Contents

Todo list

Architektur

Speicherzugriffe (memcpy, mram_read ...)

triple buffer

We took our cue from Axtmann et al. [1] for the choice of distributions.

Sorted The numbers from 0 to $n-1$ are generated in ascending order.

Reverse Sorted The numbers from 0 to $n-1$ are generated in descending order.

Almost Sorted First, the numbers from 0 to $n-1$ are generated in ascending order, then, $\lfloor\sqrt{n}\rfloor$ random pairs are sequentially drawn and swapped. There are no checks on whether pairs have common elements.

Uniform Each number is drawn independently and uniformly from the range $[0, 2^{31}-1]$.

Narrow Uniform Each number is drawn independently and uniformly from the range $[0, n-1]$.

Zipf's Each number is drawn independently from the range $[1, 100]$, with each value k drawn with a probability proportional to $1/k^{0.75}$.

Normal Each number is drawn independently according to a normal distribution with mean $\mu = 2^{31}$ and standard deviation $\sigma = \min(1, \lfloor n/8 \rfloor)$.

Bisher nicht. Als Ersatz für die Normalverteilung?

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 shown in this section was conducted on a uniform input distribution with each 32-bit integer, and the default configurations of the sorting algorithms were as follows:

InsertionSort using one explicit 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

Further measurements can be found in Appendix A.

1.1. InsertionSort

This stable sorting algorithm works by moving the ith element leftwards as long as its left neighbour is bigger, assuming that the elements at the indices 0 to $i-1$ are already sorted. Its asymptotic runtime is bad, reaching $O(n^2)$ not only in the worst case but also in the average case, where each of the $\binom{n}{2}$ pairs of input elements is in wrong order and will be swapped at some point in the execution with probability 50%. Nonetheless, InsertionSort does have some saving graces: 1. If the input array is mostly or even fully sorted, the runtime drops down to $O(n)$. 2. It works in-place, needing only $O(1)$ additional space. 3. Its program code is short, lending itself to inlining. 4. The overhead is small. Especially the last two points make InsertionSort a good fall-back algorithm for asymptotically better sorting algorithms to use on small subarrays.

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 at about 30% for short inputs (Fig. 1).

Setting an *explicit* sentinel value can be omitted by using *implicit* sentinel values. At the start of each round, one can check if the element at index 0 is at least as small as the element at index i. If yes, the former is a sufficient sentinel value, and InsertionSort can proceed as normal. If not, the latter must be the minimum of the first $i+1$ elements and, therefore, can be moved immediately to the front.

The runtimes of the explicit and implicit InsertionSorts can be compared in the Figs. 1, 11 and 12. For most input distributions, the implicit InsertionSort is logically a bit slower, effectively adding one check instruction to each round. An outlier, however, are the reverse sorted inputs. For 32-bit numbers (Fig. 11), the implicit version is up to 45% slower than the explicit one. This comes at a surprise since both versions effectively execute the same loop body while shifting everything one index backwards, with only the loop condition being different. Due to the uni-cost model, a value check on whether the preceding element is smaller (explicit) and an address check on whether the preceding position is the start of the array (implicit) should take the same amount of time. Yet, even the InsertionSort not relying on sentinel values surpasses

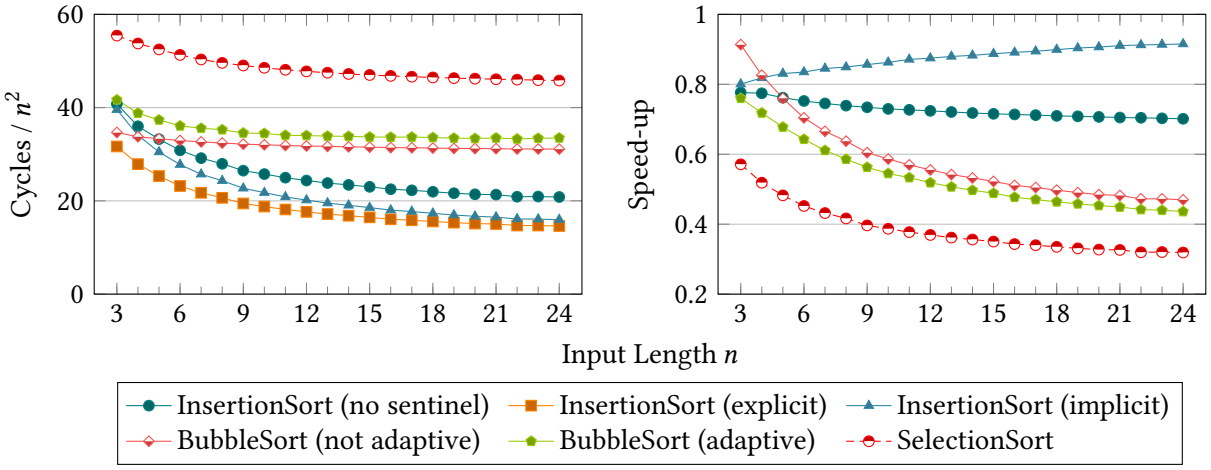


Figure 1: Comparison of sorting algorithms with a runtime in $O(n^2)$. The adaptive BubbleSort terminates prematurely if no changes were made to the input array during an iteration. The speed-ups are with respect to the InsertionSort relying on explicit sentinel values.

the implicit InsertionSort, although doing both value checks and address checks! For 64-bit numbers (Fig. 12), the implicit InsertionSort would be expected to perform better than the explicit one, considering that a value check now takes two instructions and an address check still only one. Nonetheless, the two InsertionSorts tie.

Investigating the Compilation

At times, the quality of the compilation nosedives, and InsertionSort serves as great example. But before we focus on the poor performance of the implicit InsertionSort, let us first look at something far more basic.

When sorting an element, all previous elements must already be in sorted order; an exemplary implementation of this is shown in Fig. 2a. Obviously, the first element alone is already sorted, so InsertionSort will not perform any changes and proceed to the second element. In order to get rid of some loop overhead, it would make sense to let InsertionSort start at the second element, as in Fig. 2b. Surprisingly, starting at the first element yields a runtime of 4166 cycles at 16 elements, whereas starting at the second yields a runtime of 4266 cycles. The same happens if, in Fig. 2b, one keeps `*i = start` and instead uses `curr = ++i`.

Looking at the compilation reveals the reason: In the faster case, the pointer `pred` is optimised away and, in its stead, the pointer `curr` and an offset of `-4` is used. In the slower case, the pointer `pred` is *sometimes* used with an offset to get the true value of `curr`. Two registers are used to store the addresses of `curr` and `pred` at different points in time, resulting in one additional instruction at the start of each round, nullifying any gain from starting with the second element. This is a consequence of reusing the register of the `start` pointer for `pred` instead of for `i`, whose incremented value is put into the additionally used register.

Multiple workarounds exist to sidestep this problem. One workaround is to take the faster code of Fig. 2a and add inline assembler to the start which increments the register holding the

```
1  void insertion_sort(int *start, int *end) {
2    int *curr, *i = start;
3    while ((curr = i++) <= end) {
4      int to_sort = *curr;
5      int *pred = curr - 1;
6      while (*pred > to_sort) {
7        *curr = *pred;
8        curr = pred--;
9      }
10     *curr = to_sort;
11   }
12 }
```

(a) Start at the first element and with predecessor pointer.

```
1  void insertion_sort(int *start, int *end) {
2    int *curr, *i = start + 1;
3    while ((curr = i++) <= end) {
4      int to_sort = *curr;
5      int *pred = curr - 1;
6      while (*pred > to_sort) {
7        *curr = *pred;
8        curr = pred--;
9      }
10     *curr = to_sort;
11   }
12 }
```

(b) Start at the second element and with predecessor pointer.

```
1  void insertion_sort(int *start, int *end) {
2    int *curr, *i = start;
3    while ((curr = i++) <= end) {
4      int to_sort = *curr;
5      while (*(curr-1) > to_sort) {
6        *curr = *(curr-1);
7        curr--;
8      }
9      *curr = to_sort;
10   }
11 }
```

(c) Start at the first element and without predecessor pointer.

```
1  void insertion_sort(int *start, int *end) {
2    int *curr, *i = start + 1;
3    while ((curr = i++) <= end) {
4      int to_sort = *curr;
5      while (*(curr-1) > to_sort) {
6        *curr = *(curr-1);
7        curr--;
8      }
9      *curr = to_sort;
10   }
11 }
```

(d) Start at the second element and without predecessor pointer.

Figure 2: Four different implementations of InsertionSort. Figures 2a and 2c are compiled the same. Figures 2b and 2d are compiled differently.

`start` pointer. While this does work for the explicit InsertionSort, the other two versions of InsertionSort need to know the original starting address later on and, thus, actually set the address of `i` rather late; adding inline assembler proves fare more difficult as a consequence. And as InsertionSort is to be used as fallback algorithms by other functions, which might also need to keep the original value of `start`, inline assembler is a bad choice even for the explicit InsertionSort.

Another workaround is the usage of a wrapper function calling InsertionSort with the arguments `start + 1` and `end`. This works quite well: First, the register holding `start` is incremented and, then, follows the inlined code from the actual InsertionSort. Doing so makes the runtime drop from 4166 cycles down to 4101 cycles.

Recall how in the faster version (Fig. 2a), the pointer `pred` is always deduced from the pointer `curr`. This is manually enforced in Figs. 2c and 2d, where `pred` is replaced with `curr - 1`. As a consequence, the code of Fig. 2c compiles the very same as the one of Fig. 2a, while Fig. 2d

yields the same compilation as the wrapper functions. This workaround is clearly the best of the three and, hence, the one used in the whole code base.

Alas, the struggle with the compiler endeth not herewith. A deeper look into the compilation reveals the following three instructions: `move r3, r0; jleu r4, r2, .LBB1_4; move r3, r0`. Without delving further into their meaning — the second `move r3, r0` is unneeded as it is impossible to jump directly to it and, apparently, it does also not set any flags. Copying the whole assembler code and injecting it as inline assembler but with this second `move r3, r0` removed pushes the runtime down to just 3961 cycles while still sorting correctly; the gain is even greater for input distributions other than the uniform one. New issues, especially for inlining, are introduced, though, and we deem a proper assembly implementation as out of scope for this thesis.

At this point, it should have become apparent that the poor performance of the implicit InsertionSort is also due to bad compilation. We refrain from going into details because we failed to find an easy fix and because the implicit InsertionSort is not used in the rest of the code base. Thence, a sole 'InsertionSort' refers to the version relying on explicit sentinel values henceforth. Still, the idea of implicit sentinel values will be of use for ShellSort in Section 1.2.

Note. Other known simple sorting algorithm with a runtime complexity similar to InsertionSort are SelectionSort and BubbleSort. The asymptoticity, however, hides much higher constant factors such that InsertionSort should always be preferred (cf. Figs. 1, 11 and 12).

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 [5], gets around this by doing multiple passes of InsertionSort with different step sizes: In pass p with step size h_p, the input array is divided into the subarrays of indices $(i, h_p{+}i, 2h_p{+}i, \dots)$ for $i = 0, \dots, h_p - 1$ which then get sorted individually through InsertionSort. The step sizes get smaller each pass, 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 do 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 [4, 6] and depends on the cost of the operation types (comparing, swapping, looping).

Let us first focus on small input arrays where only two passes with step sizes h_1 and 1 suffice. The previous results on InsertionSort suggest that the fastest ShellSort should make use of h_1 sentinel values. Figures 3, 13 and 14 show that, with the exception of the ShellSort with step size $h_1 = 2$, the additional passes start to pay off at around 16 elements for both 32-bit and 64-bit values with the fully random input distributions, reaching a speed-up of around 15% at around 24 elements. In case of the reverse sorted input, the speed-up is practically always positive even for very small inputs, reaching between 50% and 110% at around 24 elements.

When moving to greater input lengths (Figs. 4 and 15 to 18), the differences in performance between the two-tier ShellSorts become more pronounced; Between 48 and 64 elements, three passes get worthwhile to consider. On the one hand, the results are in accordance with Ciura [2] who, for 128 elements, determined $h = (1, 9)$ to be the optimal pair and $h = (17, 4, 1)$ to be the

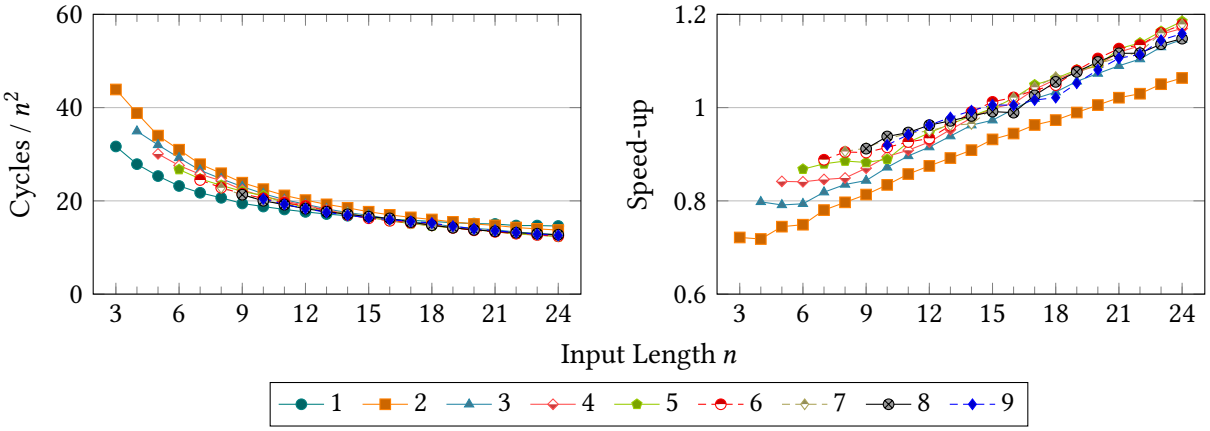


Figure 3: Comparison of InsertionSort (1) and various two-tier ShellSorts (2–9), whose step sizes h_1 are indicated by their label. The speed-ups are with respect to the InsertionSort.

optimal triplet. On the other hand, the gain from doing three passes is far smaller: While Ciura calculated an average speed-up of 33% over doing two passes, while it is only 16% on a DPU. In opposition to his results, this also makes it unlikely that doing four passes would already net any gain at this input length. Without access to Ciura's original code, giving a satisfactory explanation for the discrepancy is hard, however.

But would pushing the limits of ShellSort even be rewarding? Firstly, greater input lengths require greater steps — well into the three digits for $n \approx 1000$ [2, 6] — and those in turn require more sentinel values. Implicit sentinel values could provide relief since the slow-down from implicitness should trend to zero for large inputs, as was the case for InsertionSort.

'Secondly, there simply are better alternatives, namely QuickSort (which will be discussed in more detail in the next section). **??** shows that even though ShellSort takes just a fraction of the time InsertionSort takes — apparently achieving a runtime between $\Omega(n \operatorname{lb} n)$ and $O(n \operatorname{lb}^2 n)$ —, QuickSort beats both from 20 elements onwards. Even QuickSort's standard deviation of 1432 cycles at 128 elements is superior to ShellSort's 2670 cycles. Together with Fig. 3, 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

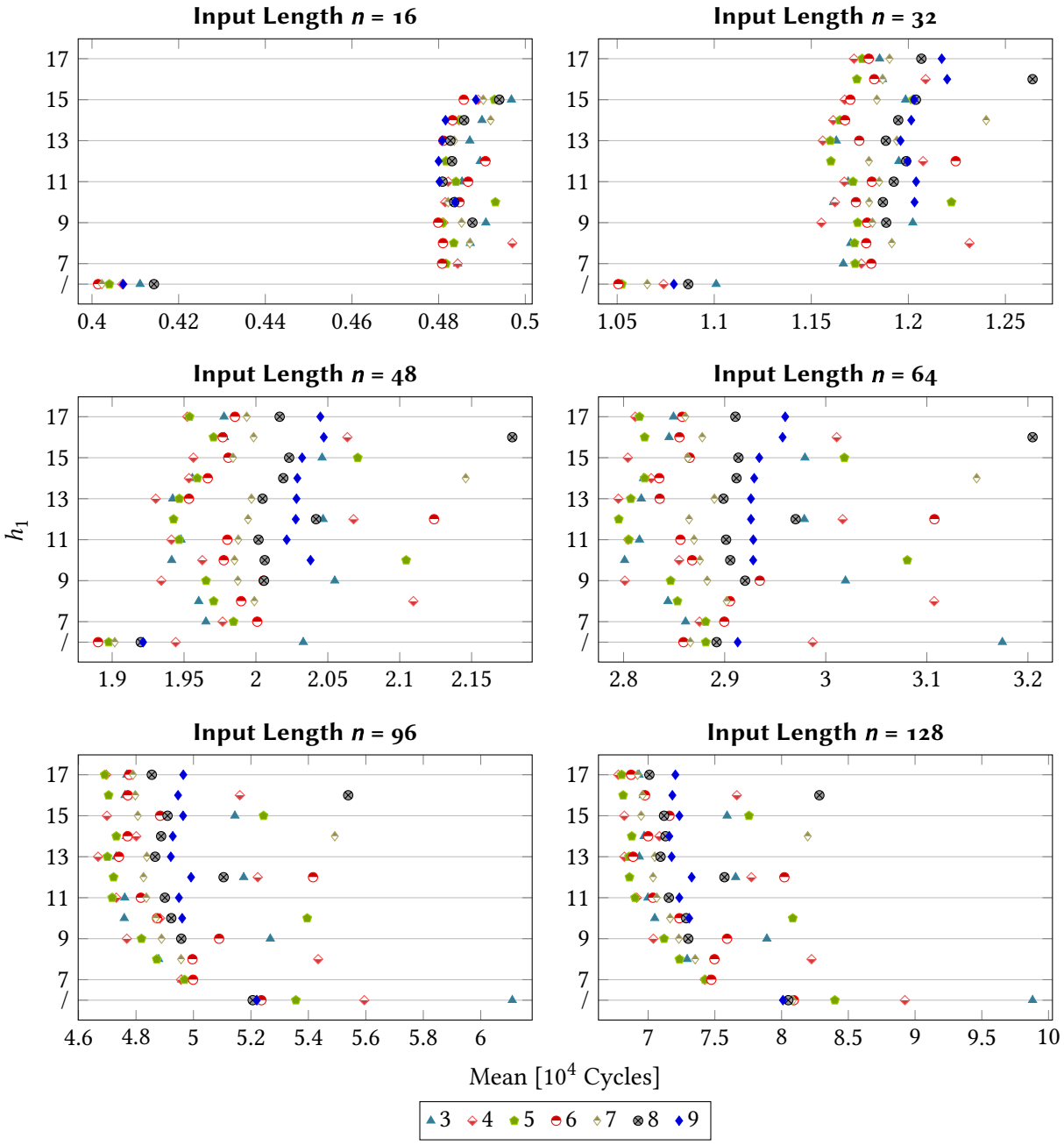


Figure 4: Runtimes of ShellSorts with two passes (/) and three passes (7–17). The coloured symbols encode the step size h_1 for two-tier ShellSorts and the step size h_2 for three-tier ShellSorts. For the latter, the step size h_1 is noted on the y-axes.

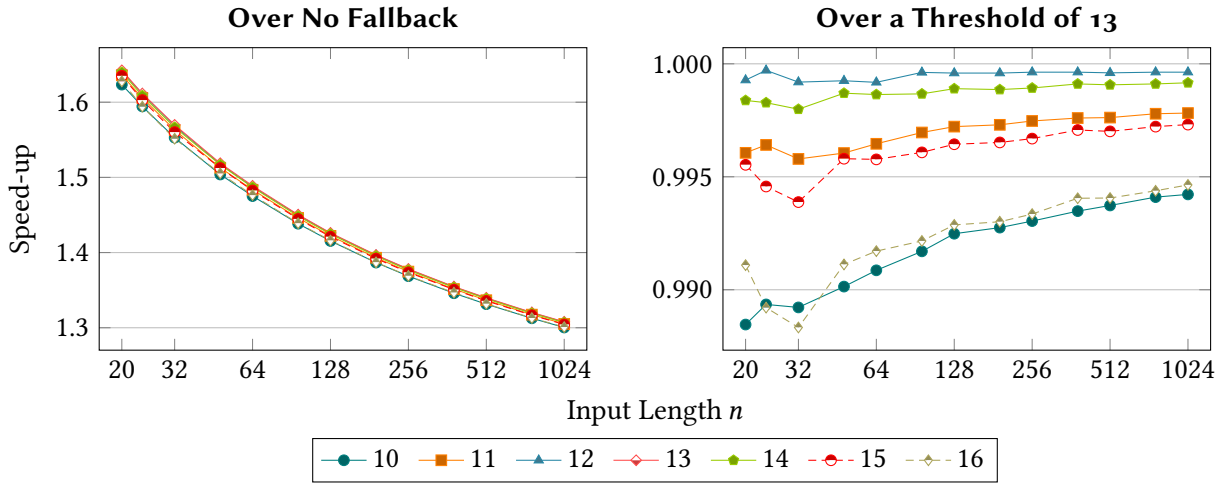


Figure 5: Comparison of QuickSorts with different thresholds for the fallback to InsertionSort, with a QuickSort without fallback algorithm and the fastest QuickSort with a threshold of 13 elements.

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, 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 **??**.

zurück-kehren nach Un-terunterab-schnitt

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.

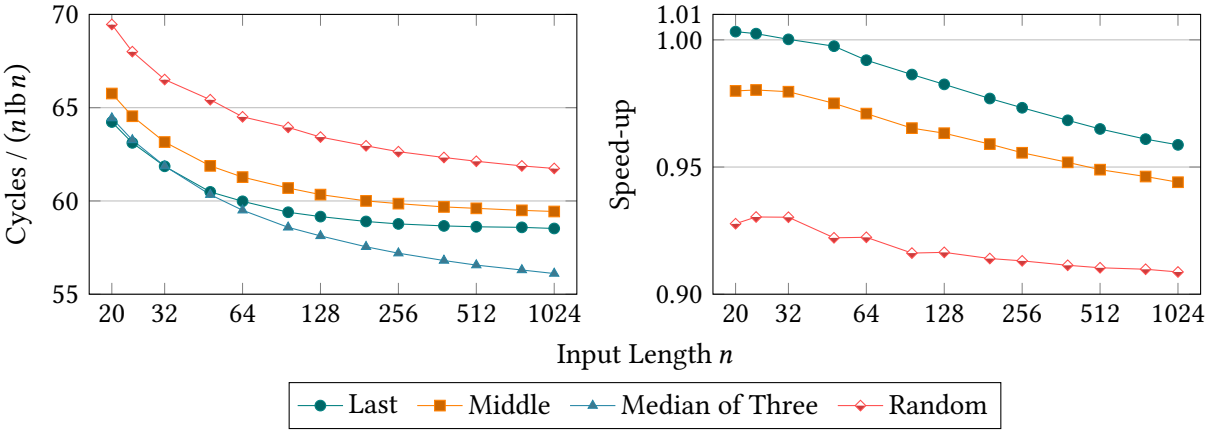


Figure 6: Comparison of QuickSort with different pivot choices. The speed-ups are with respect to the QuickSort with the median of three as pivot choice.

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 [3].

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

Stimmt nicht!

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 compilations are fundamentally the same, including the conversion of the second recursive call into a jump back to the function start. All other Implementations fare vastly worse. Common occurrences are …

- … one more instruction in the loop finding the next element to move to the right, …
- … one more instruction after such an element has been found, …
- … more stores and loads when entering and leaving the function.

Ich kann mir leider nicht alles erklären. Als Beispiel habe ich die Kompilate von Impl. 1 / Recursive / Last für Left First und Right First hochgeladen (die verkürzten Varianten besitzen nur noch Befehle und Sprungmarken). Ersteres ist ja die schnellste rekursive Variante, während letzteres deutlich schlechter abschneidet. Dennoch sehe ich bei der langsameren Variante keinen fundamental anderen Algorithmus. Je Funktionsaufruf kommen ≈3 Extra-Aufrufe hinzu (bei insgesamt ≈104 rekursiven und ≈104 'Endaufrufen' bei 1024 Elementen), was fast 70 000 Takte Unterschied nicht erklären kann.

1.4. MergeSort

MergeSort repeatedly compares two sorted subarrays and merges them into a bigger sorted array in time $\Theta(n \log n)$. Unlike QuickSort, this runtime is guaranteed. Furthermore, the sorting is stable naturally.

A simple but fast implementation of MergeSort writes all merged runs to an output array, raising the need for additional space for n elements ('Full space'). Since in this implementation, the arrays from which it is read and to which it is written switch each round, the final sorted array may not be saved where the input array was. Thus, a final round with a write-back to the original position is sometimes needed.

A slightly more sophisticated implementation needs additional space for only $n/2$ elements ('Half space'): When two adjacent runs are to be merged, the first one can be copied to an auxiliary array. Then, the copy and the second run are merged to the position of the first run. As a side effect, no write-back is ever needed and, additionally, the merging of two runs can be terminated prematurely once the last element of the copied run is merged since the last elements of the other run are already in place. As a consequence, flushes will only be performed on at most half of the runs.

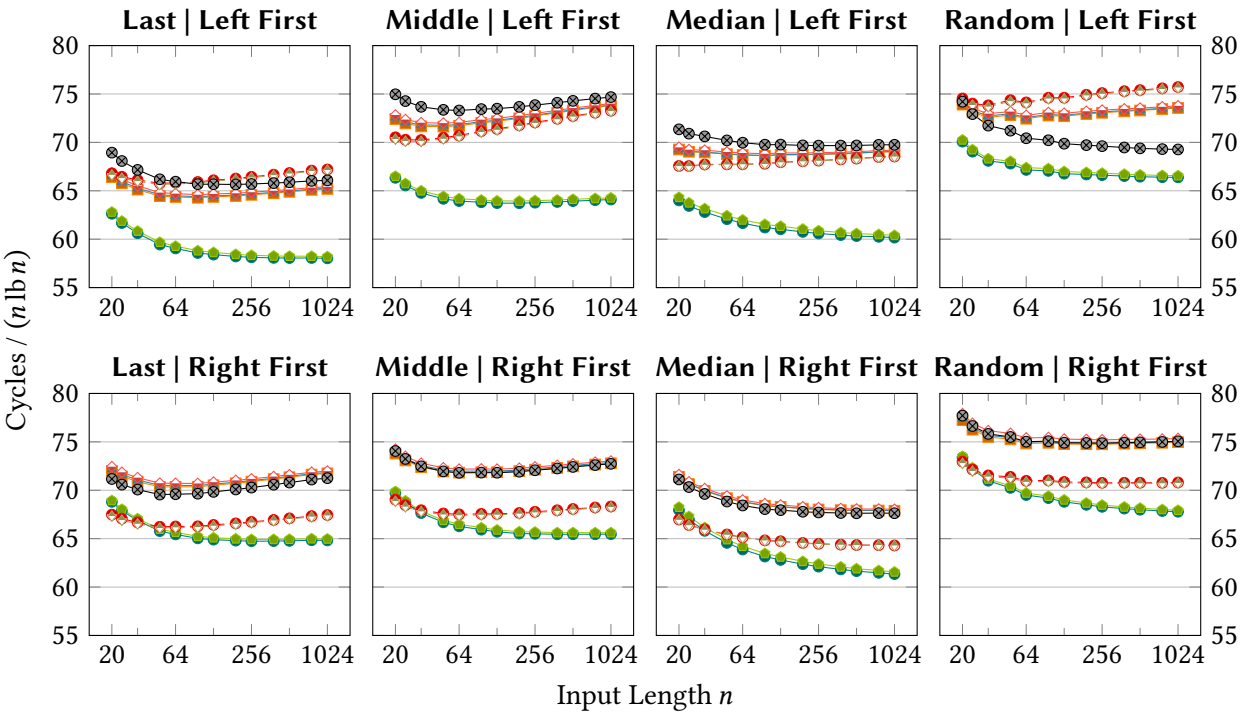
Instead of starting by merging runs of length 1, it is once again beneficial to fall back to simpler algorithms, namely InsertionSort and ShellSort. Unlike QuickSort, where partitions naturally acted as sentinels for their neighbours, it is necessary to temporarily place sentinels values in front of each starting run and later restore the original values of the preceding run. The optimal length for the starting runs depends on the length of the whole array, since it governs the number of rounds of merging, but lengths of 32 elements and 64 elements for the full space and half space MergeSorts, respectively, pose a good compromise, as suggested by Fig. 9. In both cases, a ShellSort with the step sizes $h = (4, 1)$ fares the best.

Warum? Widersprüchlich!

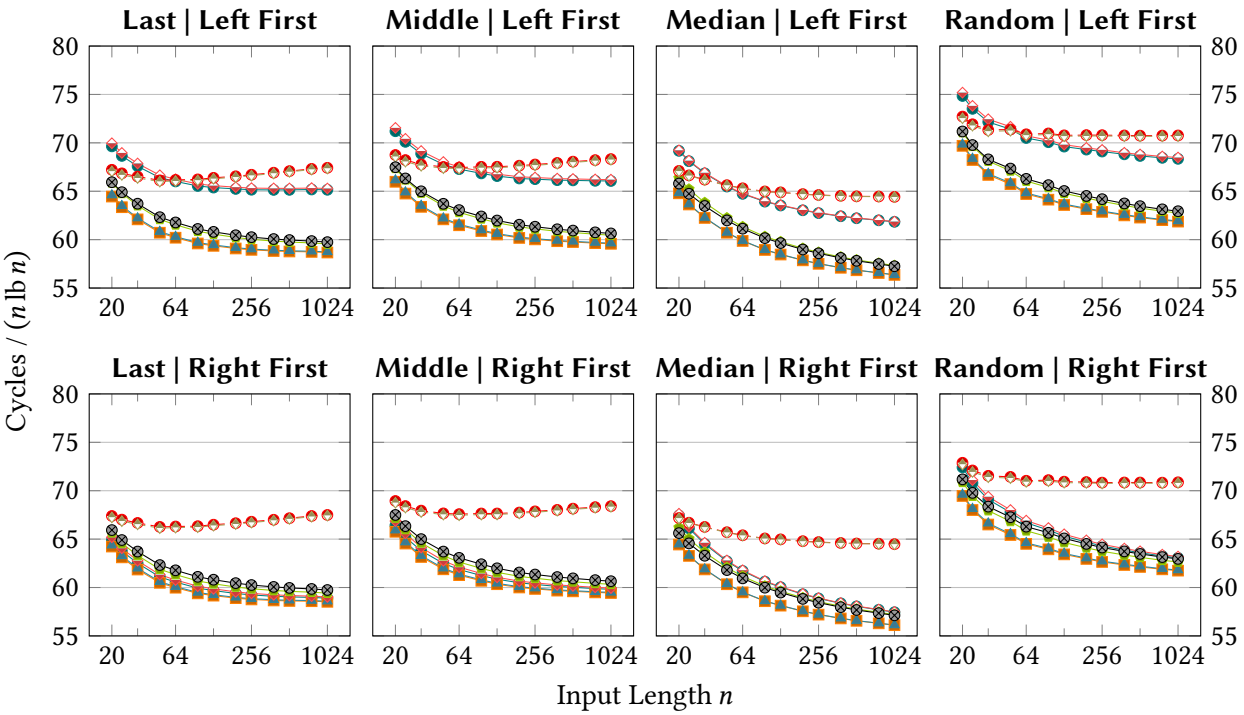
Note. Yet again, the compiler shows unforeseen behaviour. For example, when sorting the starting runs, a ShellSort with step sizes $h = (4, 1)$ may be used. For the very first run,

Compilation is worsened by:

- return, falls nur ein Start-Run
- kein Wächterwechsel beim ersten run



(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-hand partitions are sorted before the right-hand ones, while it is the reverse in the second rows.

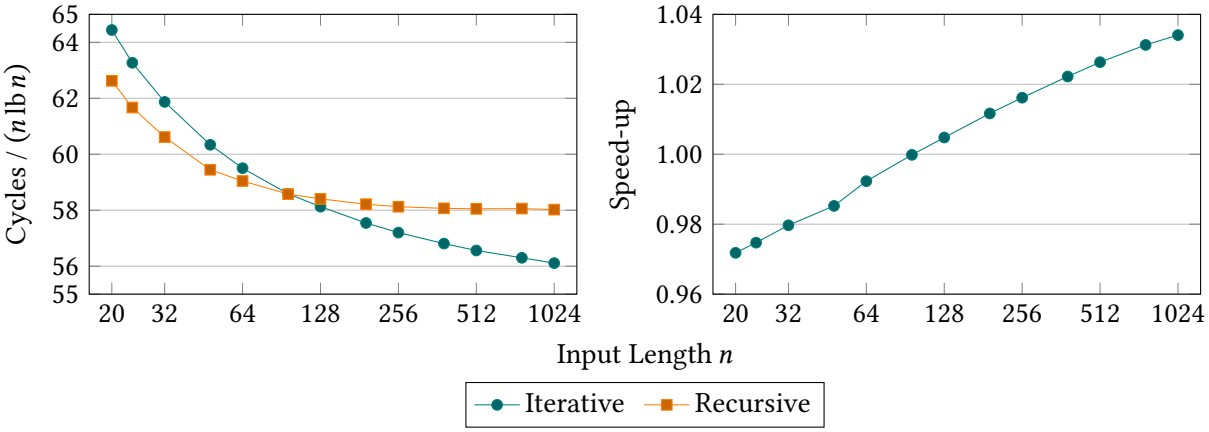


Figure 8: Comparison of the fastest recursive and iterative QuickSorts (cf. Section 1.3.1). The actual algorithm is compiled the very same in both cases, so that time differences are only due to the way QuickSort is applied to the partitions.

nicht mehr wegen des Pivots!

1.4.1. Comparison with QuickSort

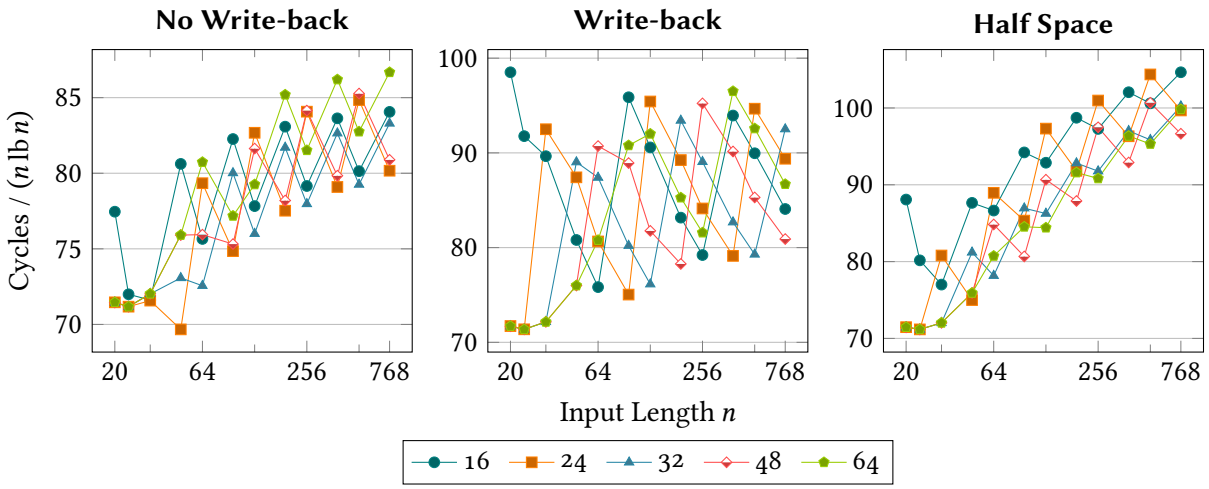


Figure 9: Comparison of MergeSorts, which need an auxiliary array of length either n ('No Write-back' / 'Write-back') or $n/2$ ('Half Space'), for different lengths of the starting runs. For length 16, they fell back to InsertionSort. For lengths 24 to 64, they fell back to a ShellSort with the step sizes $h = (4, 1)$.

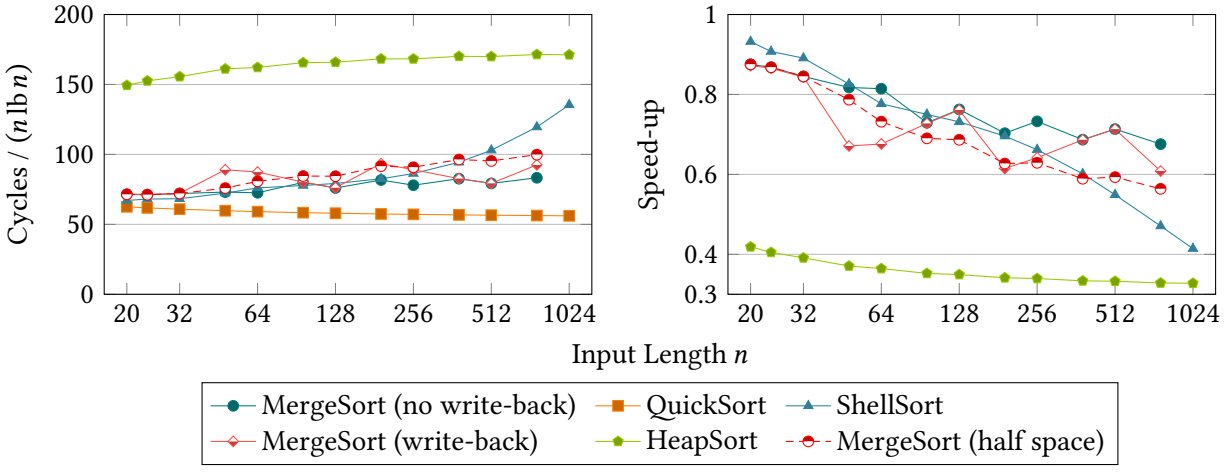


Figure 10: Comparison of MergeSort, HeapSort, ShellSort, and QuickSort. Due to MergeSort's increased space requirements, its runtime was measured only for up to 768 elements. The ShellSort uses the step sizes from **??**, which are unoptimised for large input sizes. The speed-ups are with respect to the QuickSort.

A. Further Measurements on Sorting with One Tasklet

A.1. InsertionSort

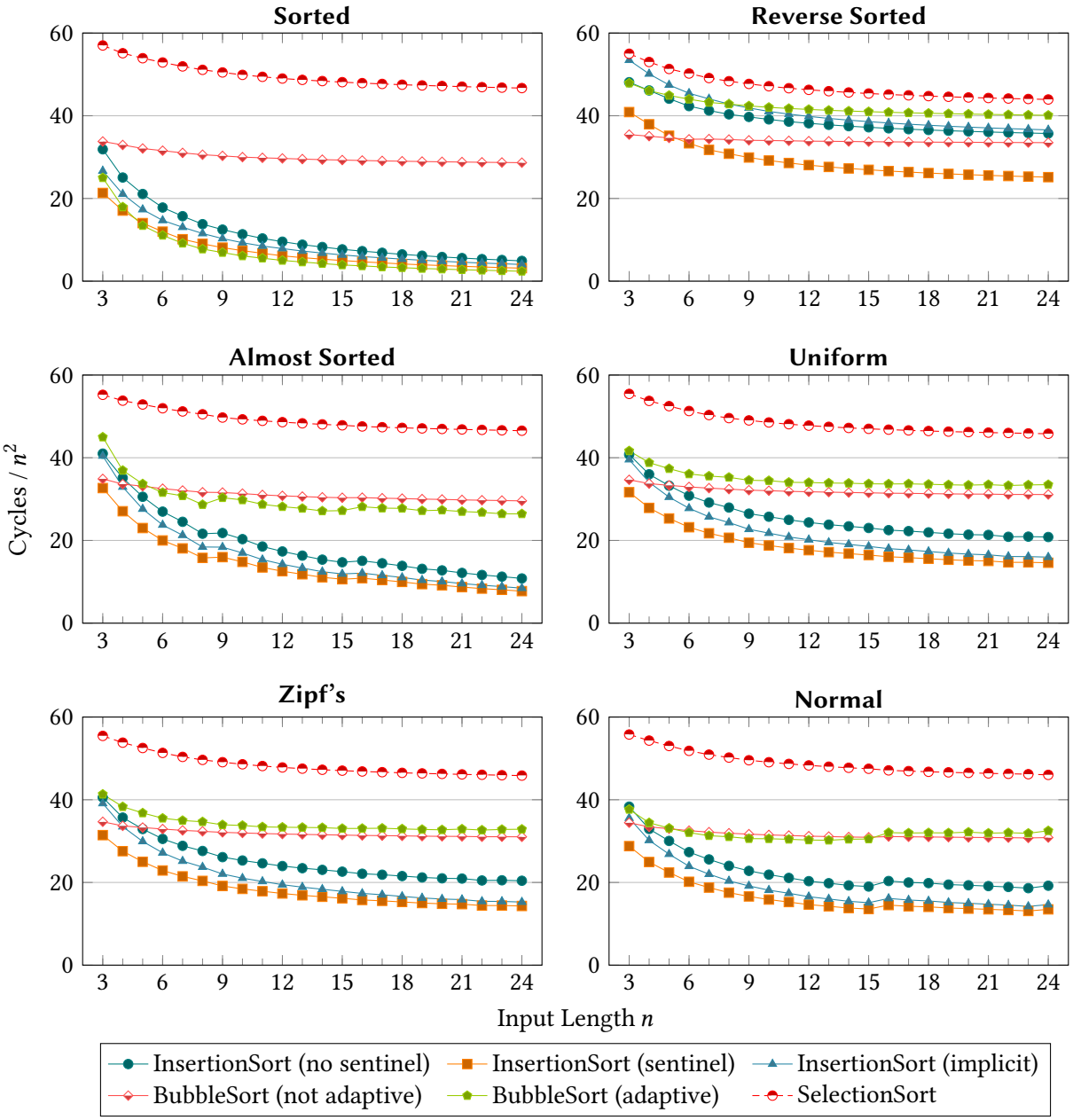


Figure 11: A continuation of Fig. 1 with more input distributions. The data type is 32-bit unsigned integers.

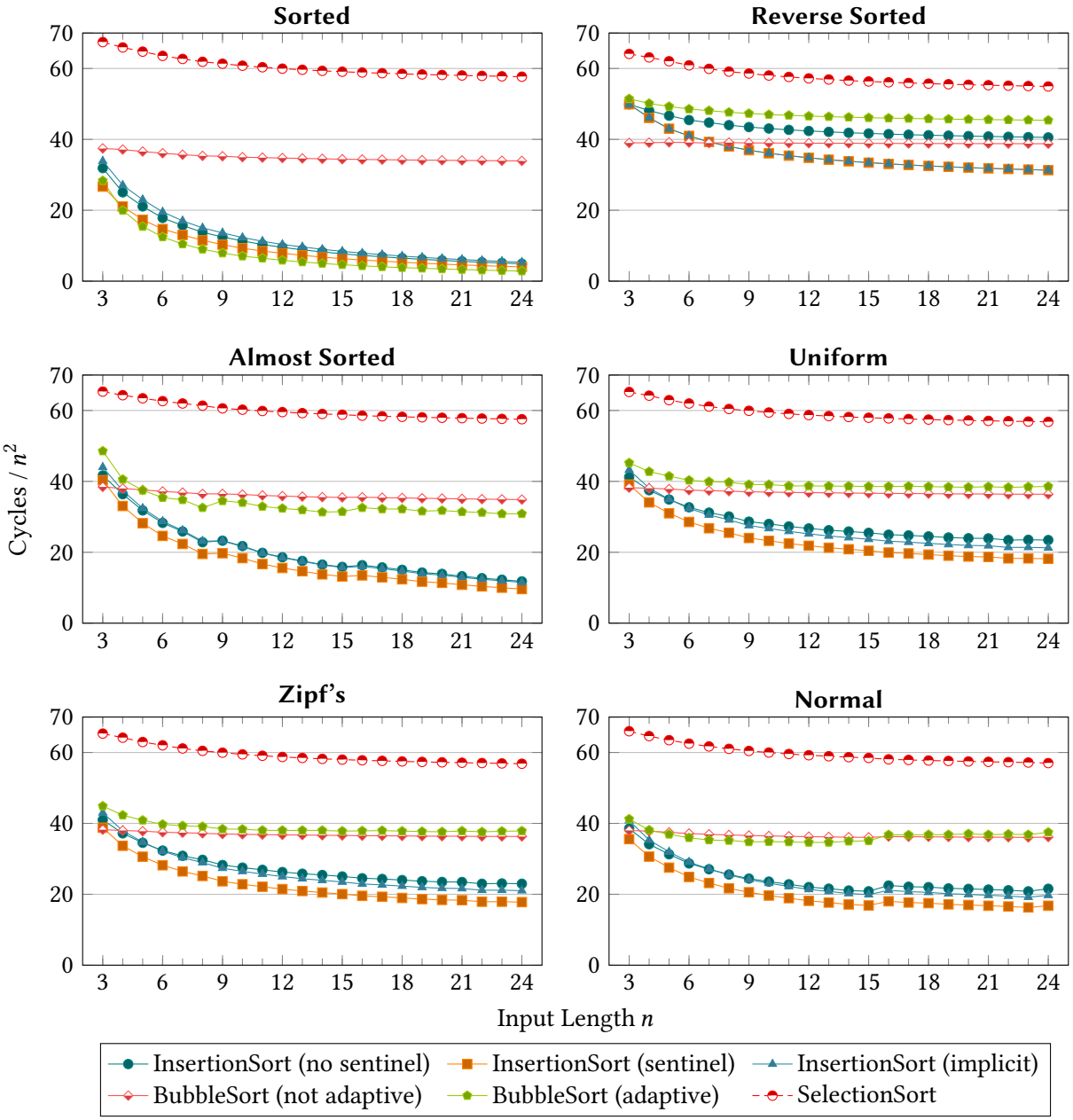


Figure 12: A continuation of Fig. 1 with more input distributions. The data type is 64-bit unsigned integers.

A.2. ShellSort

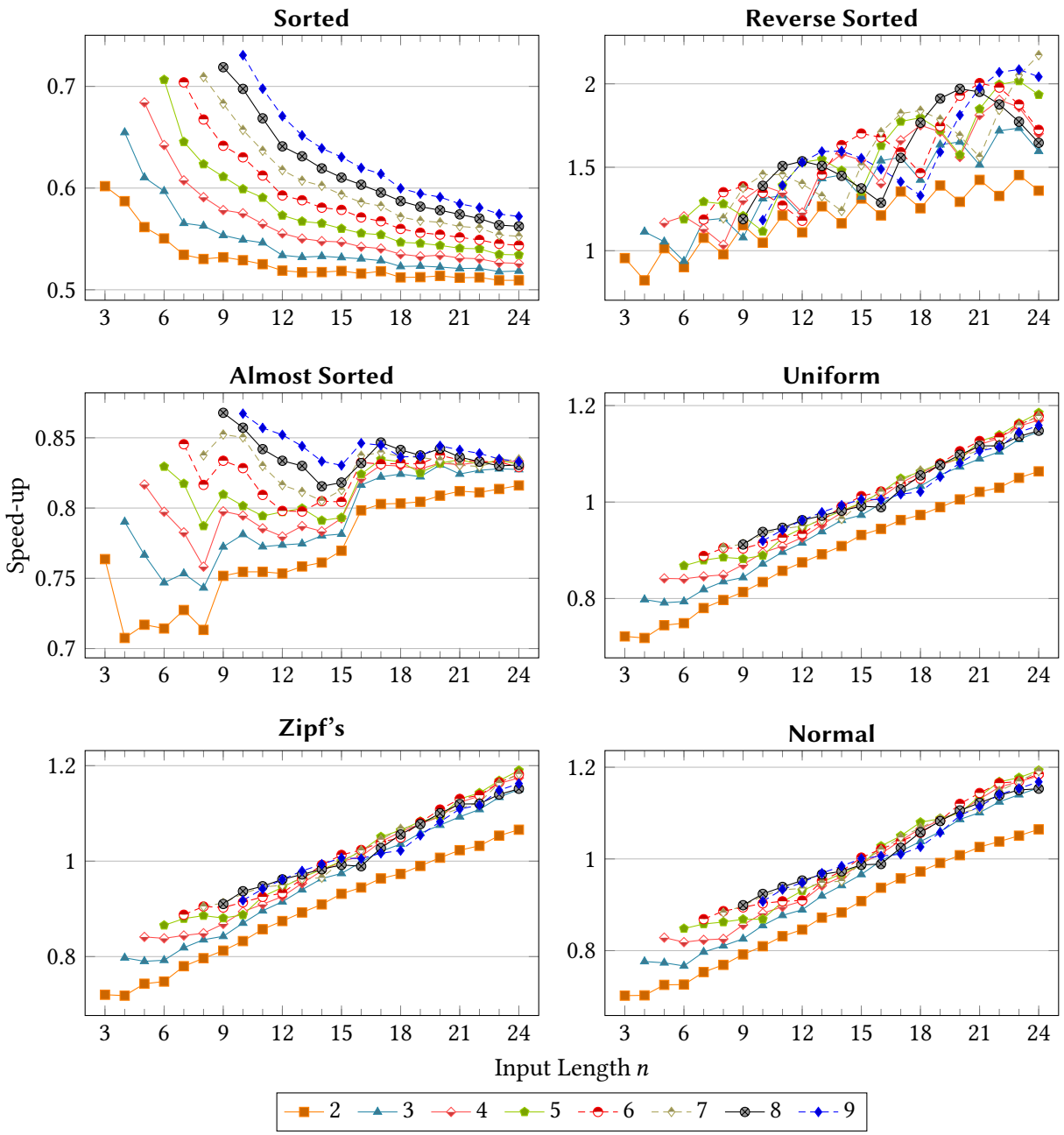


Figure 13: A continuation of Fig. 3 with more input distributions. Instead of total runtimes, the speed-ups with respect to InsertionSort are given for better clarity. The data type is 32-bit unsigned integers.

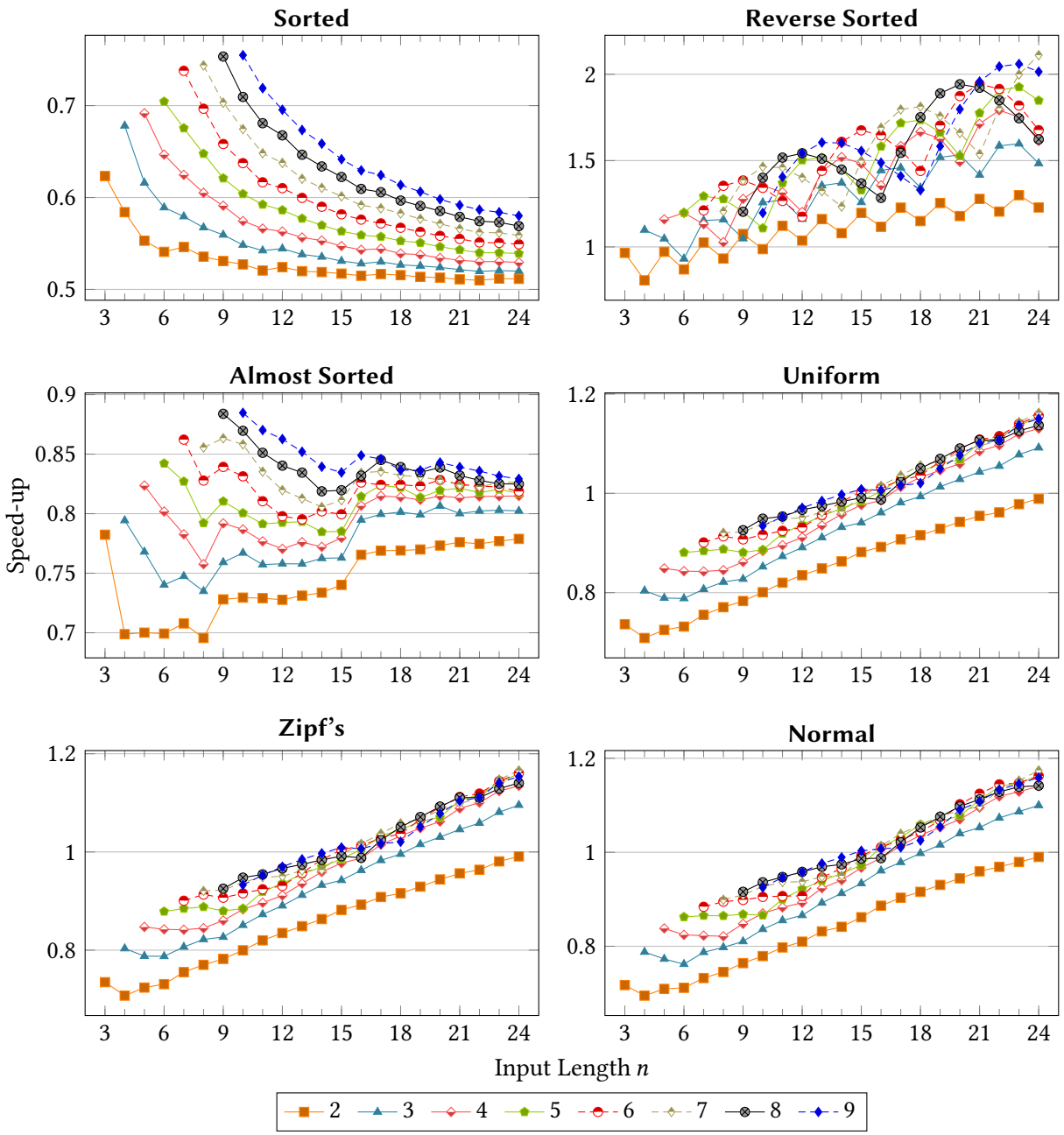


Figure 14: A continuation of Fig. 3 with more input distributions. Instead of total runtimes, the speed-ups with respect to InsertionSort are given for better clarity. The data type is 64-bit unsigned integers.

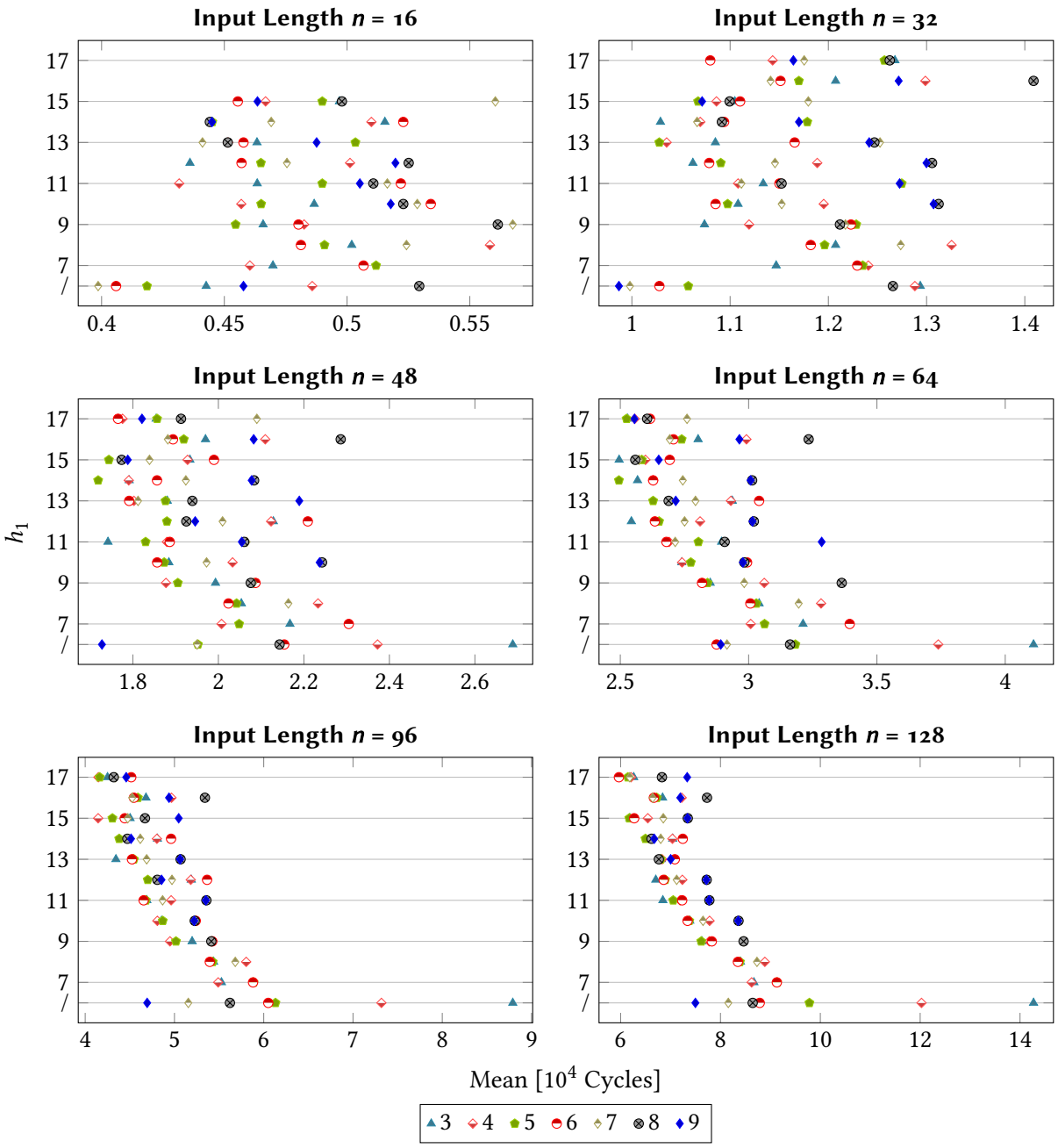


Figure 15: Runtimes of ShellSorts with two passes (/) and three passes (7–17). The coloured symbols encode the step size h_1 for two-tier ShellSorts and the step size h_2 for three-tier ShellSorts. For the latter, the step size h_1 is noted on the y-axes.
A variation of Fig. 4. The data type is 32-bit unsigned integers. The input distribution is the reverse sorted one.

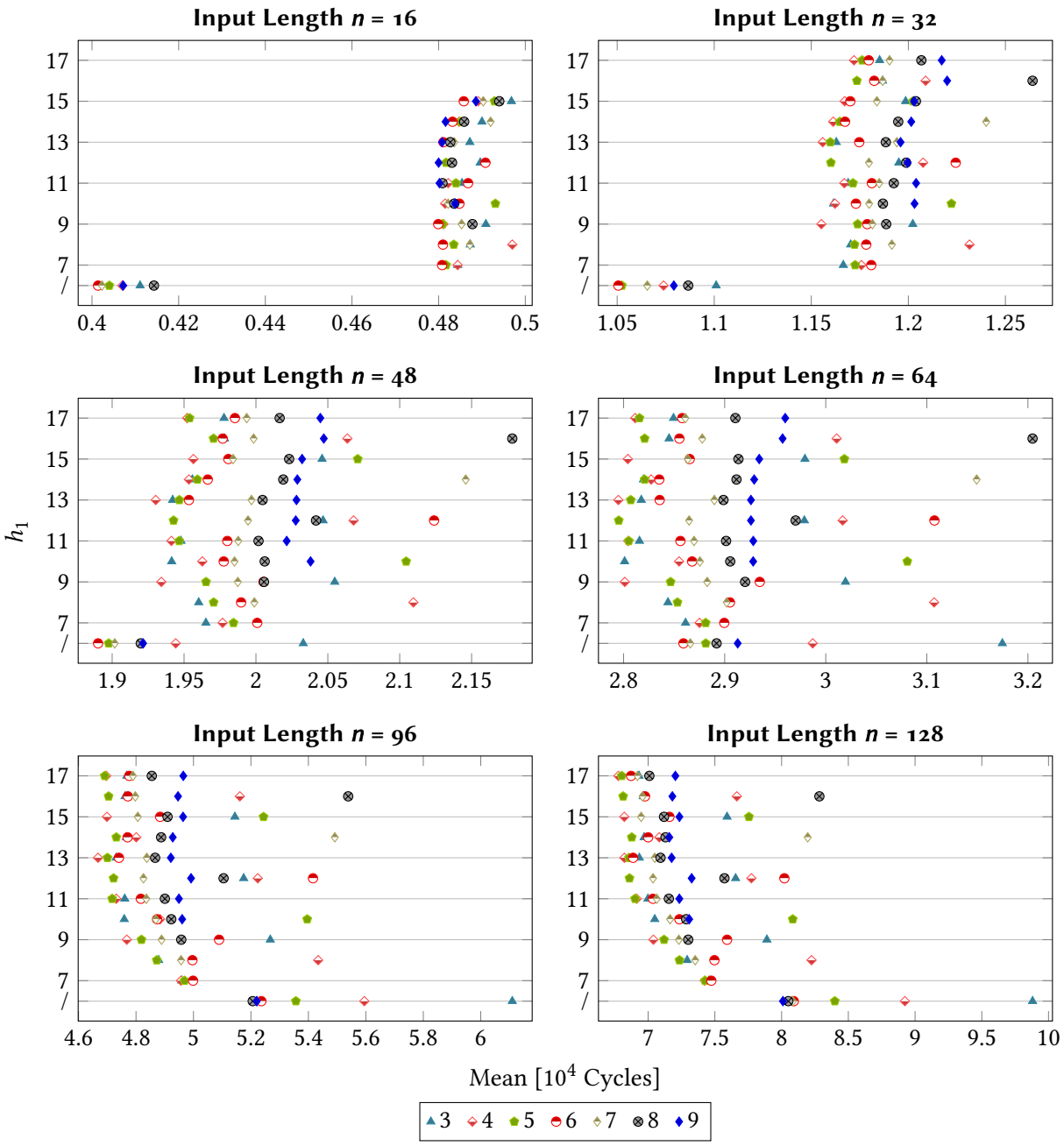


Figure 16: Runtimes of ShellSorts with two passes (/) and three passes (7–17). The coloured symbols encode the step size h_1 for two-tier ShellSorts and the step size h_2 for three-tier ShellSorts. For the latter, the step size h_1 is noted on the y-axes.
A repetition of Fig. 4. The data type is 32-bit unsigned integers. The input distribution is the uniform one.

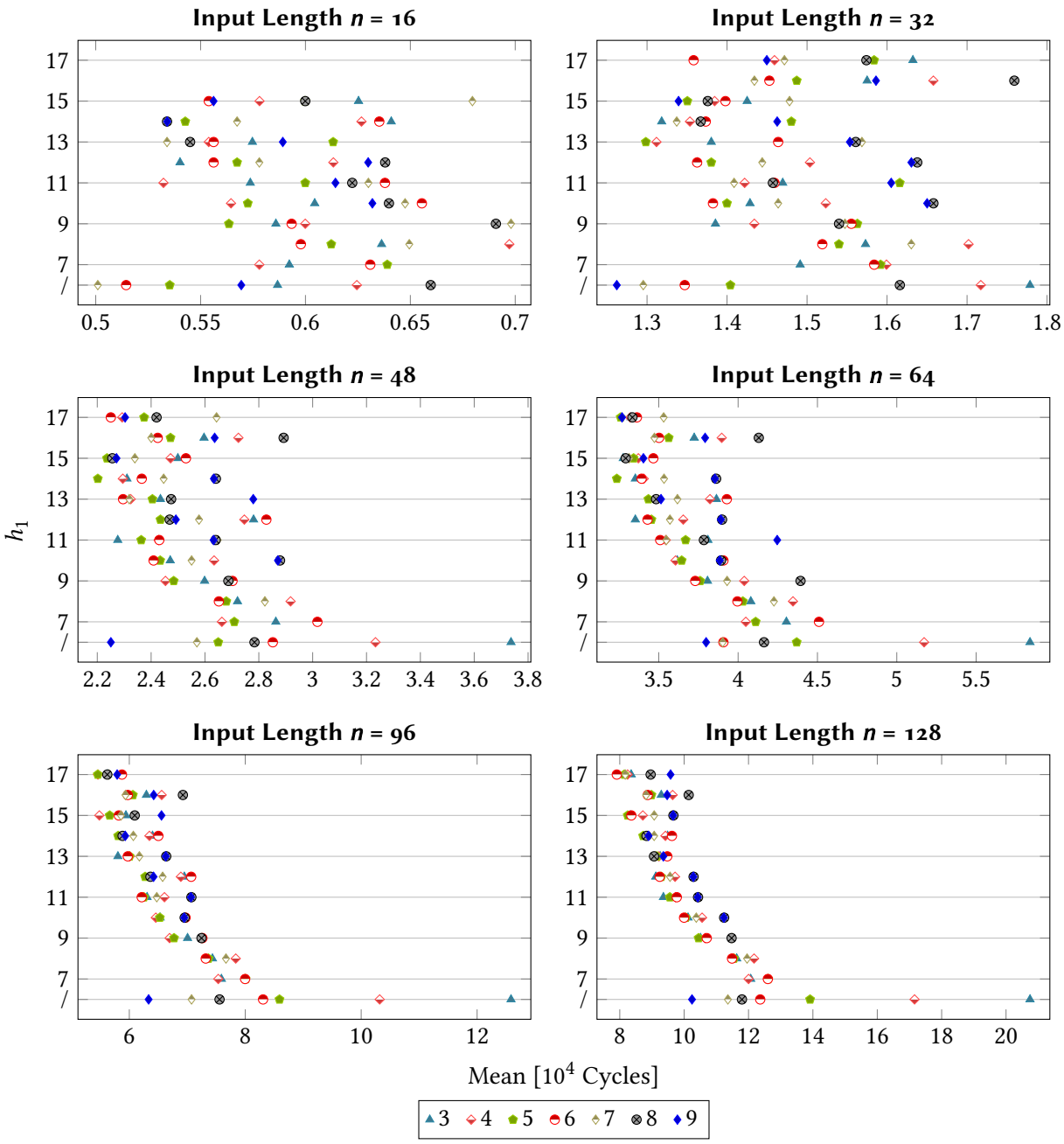


Figure 17: Runtimes of ShellSorts with two passes (/) and three passes (7–17). The coloured symbols encode the step size h_1 for two-tier ShellSorts and the step size h_2 for three-tier ShellSorts. For the latter, the step size h_1 is noted on the y-axes.
A variation of Fig. 4. The data type is 64-bit unsigned integers. The input distribution is the reverse sorted one.

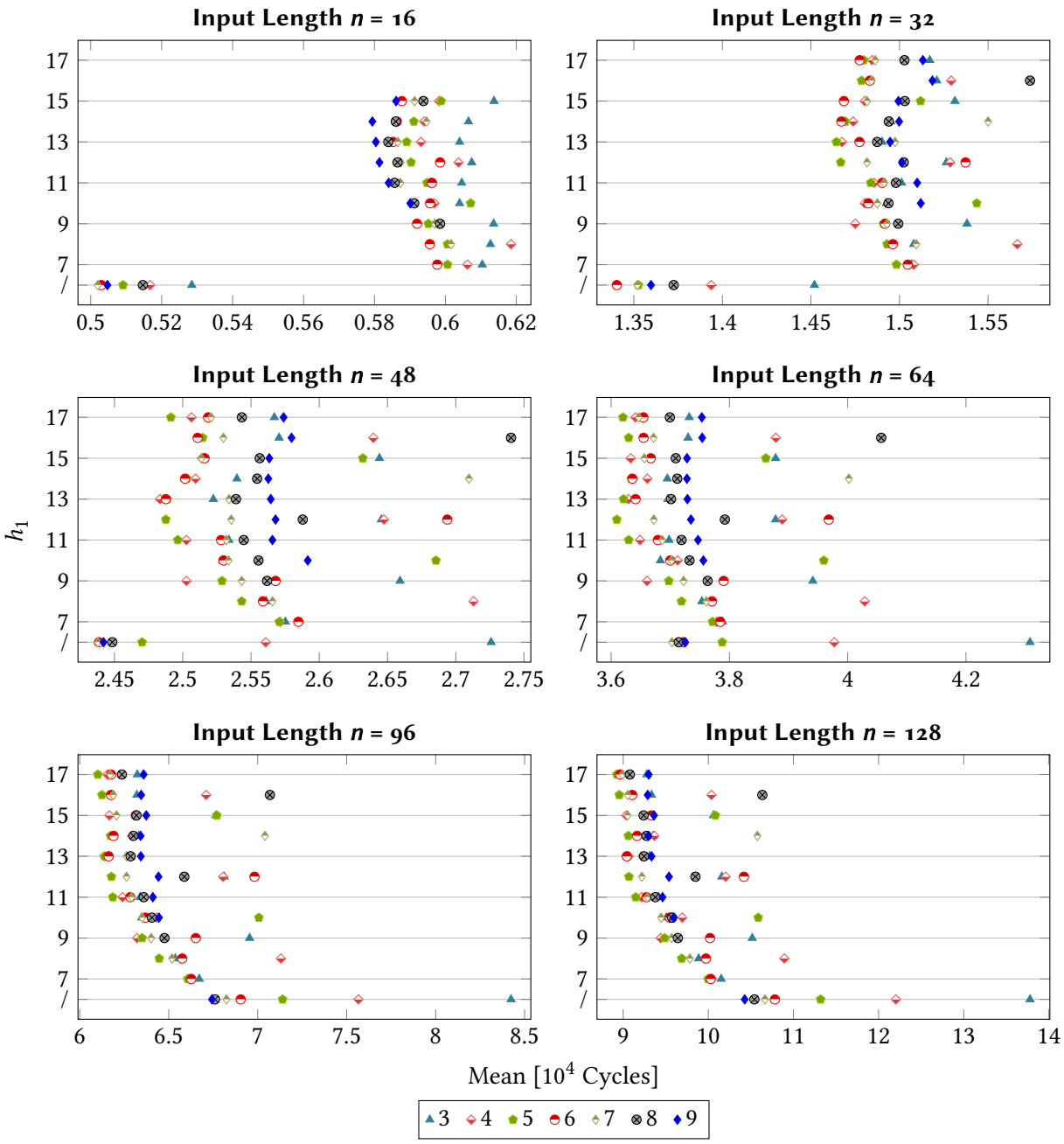


Figure 18: Runtimes of ShellSorts with two passes (/) and three passes (7–17). The coloured symbols encode the step size h_1 for two-tier ShellSorts and the step size h_2 for three-tier ShellSorts. For the latter, the step size h_1 is noted on the y-axes.
A variation of Fig. 4. The data type is 64-bit unsigned integers. The input distribution is the uniform one.

References

[1] Michael Axtmann et al. *Engineering In-place (Shared-memory) Sorting Algorithms*. 3rd Feb. 2021. arXiv: 2009.13569v2 [cs.DC].

[2] Marcin Ciura. 'Best Increments for the Average Case of Shellsort'. In: *Fundamentals of Computation Theory*. Ed. by Rūsiņš Freivalds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2nd Aug. 2001, pp. 106–117. ISBN: 978-3-540-44669-9. 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).

[3] 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).

[4] Ying Wai Lee. *Empirically Improved Tokuda Gap Sequence in Shellsort*. 21st Dec. 2021. arXiv: 2112.11112v1 [cs.DS].

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

[6] Oscar Skean, Richard Ehrenborg and Jerzy W. Jaromczyk. *Optimization Perspectives on Shellsort*. 1st Jan. 2023. arXiv: 2301.00316v1 [cs.DS].