory*

SQuicksort2 stabilized indirect Quicksort2 moving elements and position-indicating integers, see *Stabilized Quicksorts*

SQuicksort2B block-tuned version of SQuicksort2, see *Stabilized Quicksorts*

Squidsort B-level innovation – bi-adaptive (lazily ordered) version of *Frogsort* better than *Timsort* for hard sorting tasks, see *Engineering Squidsort*

Squidsort1 A-level innovation – bi-adaptive (lazily ordered) symmetric sort with 50% buffer, see *Engineering Squidsort1*

Squidsort2 A-level innovation – bi-adaptive (lazily ordered) symmetric sort with less than 50% buffer, see *Engineering Squidsort2*

Storksort A-level innovation – a distance reducing stable partitioning sort with 50% buffer, see *Partition&Pool Algorithms*

Swansort A-level innovation – a distance reducing stable partitioning sort with 100% buffer, see *Partition&Pool Algorithms*

Symmetric-asymmetry C-level innovation – embracing low-level-asymmetry within high-level-symmetry, see *Symmetry principle*

Symmetric-compiler C-level innovation – compiling symmetric language, see *Mirroring code*

Symmetric-hardware C-level innovation – instructions for mirroring code, see *Mirroring code*

Symmetric-language C-level innovation – symmetric programming language, see *Mirroring code*

Symmetric-merging B-level innovation – symmetric merging data and buffer, needs only 50% buffer or less, see *[Symmetric merging]*

Symmetric-partitioning B-level innovation – symmetric partitioning data and buffer, see *[Symmetric merging]* and *Partition&Pool Algorithms*

Symmetric-recursion C-level innovation – mutual recursion with left-right-symmetry, see *Recursion model*

Symmetric-sorting D-level innovation – defines four targets (*ascleft*, *ascright*, *descleft*), *descright* instead of the classic two (*asc*, *desc*) for stable sorting, see *Definition of sorting*

tFootprint a *[Footprint measure]* using a runTime measurement for variable-Cost, *runTime·*, see *Sustainable measurement* and *Methods&Measurement*

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), see *Asymmetries*

Timsort T-level technique – Peters:2002 Timsort, a natural-Mergesort tuned for adaptivity, but inefficient for real sorting tasks, see *The Mergesort-Dilemma* and *2nd Reference (Timsort)*

TKnuthsort A-level innovation – *Knuthsort* with *buffer-merging* (and Transport moves), see *Buffer-merging (TKnuthsort, Crocosort)*

UKnuthsort T-level technique – stable indirect size-varying *Knuthsort*, see *Size-varying algos*

Undirected-sorting C-level innovation – sorting without specifying order (*data-driven-order*), see *Bi-adpative (Octosort)*

UZacksort E-level innovation – unstable indirect size-varying *Zacksort*, see *Size-varying algos*

UZacksortB E-level innovation – unstable indirect size-varying *ZacksortB* (block-tuned), see *Size-varying algos*

VFrogsort1 A-level innovation – stable direct size-varying *Frogsort1*, see *Size-varying algos*

VKnuthsort A-level innovation – stable direct size-varying *Knuthsort*, see *Size-varying algos*

Walksort A-level innovation – block-managed mergesort with sqrt buffer, more efficient than *Sqrtsort*, see *Low-memory*, for a distance-minimizing version see *Jumpsort*

Wing-partitioning B-level innovation – stable partitioning without counting by writing the two partitions from both ends and keeping track of the number of tie-reversals, see *Partition&Pool Algorithms*

WQuicksort2 T-level technique – stabilized indirect size-varying *Quicksort2*, see *Size-varying algos*

WQuicksort2B T-level technique – stabilized indirect size-varying *Quicksort2* (block-tuned), see *Size-varying algos*

Z:ckpart a common name for *Zackpart* and *Zuckpart* (and *[Duckpart]*), see *Partial Z:cksort*

Z:cksort a common name for *Zacksort* and *Zucksort* (and *Ducksort*), see *Partial Z:cksort*

Z:ckselect a common name for *Zackselect* and *Zuckselect* (and *[Duckselect]*), see *Partial Z:cksort*

Zackheaps E-level innovation – priority queue with efficient tie-handling combining *Quickheaps* with *[Incremental Zacksort]*, see *Incremental sorting*

Zackpart F-level innovation – MECE partial sorting between l and r using *Zacksort* methods, see *Partial Z:cksort*

Zackpartleft F-level innovation – MECE partial sorting left of r using *Zacksort* methods, see *Partial Z:cksort*

Zackpartright F-level innovation – MECE partial sorting right of l using *Zacksort* methods, see *Partial Z:cksort*

Zackselect F-level innovation – MECE more informative than *Quickselect* using *Zacksort* methods, see *Partial Z:cksort*

Zacksort A-level innovation – zig-zagging *DIET-FLIP* Quicksort, see *Engineering Zacksort*

ZacksortB E-level innovation – branchless block-tuned version of Zacksort, see *Z:cksort implementation*

Zocksort A-level innovation – self-recursive DIET Quicksort, see *Engineering Zocksort*

Zuckpart F-level innovation – MECE partial sorting between l and r using *Zucksort* methods, see *Partial Z:cksort*

Zuckpartleft F-level innovation – MECE partial sorting left of r using *Zucksort* methods, see *Partial Z:cksort*

Zuckpartright F-level innovation – MECE partial sorting right of l using *Zucksort* methods, see *Partial Z:cksort*

Zuckselect F-level innovation – MECE more informative than *Quickselect* using *Zucksort* methods, see *Partial Z:cksort*

Zucksort A-level innovation – a semi-flipping *Zacksort*, see *Engineering Zucksort*

ZucksortB E-level innovation – branchless block-tuned version of Zucksort, see *Z:cksort implementation*

Bibliography

Astrelin, A. (2013). Grailsort.

Astrelin, A. (2014). Sqrtsort.

Aumüller, M. and Dietzfelbinger, M. (2015). Optimal partitioning for dual-pivot quicksort. *ACM Trans. Algorithms*, 12(2).

Aumüller, M., Dietzfelbinger, M., and Klaue, P. (2016). How good is multi-pivot quicksort? *ACM Trans. Algorithms*, 13(1).

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.

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.

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. (1973). Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461.

Brodal, G. S., Fagerberg, R., and Vinther, K. (2007). Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Edelkamp, S. and Weiss, A. (2016). BlockQuicksort: Avoiding Branch Mispredictions in Quicksort. In Sankowski, P. and Zaroliagis, C., editors, *24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA. IEEE Computer Society.

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.

Kamp, P.-H. (2011). The most expensive one-byte mistake. *Commun. ACM*, 54(9):42–44.

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.

Kuhn, T. S. (1962). *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago.

Liu, Y., Sun, X.-H., Wang, Y., and Bao, Y. (2021). Hcda: From computational thinking to a generalized thinking paradigm. *Commun. ACM*, 64(5):66–75.

Martínez, C., Nebel, M., and Wild, S. (2019). Sesquickselect: One and a half pivots for cache-efficient selection. In Mishna, M. and Munro, J. I., editors, *Proceedings of the Sixteenth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2019, San Diego, CA, USA, January 6, 2019*, pages 54–66. SIAM.

Martínez, C., Nebel, M. E., and Wild, S. (2015). Analysis of branch misses in quicksort. In Sedgewick, R. and Ward, M. D., editors, *Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2015, San Diego, CA, USA, January 4, 2015*, pages 114–128. SIAM.

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.

Musser, D. R. (1997). Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993.

Navarro, G. and Paredes, R. (2010). On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57(4):585–620.

Nebel, M. E., Wild, S., and Martínez, C. (2016). Analysis of pivot sampling in dual-pivot quicksort: A holistic analysis of yaroslavskiy's partitioning scheme. *Algorithmica*, 75(4):632–683.

Oehlschlägel, J. and Silvestri, L. (2012). *bit64: A S3 Class for Vectors of 64bit Integers*. R package first version.

O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. (1996). The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385.

Peters, O. (2014). [patch] bug 20837 - libc++ std::sort has o(n^2) 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, O. R. L. (2023a). Glidesort. Github.

Peters, O. R. L. (2023b). Glidesort, efficient in-memory adaptive stable sorting on modern hardware. Talk given at FOSSDEM23.

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

Sandlund, B. and Wild, S. (2020). Lazy search trees. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 704–715.

Sedgewick, R. (1975). *Quicksort*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York.

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

Sedgewick, R. (1977b). Quicksort with equal keys. *SIAM J. Comput*, pages 240–267.

Sedgewick, R. (1978). Implementing quicksort programs. *Commun. ACM*, 21(10):847–857. Corrigendum: CACM 22(6): 368 (1979).

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

Sedgewick, R. (1998). *Algorithms in C, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*. Addison-Wesley Professional, 3 edition.

Sedgewick, R. and Bentley, J. (2002). Quicksort is optimal. In *KnuthFest*. Stanford University.

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

Steffen, D. (2010). Design semantics of innovation. *Design Semiotics in Use*, page 82–110.

Wild, S. (2012). Java 7's dual pivot quicksort. Master's thesis, University of Kaiserslautern.

Wild, S. (2016). *Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential*. doctoralthesis, Technische Universität Kaiserslautern.

Wild, S. (2017). Quicksort mit zwei pivots und mehr: Eine mathematische analyse von mehrwege-partitionierungsverfahren und der frage, wie quicksort dadurch schneller wird. In Hölldobler, S., editor, *Ausgezeichnete Informatikdissertationen 2016*, pages 319–328, Bonn. Gesellschaft für Informatik e.V.

Wild, S. (2018a). Dual-pivot and beyond: The potential of multiway partitioning in quicksort. *it - Information Technology*, 60(3):173–177.

Wild, S. (2018b). Quicksort is optimal for many equal keys. In Nebel, M. E. and Wagner, S. G., editors, *Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2018, New Orleans, LA, USA, January 8-9, 2018*, pages 8–22. SIAM.

Wild, S. and Nebel, M. E. (2012). Average case analysis of java 7's dual pivot quicksort. In *European Symposium on Algorithms 2012 (LNCS)*, volume 7501, page 825–836.

Wild, S., Nebel, M. E., and Mahmoud, H. M. (2016). Analysis of quickselect under yaroslavskiy's dual-pivoting algorithm. *Algorithmica*, 74(1):485–506.

Wild, S., Nebel, M. E., and Neininger, R. (2015). Average case and distributional analysis of dual-pivot quicksort. *ACM Trans. Algorithms*, 11(3):22:1–22:42.

Wild, S., Nebel, M. E., Reitzig, R., and Laube, U. (2013). Engineering java 7's dual pivot quicksort using malijan. In *ALENEX*.

Williams, J. W. J. (1964). Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348.

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