Contents

Todo list

Architektur

Speicherzugriffe (memcpy, mram_read ...)

triple buffer

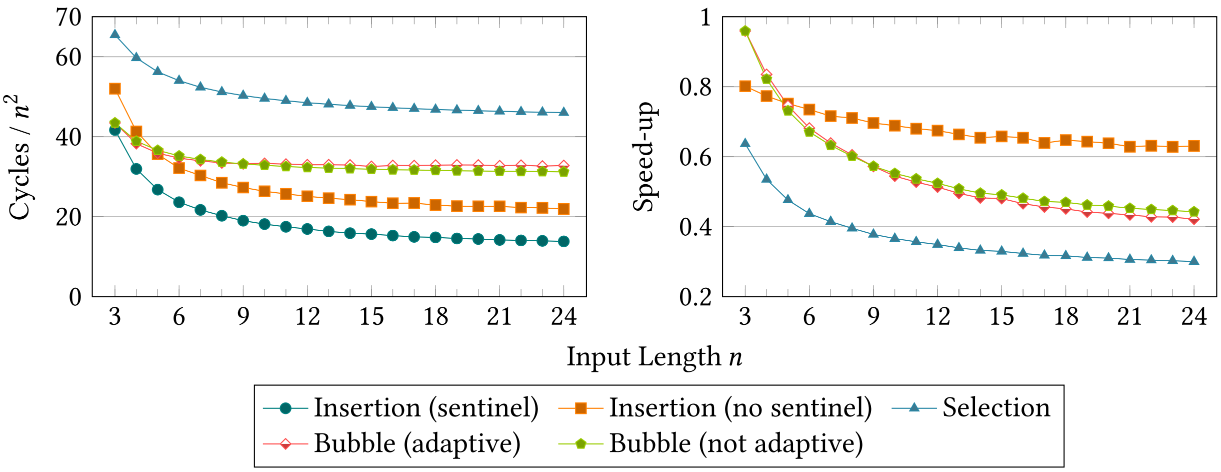


Figure 1: Comparison of sorting algorithms with a runtime in $O(n^2)$. The adaptive BubbleSort terminates prematurely if no changes were made to the input array during an iteration. The speed-ups are with respect to the InsertionSort relying on sentinel values.

1 Sorting with One Tasklet

This section covers the very first phase where each tasklet sorts on its own, i. e. sequentially. Unless specified otherwise, every measurement in this section was conducted on a uniform input distribution with each 32-bit integer drawn independently from the range $[0, 2^{32} - 1]$, and the default configurations of the sorting algorithms were as follows:

InsertionSort using one sentinel value

ShellSort using h_1 sentinel values

QuickSort iterative implementation; switching to InsertionSort whenever 13 elements or less remain in a partition; median of three as pivot; prioritising the right-hand partition over the left-hand partition

1.1 InsertionSort

This stable sorting algorithm works by moving the ith element to the left as long as its left neighbour is bigger, assuming that the elements 0 to $i - 1$ are already sorted. Even though in both the average case and the worst case, InsertionSort has a runtime of $O(n^2)$, it features quite some advantages: 1. It works in-place, needing only $O(1)$ additional space. 2. It is inherently adaptive: If the input array is mostly or even fully sorted, the runtime drops down to $O(n)$. 3. Its program code is short, lending itself to inlining. 4. The overhead is small. Especially the last two points make InsertionSort a good base algorithm for asymptotically better sorting algorithms to use on very small subarrays.

Beleg?

When moving an element to the left, two checks are needed: Does the left neighbour exist and is it smaller than the element to move? The first check can be omitted through the use of *sentinel values*: If the element at index -1 is at least as small as any value in the input array, the leftwards motion stops there at the latest. Since a DPU has no branch predictor, the slowdown from performing twice as many checks as needed is quite high and lies between 20% and 40% in the relevant input range (Fig. 1). Thence, 'InsertionSort' refers to the version relying on sentinel values henceforth.

auf Kom-

ex- und implizite Wächter- werte ben- ennen

auf Compilersperenzchen eingehen?

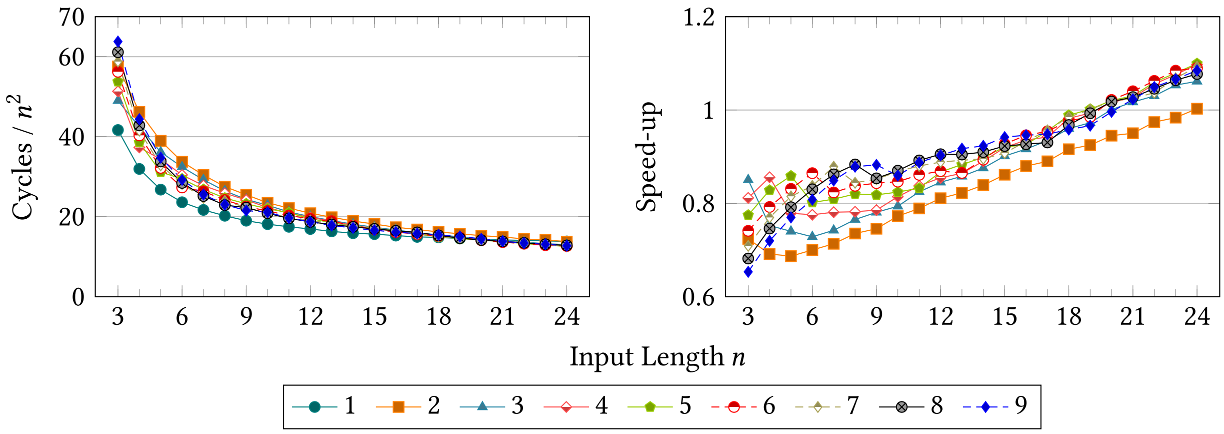


Figure 2: Comparison of InsertionSort (1) and various ShellSorts (2–9). Each ShellSort does one InsertionSort pass with a step size between 2 and 9 before doing a pass of regular InsertionSort. The speed-ups are with respect to the InsertionSort.

Note. Other known simple sorting algorithm with similar runtime complexity are SelectionSort and BubbleSort. The asymptoticity, however, hides much higher constant factors such that even for as little as three elements InsertionSort is superior (Fig. 1) and should always be preferred.

1.2 ShellSort

InsertionSort suffers from small elements at the end of the input, since those have to be brought to the front through $O(n)$ comparisons and swaps. ShellSort, proposed by Donald L. Shell in 1959 [4], remedies this by doing multiple passes of InsertionSort with different step sizes: In round r with step size h_r, the input array is divided into the subarrays of indices $(i, h_r+i, 2h_r+i, \dots)$ for $i = 0, \dots, h_r - 1$ which then get sorted individually through InsertionSort. The step size get smaller each round, with the final step size being 1 such that a regular InsertionSort is performed. Intuitively, the individual InsertionSorts are fast since elements which need to travel long distances already did big jumps. Finding the right balance between the heightened overhead through multiple InsertionSort passes and the shortened runtime of each InsertionSort pass is subject to research to this day [3, 5] and depends on the cost of the operations (comparing, swapping, looping).

Let us first focus on small input arrays where only two rounds with step sizes h_1 and 1 suffice. The previous results on InsertionSort suggest that ShellSort should make use of h_1 sentinel values lest bounds checking eats any gain up. Figure 2 shows that the additional rounds starts to pay off at around 20 elements for $h_1 \geq 3$. Bear in mind that these measurements were conducted on a uniform input distribution; if ShellSort is used by another algorithm on a subarray, these thresholds may be higher or even non-existent due to some degree of presorting.

When moving to greater input lengths (Fig. 3), the differences in performance between the two-round ShellSorts become more pronounced; especially the ones with $h_1 = 3$ and $h_1 = 4$ fall off whereas the one with $h_1 = 6$ holds its ground quite well. With more than 64 elements, three round get worthwhile to consider. Interestingly, many ShellSorts with $h_2 = 4$ take the lead whilst the ones with $h_2 = 6$ are mid-table. This is in accordance with Ciura [1] who noted $h = (17, 4, 1)$ to be the optimal triplet for 128 elements. It is noteworthy, though, that he measured the quadruplet $h = (38, 9, 4, 1)$ to be about 5% faster in the MIX machine model. On a DPU, this sequence leads to a runtime of nearly exactly 74.000 cycles, placing it only mid-table. Without access to Ciura's original code, giving a satisfactory explanation for the discrepancy is hard, however.

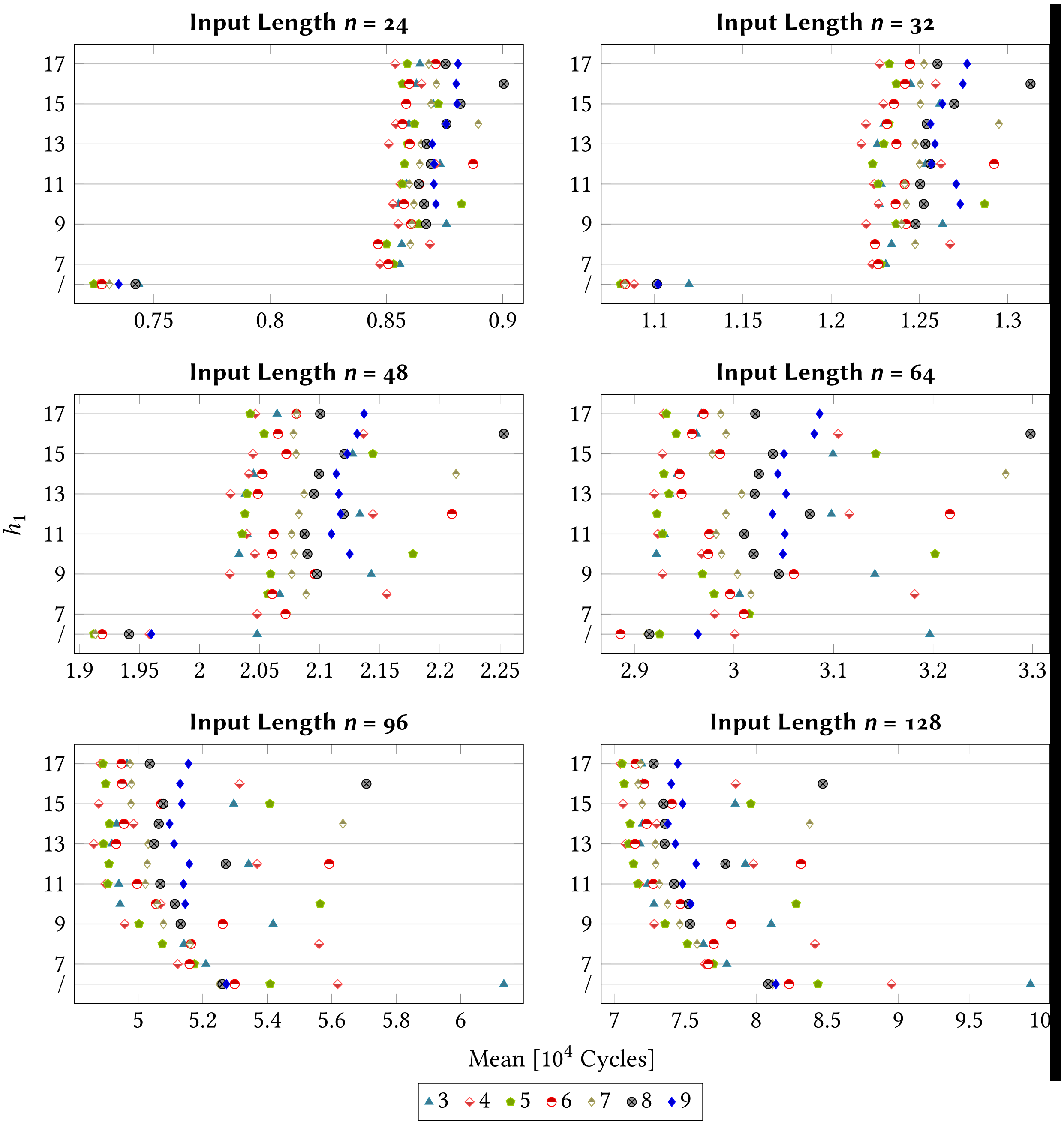


Figure 3: Runtimes of ShellSorts with two rounds (/) and three rounds (7–17). The coloured symbols encode the step size h_1 for two-round ShellSorts and the step size h_2 for three-round ShellSorts. For the latter, the step size h_1 is noted on the y-axes.

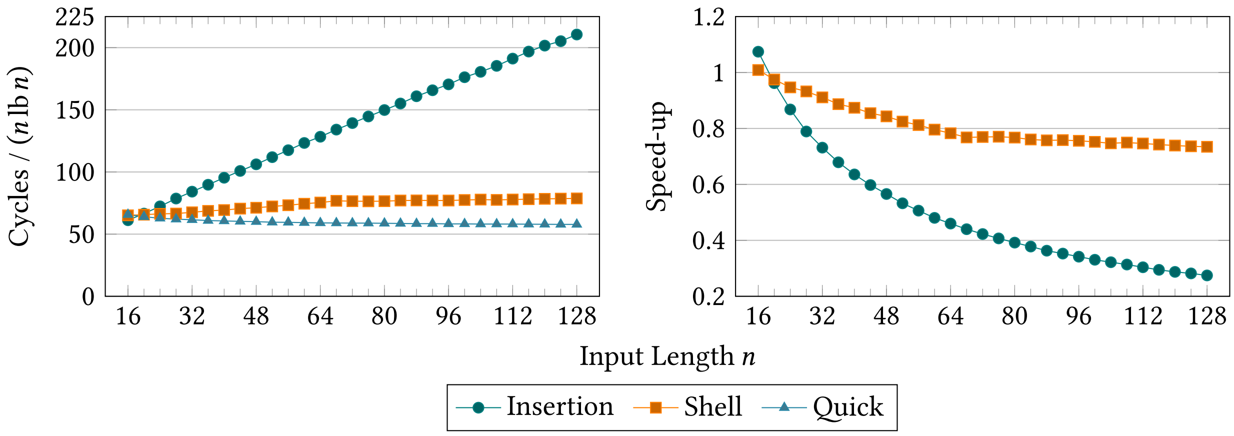


Figure 4: Comparison of InsertionSort, ShellSort and QuickSort. The ShellSort uses the steps sizes $h = (6, 1)$ for $n \leq 64$ and $h = (17, 4, 1)$ elsewise. The speed-ups are with respect to the QuickSort.

QuickSort mit Zufallspivot auch noch einbauen?

verschiedene Verteilungen?

But would pushing the limits of ShellSort even be rewarding? Two issues come up. Firstly, greater input lengths require greater steps — well into the three digits for $n \approx 1000$ [1, 5] — and those in turn require more sentinel values. But the more sentinel values are stored, the less space is available for the actual input array, leading to smaller runs and thus hurting the overall sorting algorithm. Explorative testing suggests that falling back to bounds checking for big steps is too punishing. Secondly, there simply are better alternatives, namely QuickSort (which will be discussed in more detail in the next section). Figure 4 shows that even though ShellSort takes just a fraction of the time InsertionSort takes — apparently achieving a runtime between $\Omega(n\,\mathrm{lb}\,n)$ and $O(n\,\mathrm{lb}^2\,n)$ —, QuickSort beats both from 20 elements onwards. Even QuickSort's standard deviation of 1429 cycles at 128 elements is superior to ShellSort's 2670 cycles. Together with Fig. 2, this means that ShellSort is not worth using at all and will, consequently, not be improved upon in this thesis.

1.3 QuickSort

QuickSort uses partitioning to sort in an expected average runtime of $O(n \log n)$: A pivot element is chosen from the input array, then the input array gets scanned and elements bigger or smaller than the pivot are moved to the right or left of the pivot element, respectively. Finally, QuickSort is used on the left and right partitions.

Base Cases When only a few elements remain in a partition, QuickSort's overhead predominates such that InsertionSort lends itself as fallback algorithm. As Fig. 5 demonstrates, the optimal threshold for switching the sorting algorithm is around 13 elements, netting a speed-up of 30% and more over a QuickSort without fallback algorithm. This low threshold also means that even a simple two-round ShellSort is not worth considering.

Besides falling back to InsertionSort, another base case is imaginable, namely terminating when the partition has a length of 1, 0, or even −1 elements. Realistically speaking, this should not be necessary, because even though the extra check is done with just one additional instruction, it is a rare occurrence and the InsertionSort would terminate after a few instructions anyway. Yet,

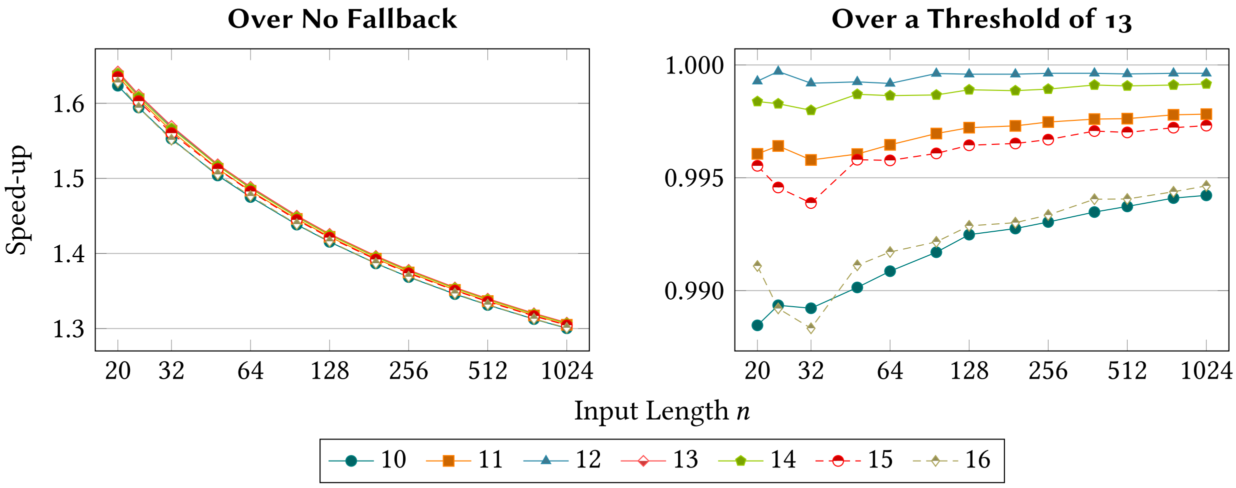


Figure 5: Comparison of QuickSorts with different thresholds for the fallback to InsertionSort, with a QuickSort without fallback algorithm and the fastest QuickSort with a threshold of 13 elements.

there are tremendous consequences for the runtime depending on the exact implementation of the base cases. Since these are likely caused by the compiler, they are laid out in ??.

zurückkehren nach Unterunterabschnitt

Recursion vs. Iteration In theory, the question of whether an algorithm should be implemented recursively or iteratively comes down to convenience. Due to the uniform cost of instructions, putting arguments automatically on the call stack or manually in an array essentially costs the same, as does jumping to the start of a loop and to the start of a function. Furthermore, in case of QuickSort, the compiler turns tail-recursive calls into jumps back to the function start, so that one partition is sorted recursively and the other iteratively. All this would suggest a recursive implementation with less code complexity.

In practice, it comes down to the compilation. Selcouthly, even parts of the algorithms which are independent from the choice between recursion and iteration can be compiled differently, such that there are implementations where iteration is faster than recursion and the other way around. Overall though, iterative implementations tend to be compiled better with superior register usage and less instructions used for the actual QuickSort algorithm. The fastest implementation is indeed an iterative one, even if it beats the fastest recursive implementations — outliers, admittedly — by less than 4%. More details are given in Section 1.3.1.

Pivot Choice Another parameter to tune is the way in which the pivot is chosen. The following were implemented and tested:

- Using the *last element* is the fastest way, requiring zero instructions.
- Choosing the *middle element* is slower than choosing the last one, requiring a calculation of its address and swapping it with the last element so that it can act as sentinel value during partitioning. The upside is that it is more suited for sorted and nearly sorted inputs.
- Taking the *median of three elements*, namely the first, middle, and last one, is even more computationally expensive but increases the chances of choosing a pivot that is neither particularly high nor particularly low.
- A *random element* is most efficiently drawn using an xorshift random number generator and rejection sampling [2].

Luckily, the pivot choice seldom has bearing on the overall compilation, making a comparison

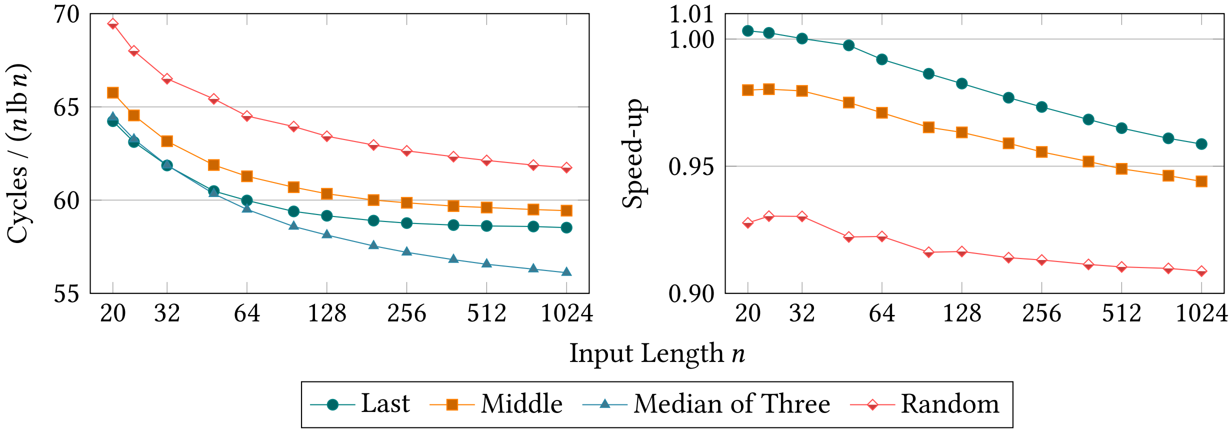


Figure 6: Comparison of QuickSort with different pivot choices. The speed-ups are with respect to the QuickSort with the median of three as pivot choice.

easier. The results are shown in Fig. 6. Choosing the middle element is cheap enough for the runtime to be slowed down by a low single-digit percentage, and the increased pivot quality from choosing the median of three elements more than offsets the cost increase, thus making it the best choice. At 1024 elements, the runtime with a random pivot is 10% worse than with the median of three elements. Since drawing the random index is more than thrice as costly as computing the middle index, a median of three random elements would likely yield even worse times, should one need randomisation. Again, more details are given in Section 1.3.1.

Prioritisation of Partitions After partitioning, in order to minimise the call stack, QuickSort should be used on the smaller of the two partitions first. For code simplicity and to reduce the overhead, no such mechanism was implemented. As shown in Section 1.3.1, the choice between always sorting the left-hand partition or the right-hand partition first can have tremendous effects nevertheless.

1.3.1 Investigating the Compilation

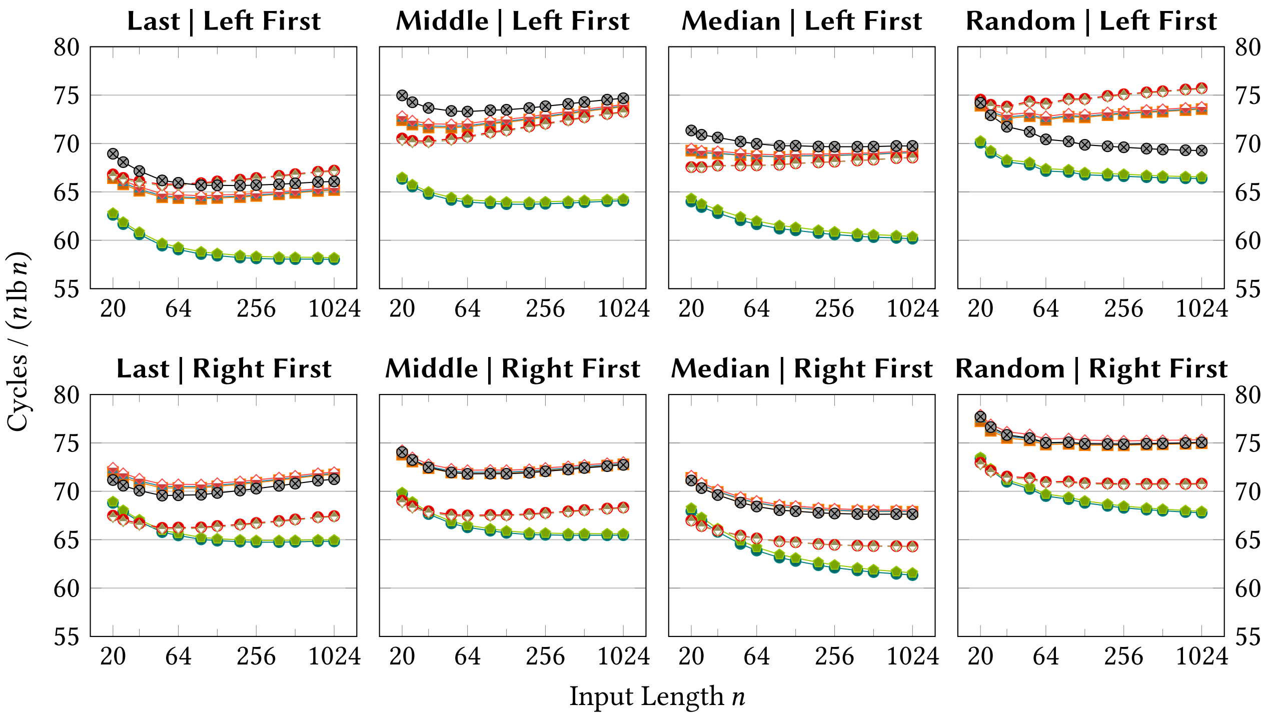
The quality of the compilation and thus the real performance of QuickSort is erratic to such an extent that one implementation variant may see a speed-up of 25% over another one even with the same pivot choice although virtually none would be expected. As hinted in the preceding paragraphs, this raises the need for a benchmark suite with the following parameters: base case handling, recursion/iteration, pivot choice, and partition prioritisation. Before the results are discussed, the first parameter shall be explained in more depth.

Besides falling back to InsertionSort if 13 elements remain ('treshold undercut'), another base case is imaginable, namely a full termination if 1, 0, or −1 elements remain ('trivial length'). Theoretically, it should not be needed to check for trivial lengths because even though it is doable with just one additional instruction, such small partitions are rare and the InsertionSort would terminate after a few instructions anyway. Nonetheless, its inclusion or exclusion can have significant impacts. The following Implementations were tested:

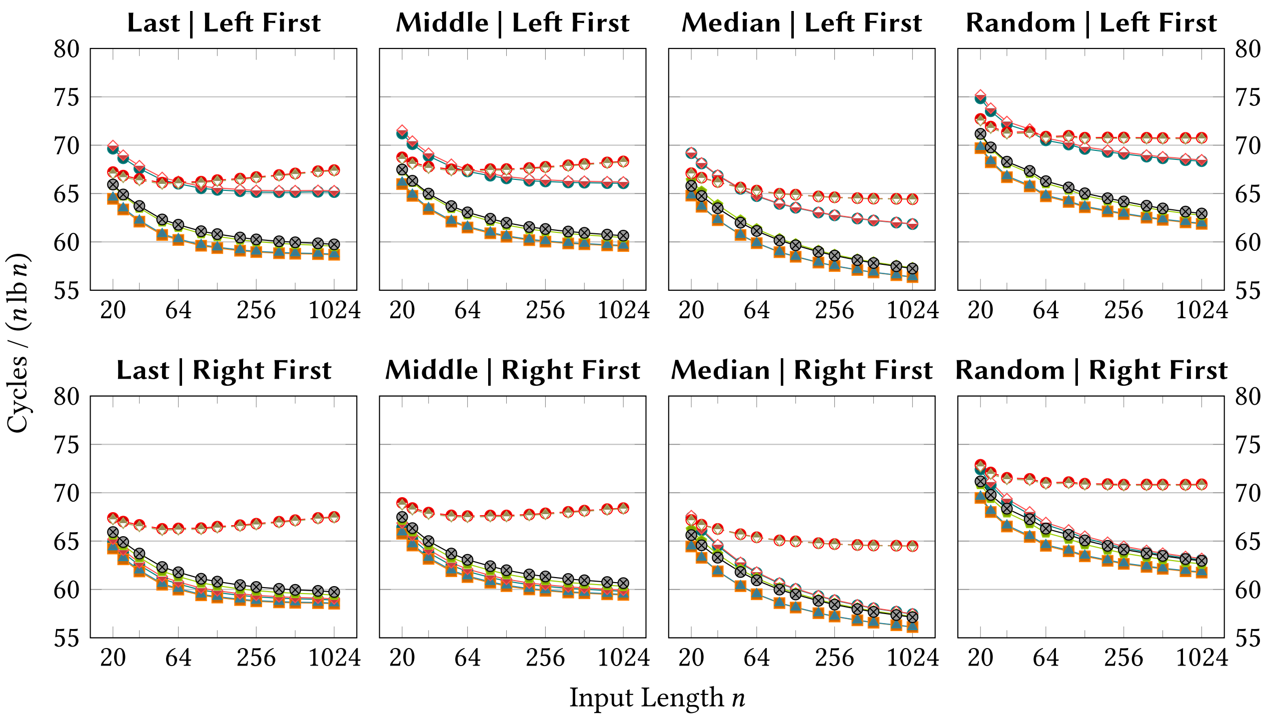
1. If the length is trivial, terminate. If not and if the threshold is undercut, sort with InsertionSort. Otherwise, sort with QuickSort and use QuickSort on both partitions.
2. If the threshold is undercut, check if the length is trivial and terminate or sort with InsertionSort, respectively. Otherwise, sort with QuickSort and use QuickSort on both partitions.
   - This Implementation significantly reduces the number of checks for trivial length.

3. If the threshold is undercut, sort with InsertionSort. Otherwise, sort with QuickSort and use QuickSort on both partitions.
   - This Implementation forgoes the check for a trivial length completely, at the cost of unneeded InsertionSorts.
4. If the threshold is undercut, sort with InsertionSort. If not and if the length is trivial, terminate. Otherwise, sort with QuickSort and use QuickSort on both partitions.
   - This Implementation, while nonsensical from a logical point of view, gives the compiler an explicit guarantee that the partitioning loop does not end immediately.
5. If the threshold is undercut, sort with InsertionSort. Otherwise, sort with QuickSort. Then check for either partition if its length is trivial and use QuickSort if not.
   - This Implementation, as well as the next two, gets rid of some unneeded uses of QuickSort. In the recursive case, these Implementations lose the property of being tail-recursive.
6. Sort with QuickSort. Check for either partition if the threshold is undercut and use InsertionSort or QuickSort, respectively.
7. Sort with QuickSort. Check for either partition if its length is trivial or if the threshold is undercut and use InsertionSort, QuickSort, or nothing, respectively.
8. If the threshold is undercut, terminate. Otherwise, sort with QuickSort and use QuickSort on both partitions. After all QuickSorts are done, sort the whole input array with InsertionSort.
   - This Implementation always does one pass of InsertionSort. For example, the other Implementations do roughly 90 at 1024 elements.

All results are shown in Fig. 7. When using recursion, Impl. 1 and 5 perform the best, especially for large inputs. Their



(a) Recursive Approach



(b) Iterative approach

Figure 7: Comparison of the different implementations (1–8) of QuickSort for all possible pivot choices. In the first rows, the left partitions are sorted before the right ones, while it is the reverse in the second rows.

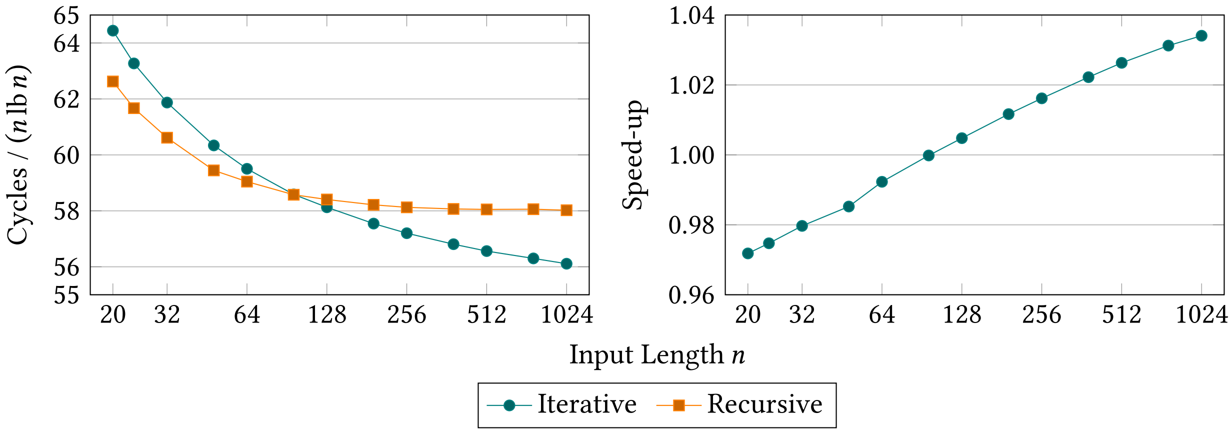


Figure 8: Comparison of the fastest recursive and iterative QuickSorts (cf. Section 1.3.1). The actual algorithm is compiled the very same in both cases, so that time differences are only due to the way QuickSort is applied to the partitions.

nicht mehr wegen des Pivots!

2 References

[1] Marcin Ciura. 'Best Increments for the Average Case of Shellsort'. In: *Fundamentals of Computation Theory*. Ed. by Rūsiņš Freivalds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 106–117. DOI: 10.1007/3-540-44669-9_12. URL: https://web.archive.org/web/20180923235211/http://sun.aei.polsl.pl/~mciura/publikacje/shellsort.pdf (visited on 24/05/2024).

[2] Lukas Geis. *Random Number Generation in the Pim-Architecture*. Research Project Report. Version 8c11f1f. Goethe University Frankfurt, 2024. 13 pp. URL: https://github.com/lukasgeis/upmem-rng/blob/main/report/report.pdf (visited on 19/05/2024).

[3] Ying Wai Lee. *Empirically Improved Tokuda Gap Sequence in Shellsort*. 21st Dec. 2021. arXiv: 2112.11112 [cs.DS].

[4] Donald L. Shell. 'A high-speed sorting procedure'. In: *Commun. ACM* 2 (1959), pp. 30–32. URL: https://api.semanticscholar.org/CorpusID:28572656.

[5] Oscar Skean, Richard Ehrenborg and Jerzy W. Jaromczyk. *Optimization Perspectives on Shellsort*. 1st Jan. 2023. arXiv: 2301.00316 [cs.DS].