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 repeated a thousand times on a uniform input distribution with 32-bit integers, 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

MergeSort half space; starting run length of 32 elements

HeapSort top-down for 32-bit integers; bottom-up with swap disparity for 64-bit integers

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 greater, assuming that the elements at the indices 0 to $i-1$ are already sorted. Its asymptotic runtime is bad, reaching $\mathrm{O}(n^2)$ 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 $\mathrm{O}(n)$. 2. It works in-place, needing only $\mathrm{O}(1)$ 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 short subarrays.

Sentinel Values 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 the smallest possible value of the chosen data type, it is at least as small as any value in the input array, and 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 such 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.

Evaluation of the Performance

The runtimes of the explicit and implicit InsertionSorts can be compared in the Figs. 1, 10 and 11. For most input distributions, the implicit InsertionSort is logically a bit slower, effectively adding

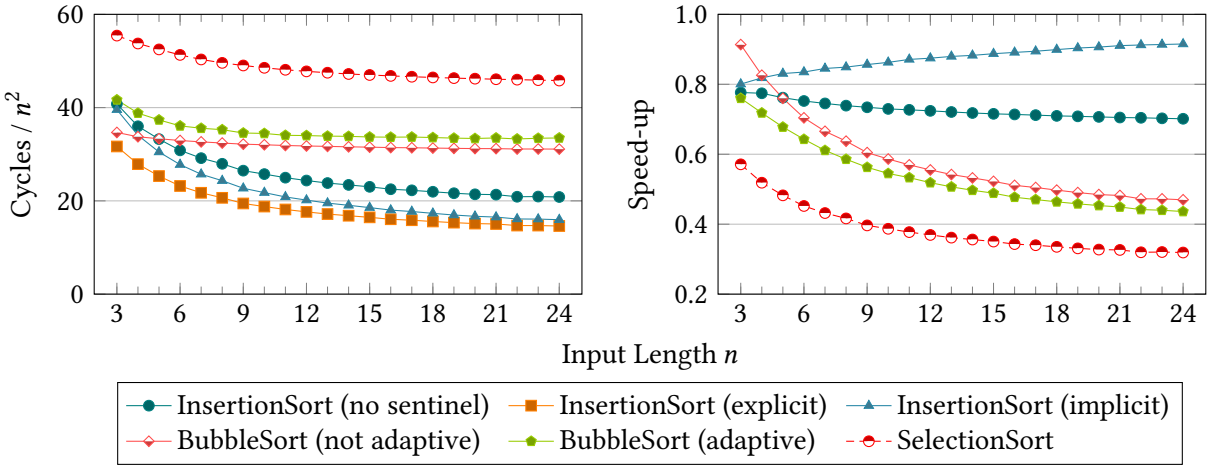


Figure 1: Comparison of sorting algorithms with a runtime in $O(n^2)$. The adaptive BubbleSort terminates prematurely if no changes were made to the input array during an iteration. The speed-ups are with respect to the InsertionSort relying on explicit sentinel values.

one check instruction to each round. An outlier, however, are the reverse sorted inputs. For 32-bit numbers (Fig. 10), the implicit version is up to 45% slower than the explicit one. This comes as 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 the implicit InsertionSort, although doing both value checks and address checks! For 64-bit numbers (Fig. 11), 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.

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 (see Figs. 1, 10 and 11).

Investigation of 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

```
1 void InsertionSort(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 InsertionSort(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 InsertionSort(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 InsertionSort(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.

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 `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, .LBB; 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.

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).

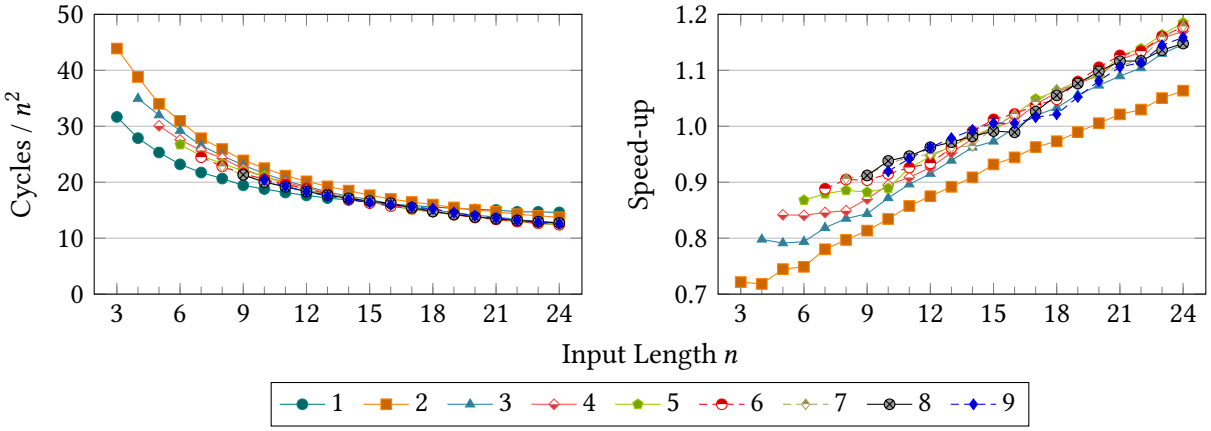


Figure 3: Comparison of InsertionSort (1) and various two-tier ShellSorts (2–9), whose step sizes h_1 are indicated by the labels. The speed-ups are with respect to the InsertionSort.

Evaluation of the Performance

Let us first focus on short 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, 12 and 13 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 short inputs, reaching between 50% and 110% at around 24 elements.

When moving to greater input lengths (Figs. 4 and 14 to 17), 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 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 longer inputs, as was the case for InsertionSort.

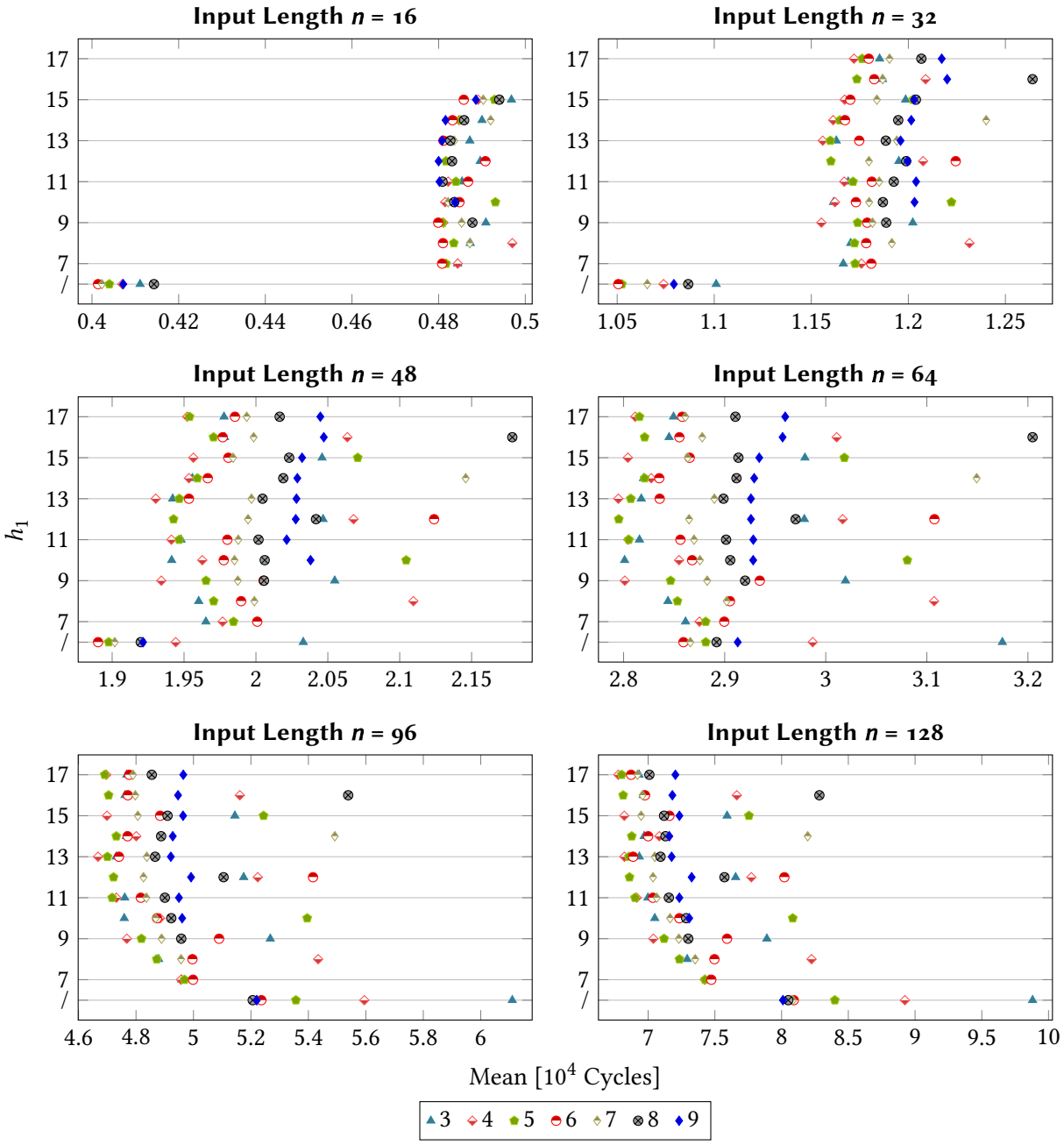


Figure 4: Runtimes of ShellSorts with two passes (/) and three passes (7–17). The coloured symbols encode the step size h_1 for two-tier ShellSorts and the step size h_2 for three-tier ShellSorts. For the latter, the step size h_1 is noted on the y-axes.

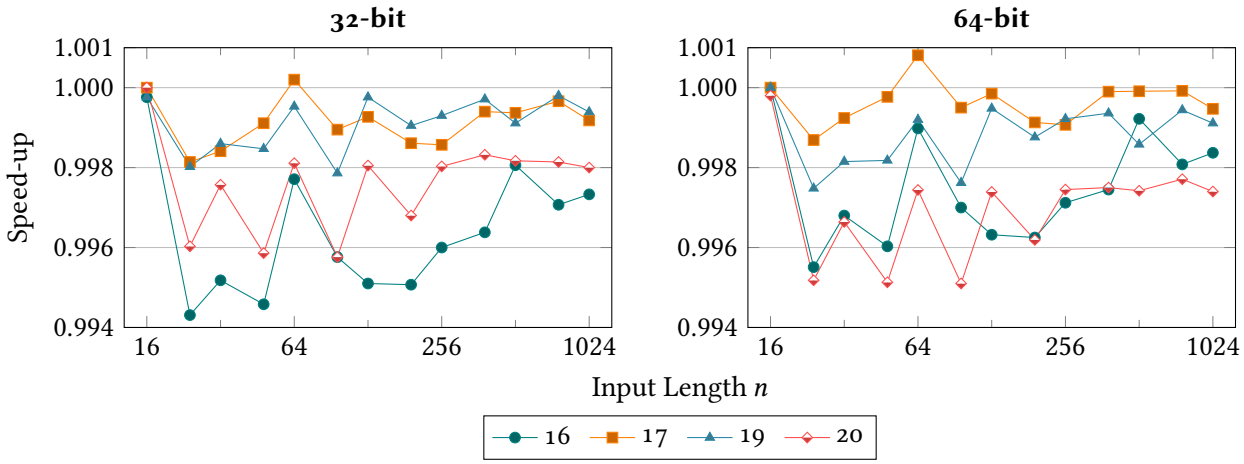


Figure 5: Speed-ups of QuickSorts with different thresholds for when to fall back to InsertionSort over a threshold of 18 elements. Using ShellSort was not beneficial overall, likely because many partitions fall below the thresholds.

'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 greater or lesser than the pivot are moved to the right or left of the pivot, respectively. Finally, QuickSort is used on the left and right partitions. The QuickSorts implementations presented here are neither stable nor in-place.

Base Cases When only a few elements remain in a partition, QuickSort's overhead predominates such that InsertionSort lends itself as fallback algorithm. As seen in Fig. 5, the optimal threshold for switching the sorting algorithm is 18 elements for uniform inputs and likely similar for inputs following Zipf's or normal distributions. For sorted and almost sorted inputs, the threshold is higher since QuickSort does not move elements which are already on the correct side of the pivot so (almost) no changes are made. Since InsertionSort does little more than one scan of these inputs, falling back earlier and, thus, ending the sorting process is better. For reverse sorted inputs, the threshold is also higher even though these are the worst-case inputs for InsertionSort. The reason lies within the implementation of QuickSort which repeatedly

swaps the foremost element greater than the pivot and the hindmost element lesser than the pivot. This way, reverse sorted inputs are reversed in the very first partition step. However, these input distributions should be catered for by a pattern-defeating QuickSort as laid out in , hence the 18 elements as default threshold.

Verweis auf später

Besides falling back to InsertionSort, another base case is imaginable, namely terminating when a partition has a length of at most 1 elements. Realistically speaking, checking for this should not be necessary, because even though the extra check is doable with just one additional instruction, it occurs rarely, 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, as seen later in 'Investigation of the Compilation'.

Recursion vs. Iteration In theory, the question of whether a DPU algorithm should be implemented recursively or iteratively comes down to convenience. Due to the uniform costs 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 due to less code complexity.

Kann jump r23 es kaputtmachen?

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%.

zurück-kehren

Partition Prioritisation Whether the left-hand or the right-hand is put on the stack should not make any difference for the runtime. However, always putting the longer partition on the stack guarantees that the problem size is at least halved each step, meaning the call stack stores $O(\log n)$ elements at most. This last approach, as shown later, is linked to huge speed penalties, so always prioritising the same partition is actually used throughout this Thesis in general. But even then, the choice between the two sides can have tremendous effects.

zurück-kehren

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 additional instructions.
- Taking the *median of three elements*, namely the first, middle, and last one, is far more computationally expensive since the position of the middle element must be calculated, the median be determined, and the pivot be swapped with the last element of the array, where it acts as sentinel. The plus side is that this method increases the chances of choosing a pivot that is neither particularly high nor particularly low. This leads to more balanced partitions such that the call stack is less likely to overflow and the base cases are reached faster.

- A *random element* is most efficiently drawn using an xorshift random number generator and rejection sampling [3]. This takes some instructions but impedes deterministically chosen worst-case inputs.
- Taking the *median of three random elements* is a combination of the previous two methods. For simplicity, there is no check on whether an element is drawn twice or thrice. Since the partitions are rather long, this should happen seldom, anyhow.

Luckily, the pivot choice seldom has bearing on the overall compilation, making a comparison easier. The results are shown in ??. 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.

Stimmt nicht!

zurück-kehren

Investigation of 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 short 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.

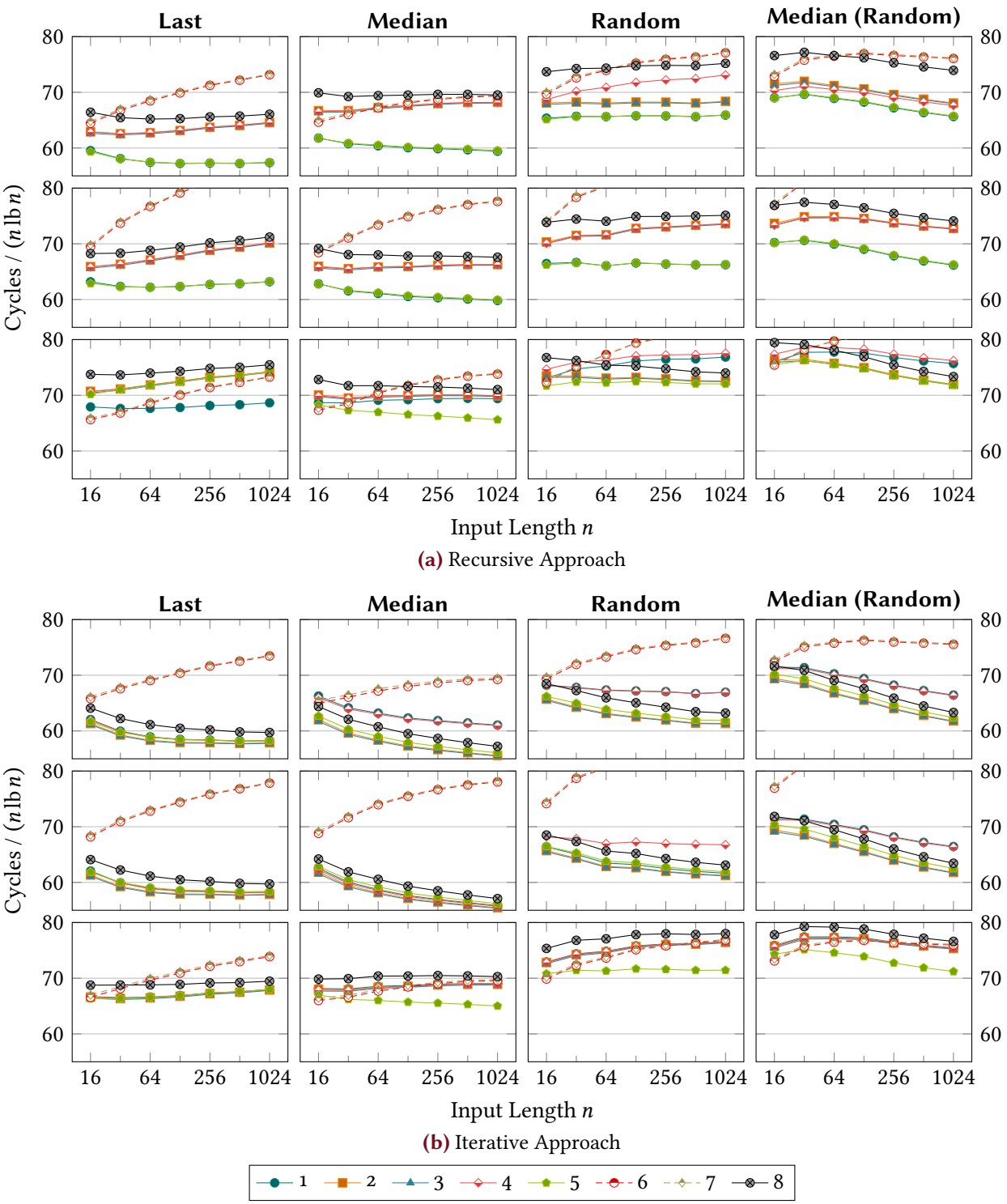
- This Implementation, as well as the next two, gets rid of some unneeded uses of QuickSort. In the recursive case, these Implementations lose the property of being tail-recursive.

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. 6. When using recursion, Impl. 1 and 5 perform the best, especially for longer 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.



(a) Recursive Approach

(b) Iterative Approach

Figure 6: Comparison of Impl. 1 to 8 and different pivots. Left-hand partitions are prioritised in the first rows, right-hand ones in the second rows, and shorter ones in the third rows.

1.4. MergeSort

MergeSort repeatedly compares two sorted subarrays and merges them into a longer sorted array in time $\Theta(n \log n)$. Unlike QuickSort, this runtime is guaranteed. Furthermore, the sorting is naturally stable but at the cost of not happening in-place.

Starting Runs Instead of starting by merging runs of length 1, it is beneficial to first create longer starting runs using ShellSort. Unlike QuickSort, where each partition naturally acted as sentinel for the subsequent one, 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 step sizes used for ShellSort — namely $h = (1)$ for lengths up to 16, $h = (6, 1)$ for lengths up to 48, and $h = (12, 5, 1)$ for everything above — have been chosen based on the results in Section 1.2, according to which these step sizes offer top performance for uniformly distributed inputs and medial performance for the reverse sorted inputs. Spot-check inspection suggest no deterioration of ShellSort's compilation due to inlining.

Memory Footprint A simple but fast implementation of MergeSort writes all merged runs to an auxiliary array, raising the need for space for n additional elements ('full space'). After a round is finished and all pairs of runs have been merged, the input array and the auxiliary array switch roles, and the merging starts anew. Are the final sorted elements supposed to be saved in the original input array, a final round with a write-back from the auxiliary array to the input array is needed for some input lengths.

A slightly more sophisticated implementation needs space for only $n/2$ additional 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 start 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. Further optimised, MergeSort would not need to copy the first runs immediately. It suffices to search for the foremost element of the first run which is greater than the first element of the second element. All previous elements are already in the correct position so only the following elements need to be copied to the auxiliary array. This optimisation, although examined during development, was not in use when measuring runtimes since it unfortunately complicates another optimisation, namely unrolling.

Unrolling There are four common reasons for *flushing*, that is, writing — many oftwhiles — consecutive elements:

1. When two runs are merged and the end of one of them is reached, the remaining elements of the other one can be moved safely to the output location. Especially with the sorted, reverse sorted, and almost sorted input distributions, the number of remaining elements will be high.
2. The number of runs is odd, so the full-space MergeSort moves the last run to the output location immediately.
3. The full-space MergeSort may write all elements from the auxiliary array back to the input array if the former contains the final sorted sequence.

4. The half-space MergeSort copies runs, whose length are always a multiple of the the starting run length, before each merger of pairs.

Therefore, flushing account for a considerable part of the runtime, and reducing the loop overhead (variable incrementation and bounds checking) is helpful. This can be done via *unrolling*: As long as at least, let us say, x elements still need to be flushed, the x foremost elements are moved first and then all necessary variables are incremented by x. Is x a compile-time constant, the compiler implements the moving of the elements through x instruction which use constant, pre-calculated offsets. Once less than x elements remain, an ordinary loop which moves elements individually is used. In good cases, this approach reduces the loop overhead to an xth, whilst in bad cases, where less than x are to be flushed, the overhead is increased by one additional check.

Due to time reasons, we refrained from doing automatic and extensive tests and relied on manual and exploratory tests to come up with the following strategy: When the full-space MergeSort performs a write-back or when the half-space MergeSort copies the first run, x is set to the starting run length. In all other cases, x is set to 24. This strategy, albeit not optimal, makes the MergeSorts significantly faster: Sorting sorted, reverse sorted, and almost sorted inputs sees speed-ups up to 30%, whereas sorting more random inputs still sees speed-ups for the most part and slow-downs into low single-digits at worst, depending on the starting run length.

Investigation of the Compilation

Yet again, the compiler shows unforeseen behaviour. The following is a collocation of some of the issues found while engineering MergeSort. They will not be discussed in detail here but still provide a point of reference for future work:

- As already mentioned, sorting the starting runs via ShellSort requires the placement and later removal of temporary sentinel values. For the very first starting run, one can omit storing and restoring the overwritten elements by using permanent sentinel values; this optimisation was in use when measuring runtimes. On the downside, this leads to a bigger compilation as ShellSort is inlined twice. If the size of the whole compilation is already close to the maximum, one might be inclined to handle the first starting run just like the others. Realistically, this should slow down the total runtime by just a few hundreds of cycles, yet the real slow-down is in the thousands.
- If the input is so short that it fits entirely within the first starting run, one can immediately end the execution after ShellSort is done. Several implementations were tested, with unsatisfactory results: Some increased the runtime for longer inputs, others decreased it but also increased it for shorter inputs. The settled-on implementation is of the former variety since the increases hit shorter inputs harder relatively and a more thorough solution would not further the purpose of this section.
- Concerning the half-space MergeSort: Treating the copied run logically as the second run and the uncopied run as the first one nets a noticeable decrease in runtime compared to an implementation with flipped logic. Even worse, only with the former does unrolling improve the speed, being an impairment with the latter! This behaviour occurs with both immediate and deferred copying of the first runs. An inspection of the issue unearthed marvels like code

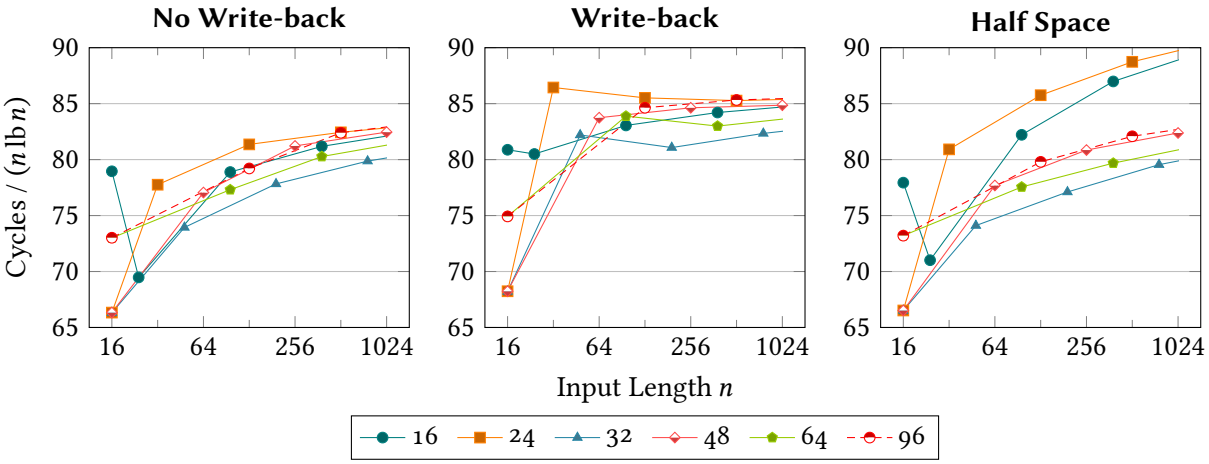


Figure 7: Comparison of MergeSorts, which need an auxiliary array of length either n ('No Write-back' / 'Write-back') or $n/2$ ('Half Space'), for different lengths of the starting runs. The MergeSorts use a ShellSort with the step sizes $h = (1)$ for length 16, $h = (6, 1)$ for lengths 24 to 48, and $h = (12, 5, 1)$ for lengths 64 and 96, respectively.

of the form

```
*i++ = *j; a = b - i; c = i; i = d;
```

leading to 5% longer runtimes compared to

```
*i = *j; a = b - (i + 1); c = i + 1; i = d;
```

even though executed at most once per merger, but we could sadly not pinpoint the fundamental cause for the behaviour.

Evaluation of the Performance

Three implementations have been tested: full space MergeSort without write-backs, full space MergeSort with write-backs, and half space MergeSort. Figures 7 and 18 to 21 show their performance for various starting run lengths. Please note that the plots are smoothed: Whenever the number of rounds increments, the runtimes hike, making the zigzagging plots cross each other unswervingly and, thereby, hard to read. Thence, the figures contain marks for select measurements only in such a way that the resulting plots act as an upper bound on the runtime.

The measurements show that the MergeSorts guarantee a runtime of $O(n \operatorname{lb} n)$ as expected. The differences in runtime between the different input distributions are small compared to QuickSort and are ascribable to ShellSort and to the differing suitability of the unrolling; cases where the usage of ShellSort worsened the runtime are unbeknown.

Even though the tested starting run lengths range from 16 to 96 elements, the mean runtime differences are surprisingly small. Notwithstanding that the optimal choice depends on the specific input length because of the zigzagging, a starting run length of 32 elements fares decidedly well on average across all tested scenarios.

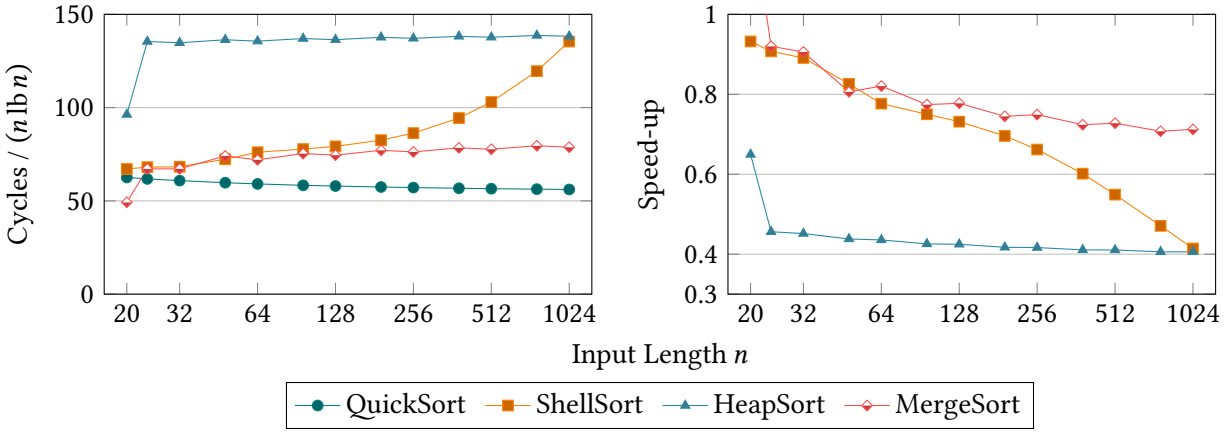


Figure 8: Comparison of MergeSort, HeapSort, ShellSort, and QuickSort. Due to MergeSort's increased space requirements, its runtime was measured only for up to 768 elements. The ShellSort uses the step sizes from **??**, which are unoptimised for long inputs. The speed-ups are with respect to the QuickSort.

The half-space MergeSort delivers a strong performance despite its vastly lower memory footprint. With 32-bit integers, it beats the full-space MergeSort without write-backs by 11% on sorted inputs and effectively ties on all other inputs but the reverse sorted ones where it narrowly falls behind. Naturally, the full-space MergeSort with write-backs is consistently (with the exception of reverse sorted inputs) at a disadvantage, despite seeing some light with inferior starting run lengths. With 64-bit integers, the full-space MergeSort without write-backs manages to turn the ties into scant leads in the range from 1% to 3%. Using the MergeSort with write-backs is still unprofitable.

In summary, a proper implementation of half-space MergeSort with deferred copying and fine-tuned unrolling would require some work but has the potential to be the overall best stable sorting algorithm.

1.5. HeapSort

Another sorting algorithm with a guaranteed runtime of $\Theta(n \log n)$ is HeapSort, which is unstable but in-place. A max-heap is a binary tree of logarithmic depth whose layers are fully filled, possibly with the exception of the last layer, which must be filled from left to right. Each vertex contains a key, and the key of each father must be at least as great as those of his sons. As a consequence of this heap order, the root contains the greatest key.

A heap with n keys can be represented as an array of length n using a bijective mapping between the vertex positions and the array indices (see later). After the heap has been built in-place from the input array in time $O(n)$, the sorting works as follows: At the start of round $r = 1, \dots, n$, the first $n - (r - 1)$ elements of the array represent the heap and the last $r - 1$ elements the end of the sorted output. Upon removal of the root, which contains the rth greatest element of the input, the heap structure must be restored in time $O(\log n)$. Since the heap has shrunken by one key, the key of the removed root can be stored at the freed-up position directly

after the end of the heap.

Sifting Direction After the heap is built, the *top-down* HeapSort proceeds as follows: At the start of each round, the root and the rightmost leaf ('last leaf') in the bottom layer swap places. The root is now in the right position, but the formerly last leaf may violate the heap order, that is, the root may have a lesser key than one or both of its sons. The greater of the two sons is determined, and the root and the greater son swap places. This downwards-sifting of the former leaf continues iteratively until the heap order is restored.

In contrast, the *bottom-up* HeapSort [7] works as follows: At the start of each round, the key of the root is removed so that a hole is now at the top of the heap. Then, the greater of the two sons of the hole is determined, and they swap places. This downwards-sifting of the hole continues iteratively until it becomes a leaf. Now, the last leaf is moved to the position of the hole, which could violate the heap order if the moved leaf is greater than its father. If so, it needs to be sifted upwards by iteratively swapping positions with its respective father until the heap order is restored. At last, the original root key can be put where the formerly last leaf used to be.

The motivation behind these variants is at follows: In each step where the top-down HeapSort sifts the formerly last leaf downwards, two value checks (Which son is greater? Is the father lesser than the greater son?) need to be done. The leaves of a heap tend to be small so the downwards-sifting lasts awhile. As opposed to this, each step of the bottom-up HeapSort needs only one value check (Is the father lesser than the greater son?). Both HeapSorts sift downwards similarly long so many checks can be saved. Since the last leaf effectively takes the place of another leaf and since both are likely small, the upwards-sifting should be short-lived and, hopefully, not eat the gain up.

The upwards-sifting reverts some of the changes done by the downwards-sifting. The bottom-up HeapSort can be brought to swap parity with the top-down HeapSort with the following change: The downwards-sifting is traced but the keys are not actually moved. Once the leaf where the hole would end up is reached, the sifting is backtracked until the bottommost key which is at least as great as the last leaf. This is the position where the last leaf would end up after the upwards-sifting, so all keys below can stay put and all keys above move to their fathers' positions, that is, thither the swaps from the downwards-sifting would have put them. This makes the downwards-sifting even cheaper but the upwards-sifting must now go all the way up to the root.

Indexing With a zero-based indexing, the sons of a vertex i can be calculated with the well-known formulas $2i + 1$ and $2n + 2$. With a one-based indexing, the formulas turn into $2i$ and $2i + 1$. The compiler automatically turns the multiplication by two into a left-shift by one. Since DPUs can execute an instruction called `lsl_add` which first shifts leftwards and then adds an offset (useful e. g. for array indexing), the formulas $2i + 1$ and $2i$ take the same amount of time to compute.

Nevertheless, the zero-based indexing is about 7% slower despite `lsl_add` being indeed in use. The reason is that only the number of bits to shift can be passed as immediate value, that is as plain number, but not the offset, which must be passed via a register. While DPUs

have a read-only register storing the number 1 at disposal, read-only registers can only ever be the first register argument, not the second one, which, for `lsl_add`, would be the offset. As a consequence, the compiler moves the number 1 to a register whenever $2i + 1$ is to be computed, only to immediately overwrite the 1 with the result from `lsl_add`. Hence, the calculation of $2i + 1$ does take one more instruction than $2n$ after all.

Sentinel Values When HeapSort sifts a vertex downwards, it needs to determine the greater of its two sons before deciding whether and whither to move. If and only if the heap has an even number of vertices, there is a left son without a right brother: the rightmost leaf in the bottom layer. Instead of adding some check on whether the right brother exists, one can rather add the missing leaf and give it the smallest possible key each time the heap reaches an even size. Thus, if it has been confirmed that a left son exists, a right one does also exist, and if two brothers contain the same key, the left one should be considered greater.

Likewise, whenever HeapSort sifts upwards and considers the father $i/2$ of a vertex i, it will only proceed if the father is lesser. Since the fatherless root has index 1 and the result of an integer division is truncated towards 0 in C, the formula yields 0, so it makes sense to set the element at index 0 to the greatest possible key to stop any upwards-sifting. The savings from these approaches were around the 13% mark.

Code Duplication A strategy particularly useful for HeapSort, although also employed in MergeSort, is code duplication. Handling the greater of two sons is the fastest if the logic is written twice, once for either son, and then executed conditionally; logic written once for a generalised variable holding the greater son is compiled considerably worse. The savings from this approach were around the 7% mark.

Evaluation of the Performance

The measurements are visualised in Figs. 9, 22 and 23. In general, the performance of HeapSort is even less volatile than that of MergeSort with respect to the input distribution. Reverse sorted inputs are sorted the fastest since they are already max-heaps so the heap-building phase is short, while sorted inputs are sorted the slowest since they are min-heaps so the heap-building phase is long. Nonetheless, the reverse sorted inputs get sorted just about 10% faster and less on average. This low volatility cannot hide the fact that the total runtime is abysmal across the board: Compared to MergeSort, the runtimes are between 50% to 100% higher, depending on the input distribution!

The normalised runtimes of the top-down HeapSort and the bottom-up HeapSort with swap parity show a slight upwards trends, whereas those of the bottom-up HeapSort with swap disparity mostly shows a slight downwards trends with the exception of reverse sorted inputs, where it is also an upwards trend, albeit even slighter. Interesting are their rankings relative to each other: Value checks on 64-bit integers take two instructions, so that the savings of the bottom-up HeapSort with swap disparity allow it to outperform the top-down HeapSort even for short inputs, and the advantage grows with the input length. This makes sense as roughly 50% of the vertices are leaves and 25% are fathers of leaves, no matter the total size. Therefore, the percentage of formerly last leaves sifting down to the bottom remains steady

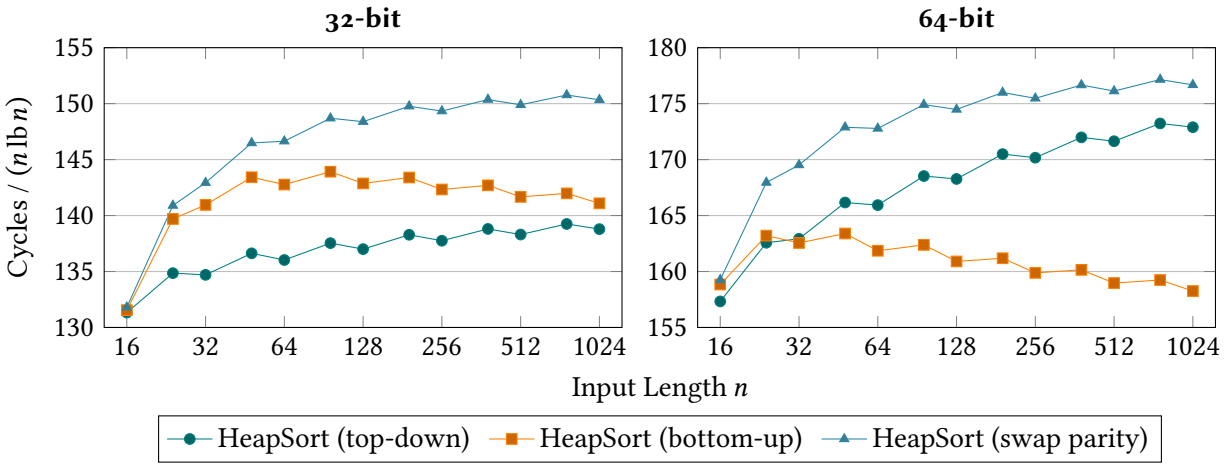


Figure 9: Comparison of the runtimes of three different HeapSort implementations on uniformly distributed 32-bit integers and 64-bit integers, respectively.

but the travelled distance increases. Value checks on 32-bit integers, on the other hand, take only one instruction, so that their reduction is overshadowed by the increased overhead from the longer downwards-sifting and the added upwards-sifting, whence the lead of the top-down HeapSort. Indeed, at around 2000 elements, the bottom-up HeapSort with swap disparity overtakes the top-down HeapSort because of their inverse trends, but the lead is narrow even at 6000 elements. However, these long inputs are less interesting anyway due to the limited WRAM and the multiple tasklets used in most applications.

The bottom-up HeapSort with swap parity consistently trails behind. This comes as no surprise since the overhead of its considerably prolonged upwards-sifting bears no proportion to the few swaps saved. This holds true even for 64-bit integers as moves still cost only one instruction. Unrolling the upwards-sifting proved to be unhelpful.

Investigation of the Compilation

The reason for the astoundingly poor performance is unclear. Building the heap makes up about 10% of the total runtime, so it can be excluded as reason. One major difference to the other sorting algorithms is that array indices instead of pointers are used. Nonetheless, this should not make a difference since the manipulation of indices and pointers take equally long, not least because of the `lsl_add` instruction, and since the same instructions are used to load and store data. Even so, this could factor in a suspected deterioration of the compiler optimisations. Unfortunately, time restrains bar as us from a profound investigation.

While engineering, some strange observations were made. For example, the runtime difference between stopping HeapSort when only one element remains in the heap and stopping HeapSort when only three elements remain (which then get sorted by InsertionSort) can be in the tens of thousands of cycles in both directions. Stopping HeapSort even earlier has comparatively little effect. The measurements were conducted without employing InsertionSort.

The undisputedly strangest observation was the following: Before building a heap, a single

sentinel leaf must be inserted if the input length is even. Adding this leaf if the input length is odd makes no difference algorithmically, being a left leaf never to be accessed due to bounds checks. However, adding an if-statement within which the sentinel leaf is placed has dramatic effects compared to placing the sentinel value unconditionally: Since the parity of the input length is needed later anyway, the version with the if-statement is expected to gain one instruction. Yet, when measuring the runtimes on 1024 elements, for example, one can observe anything between a reduction by 5000 cycles over changes within the margin of error to increases by 25 000 cycles, depending on the implementation and the input distribution. A comparison of the compilations reveals minute differences at the beginnings of functions, none of which affect loops. Adding one or more sentinel leafs outside of the functions has no impact on this behaviour.

A. Further Measurements on Sorting with One Tasklet

A.1. InsertionSort

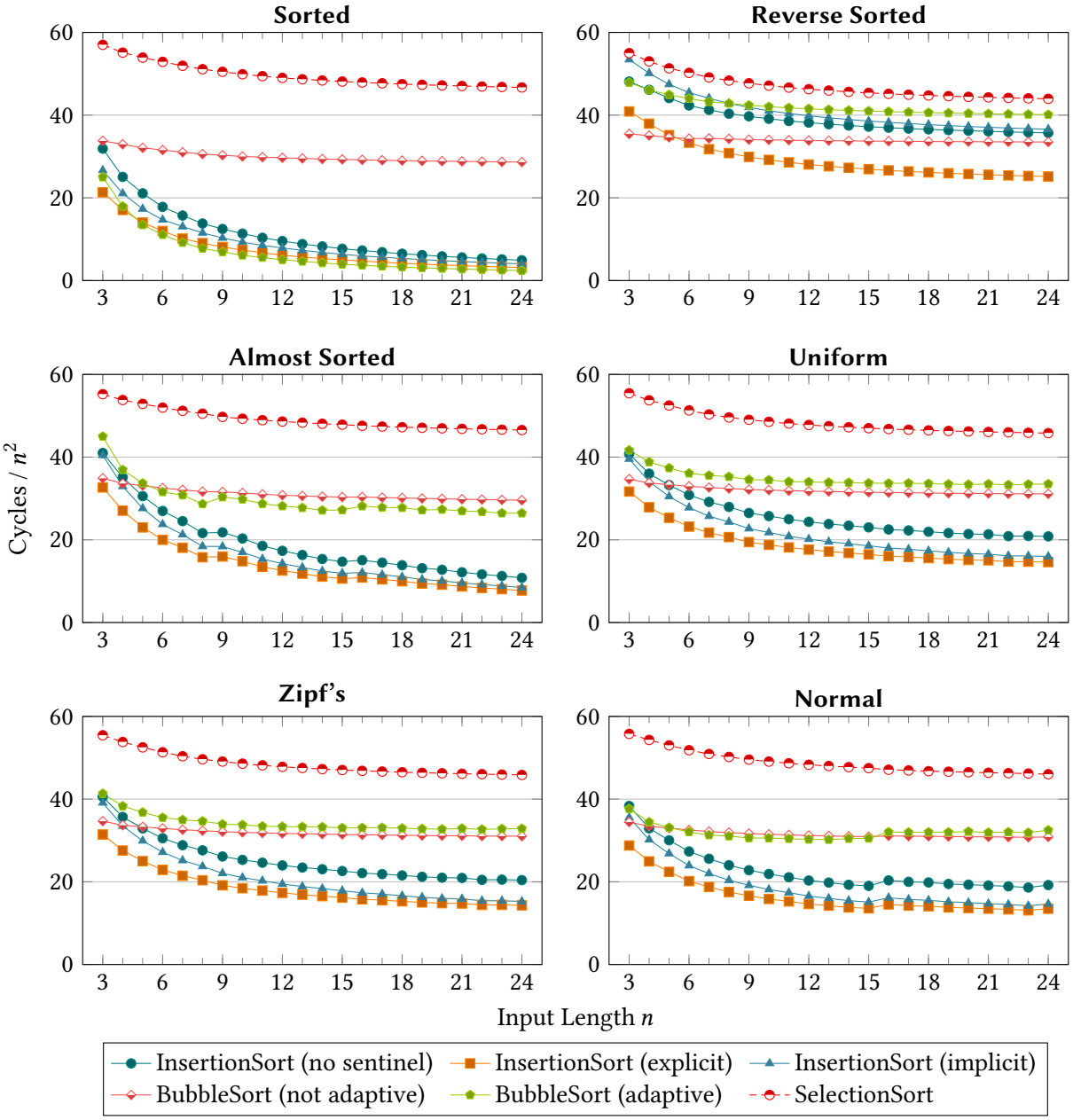


Figure 10: An extension to Fig. 1. The data type is 32-bit unsigned integers.

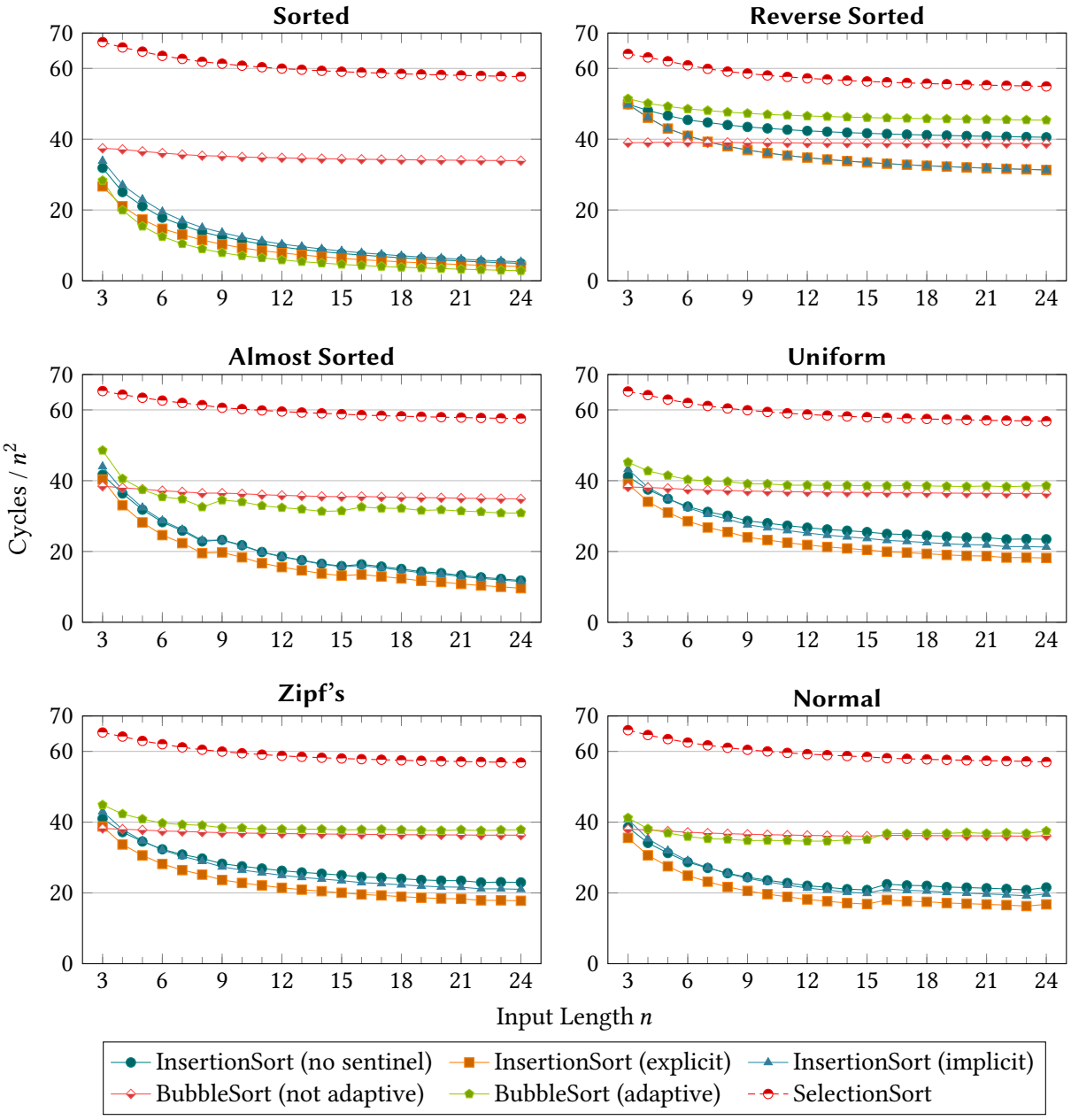


Figure 11: An extension to Fig. 1. The data type is 64-bit unsigned integers.

A.2. ShellSort

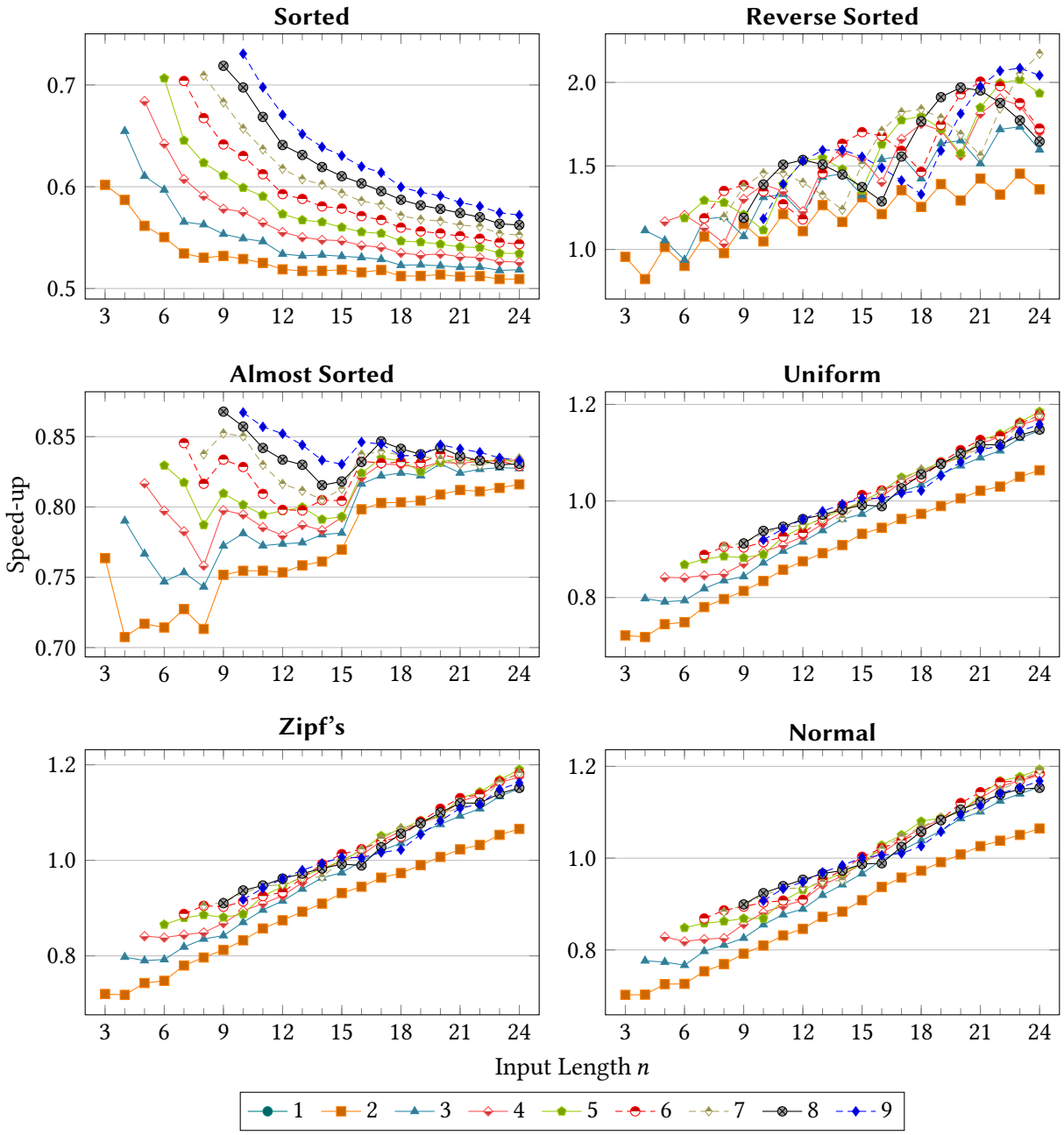


Figure 12: An extension to Fig. 3. Instead of total runtimes, the speed-ups with respect to InsertionSort are given for better clarity. The data type is 32-bit unsigned integers.

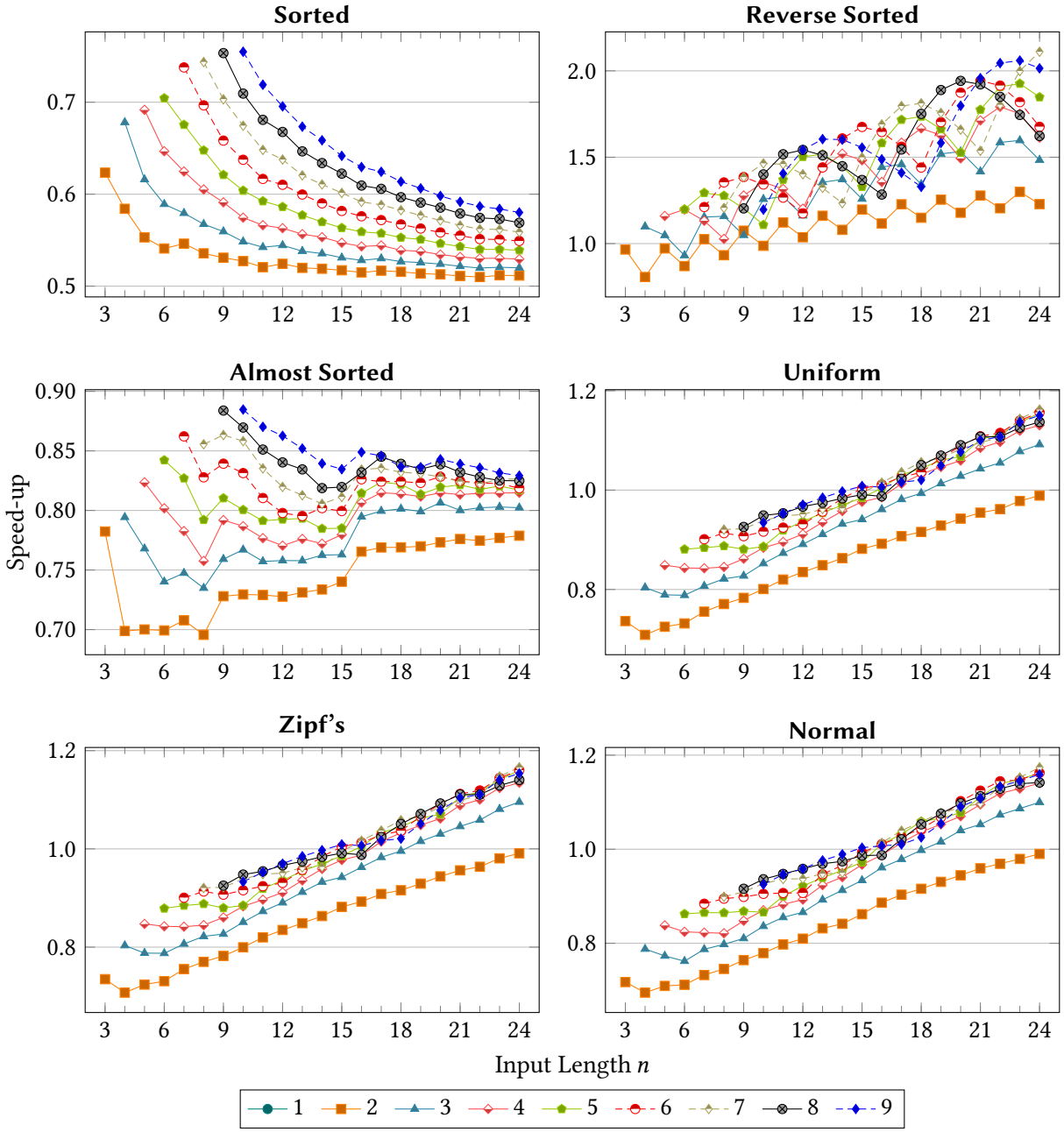


Figure 13: An extension to Fig. 3. Instead of total runtimes, the speed-ups with respect to InsertionSort are given for better clarity. The data type is 64-bit unsigned integers.

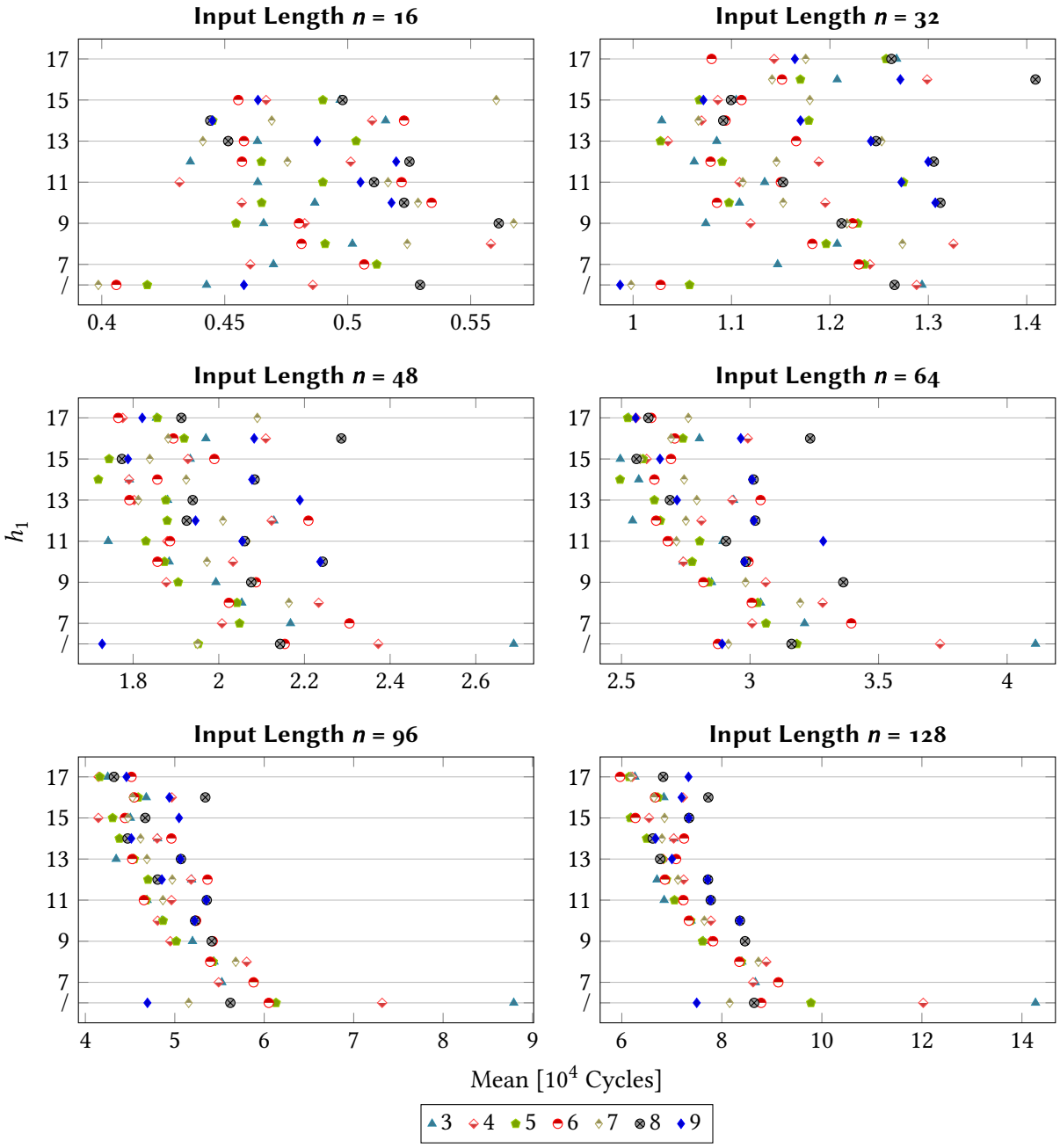


Figure 14: An extension to Fig. 4. The data type is 32-bit unsigned integers. The input distribution is the reverse sorted one.

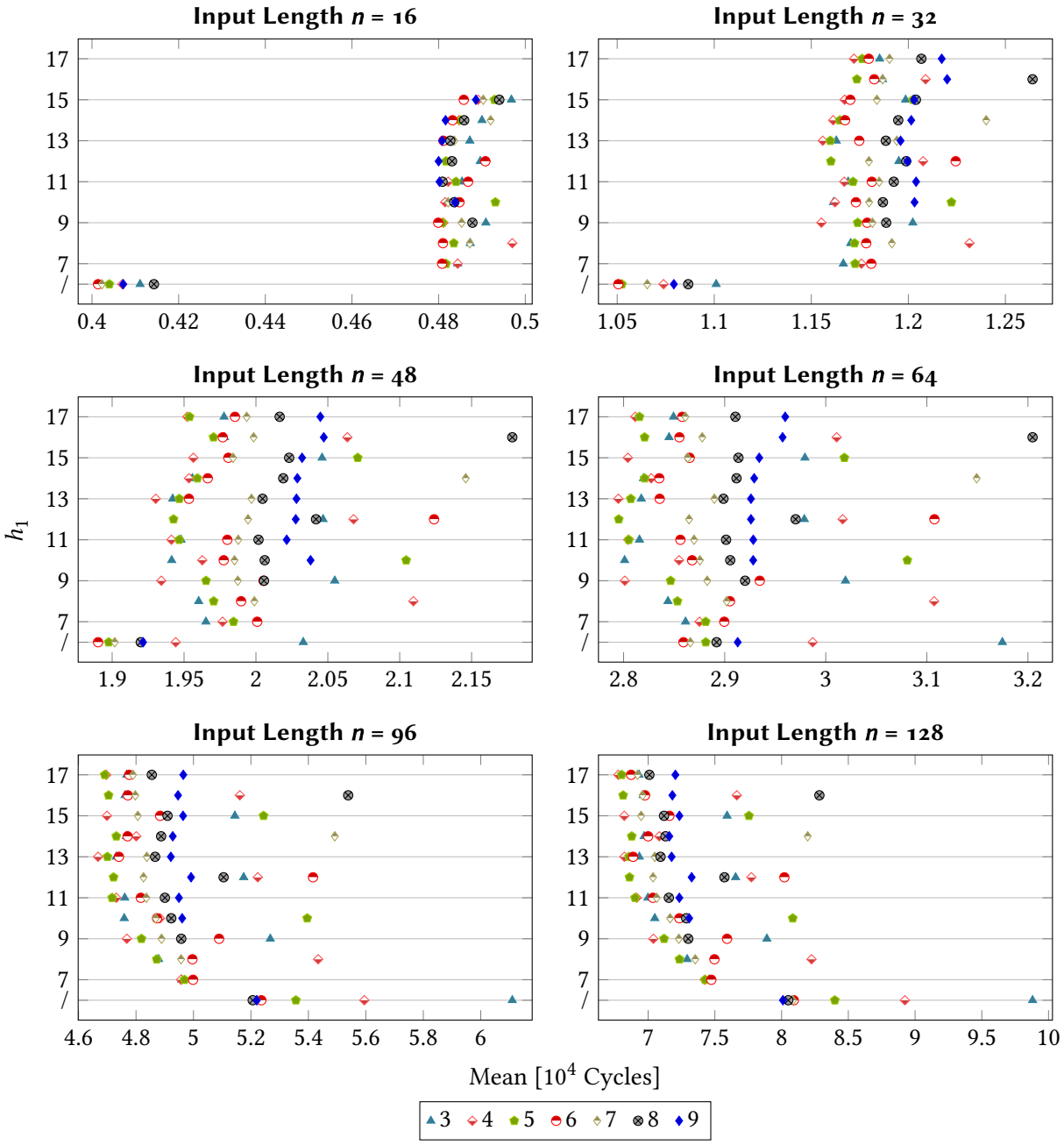


Figure 15: A repetition of Fig. 4. The data type is 32-bit unsigned integers. The input distribution is the uniform one.

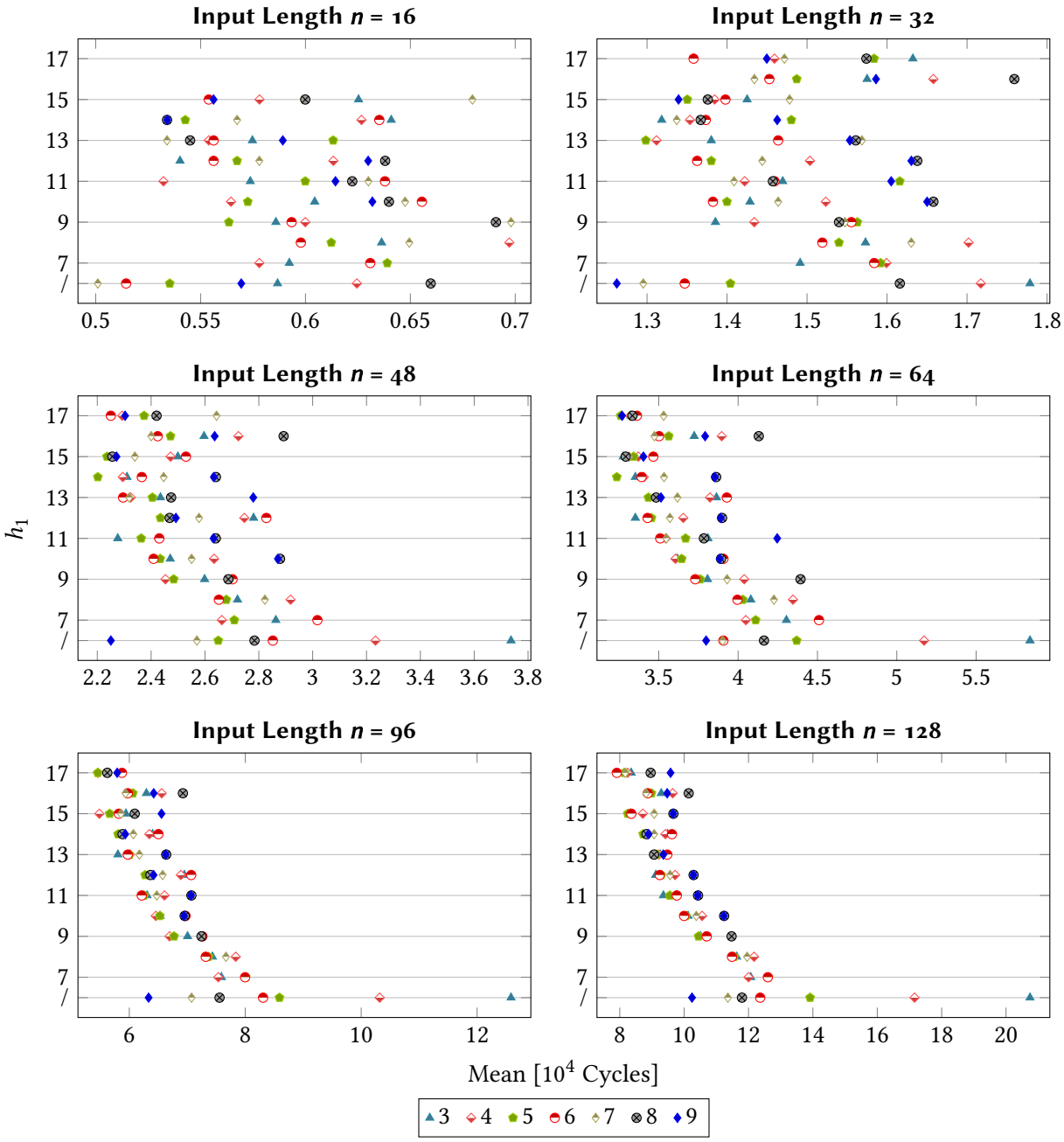


Figure 16: An extension to Fig. 4. The data type is 64-bit unsigned integers. The input distribution is the reverse sorted one.

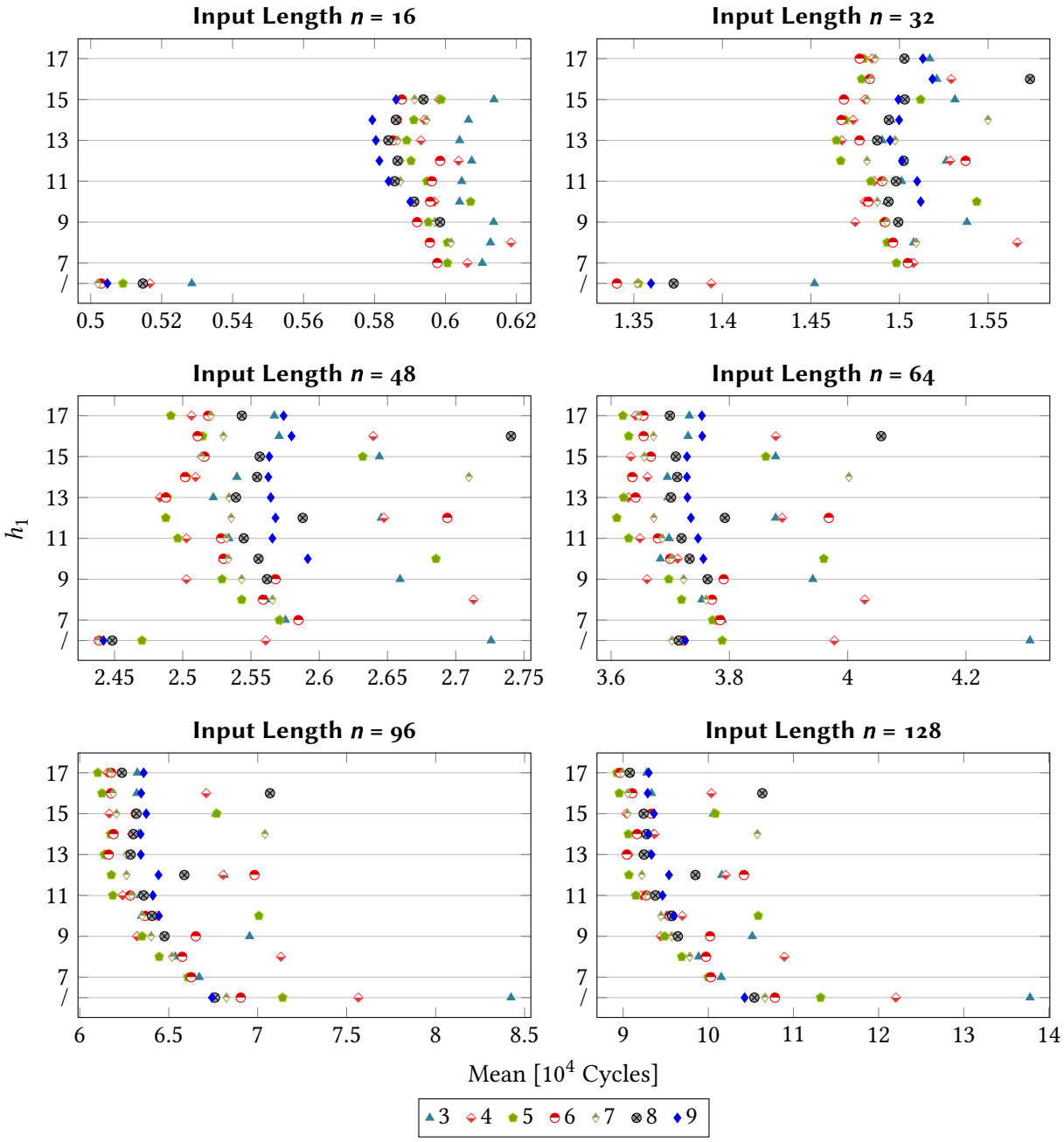
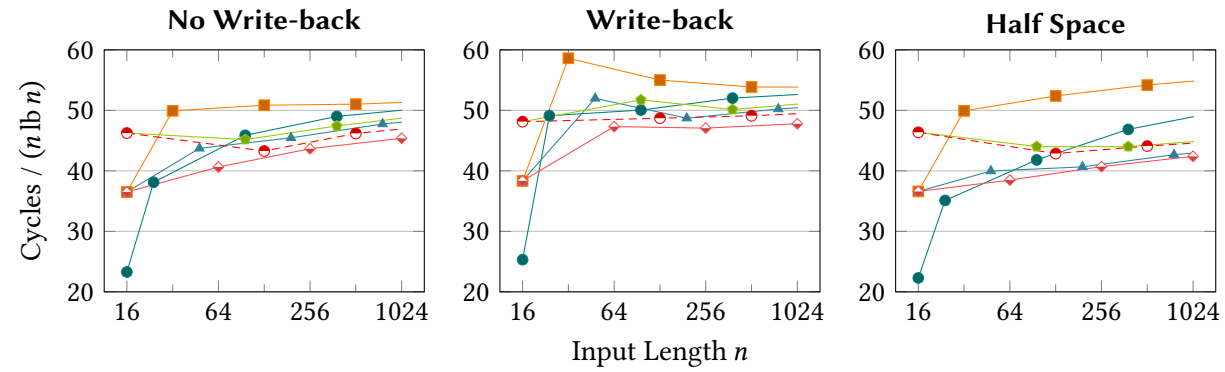
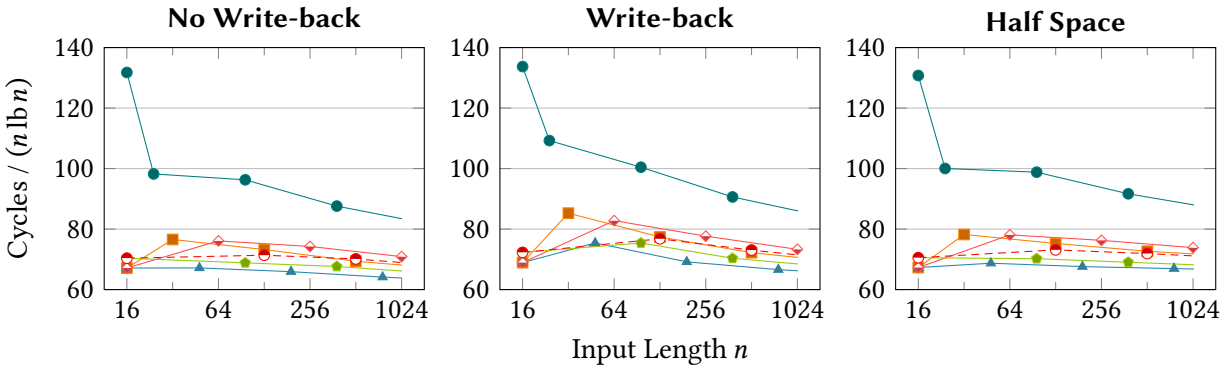


Figure 17: An extension to Fig. 4. The data type is 64-bit unsigned integers. The input distribution is the uniform one.

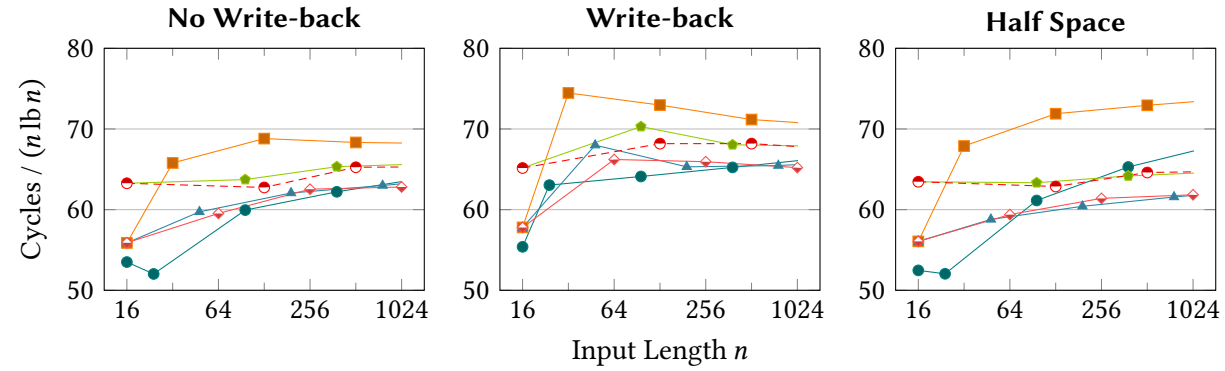
A.3. MergeSort



(a) Sorted



(b) Reverse Sorted



(c) Almost Sorted

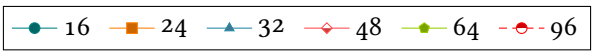
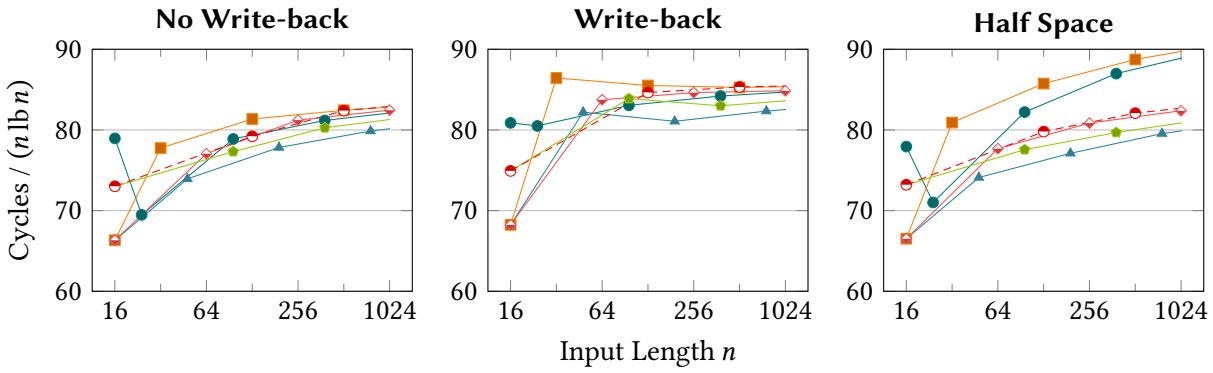
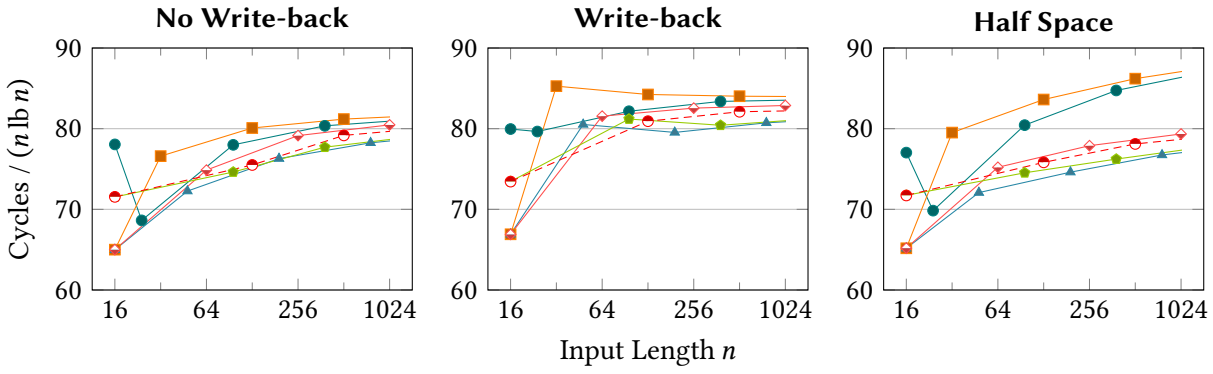


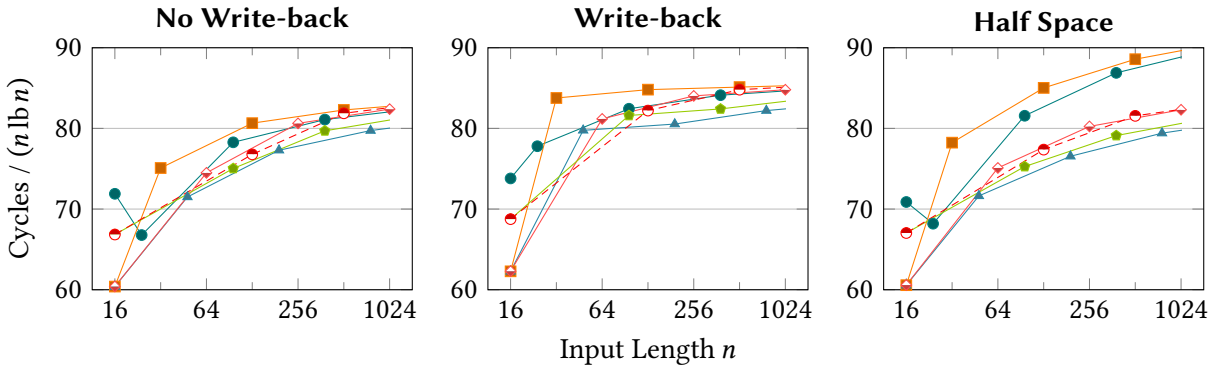
Figure 18: An extension to Fig. 7. The data type is 32-bit unsigned integers.



(a) Uniform



(b) Zipf's



(c) Normal

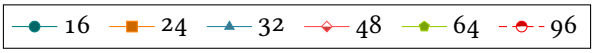
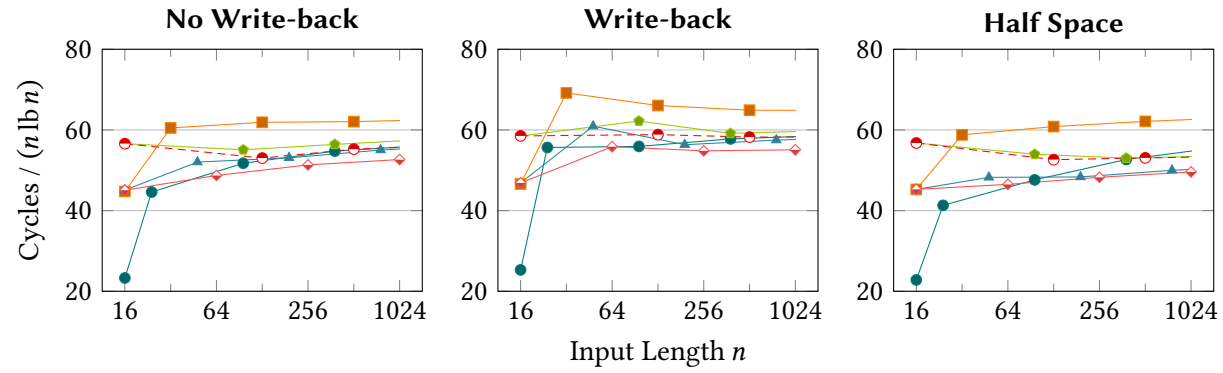
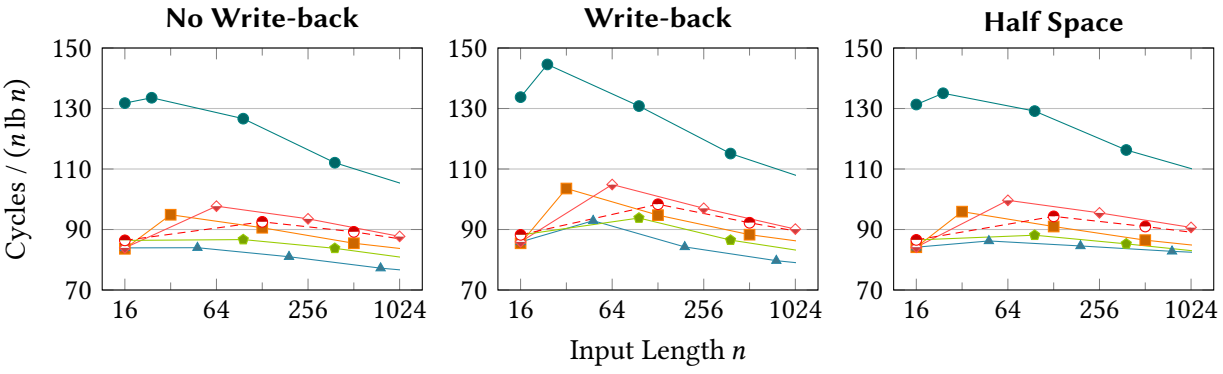


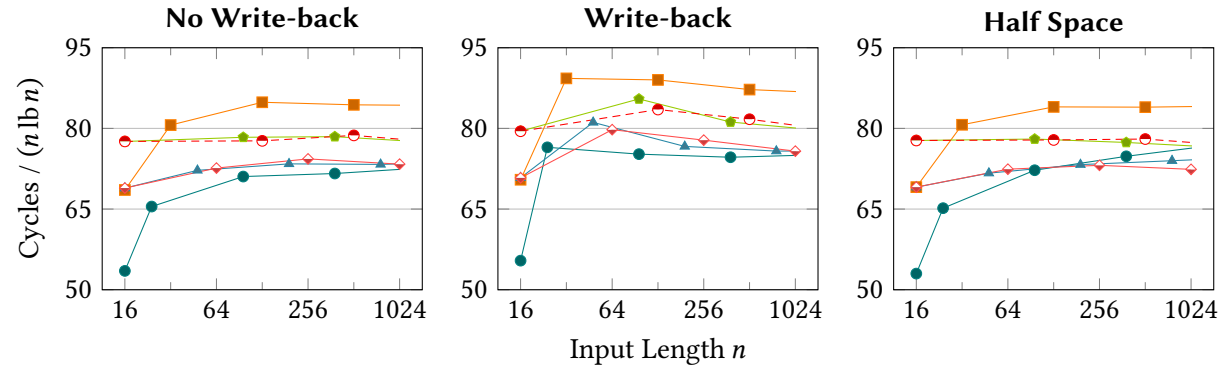
Figure 19: An extension to Fig. 7. The data type is 32-bit unsigned integers.



(a) Sorted



(b) Reverse Sorted



(c) Almost Sorted

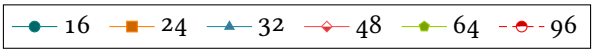
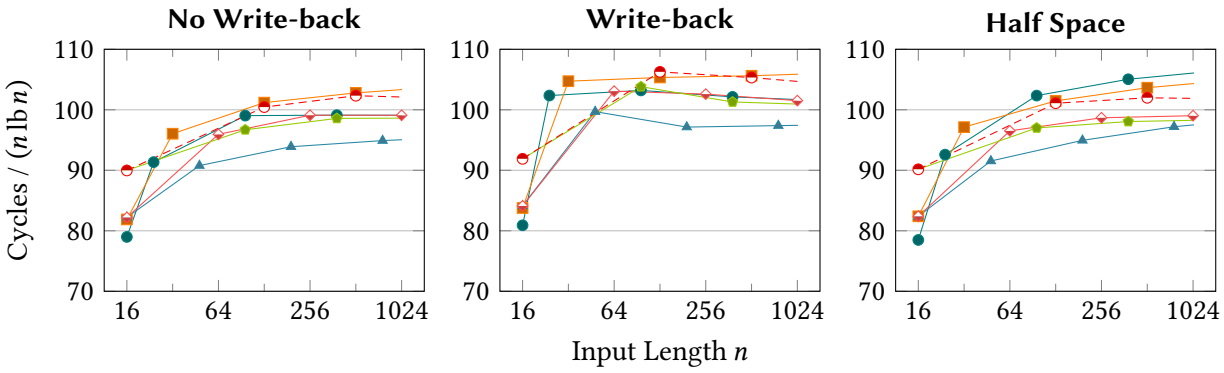
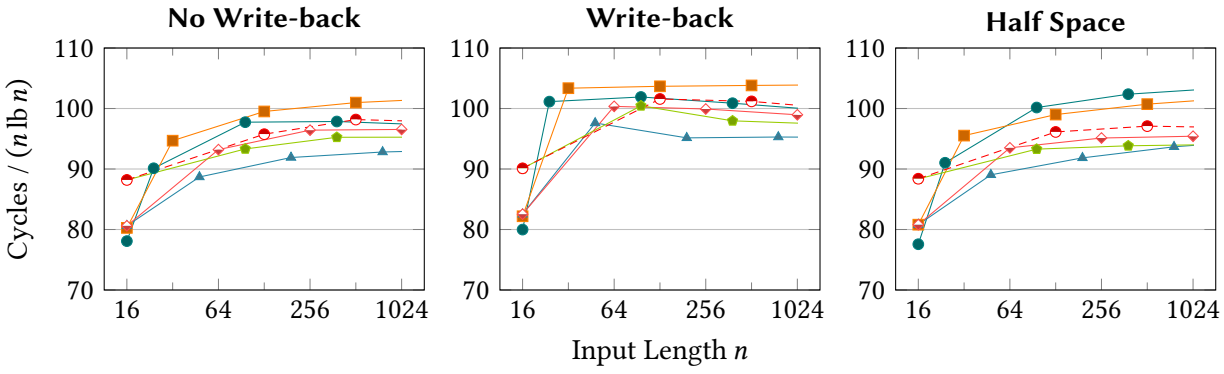


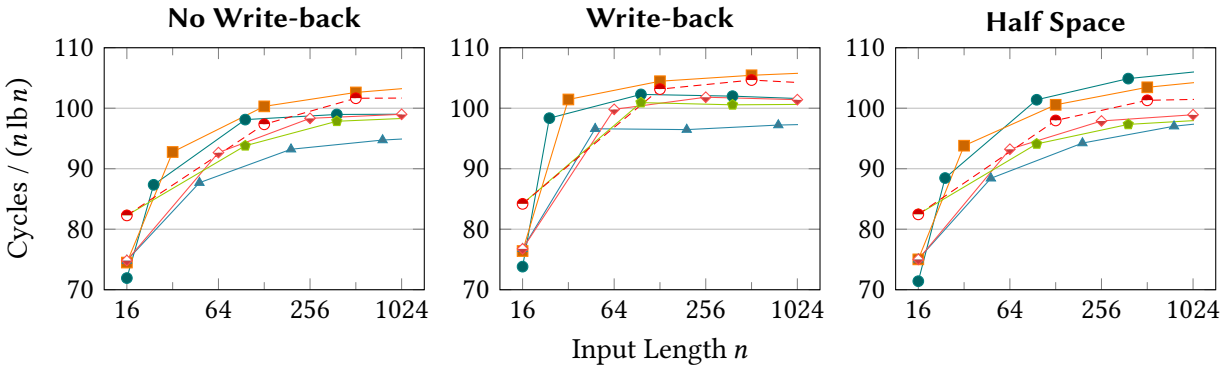
Figure 20: An extension to Fig. 7. The data type is 64-bit unsigned integers.



(a) Uniform



(b) Zipf's



(c) Normal

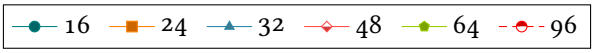


Figure 21: An extension to Fig. 7. The data type is 64-bit unsigned integers.

A.4. HeapSort

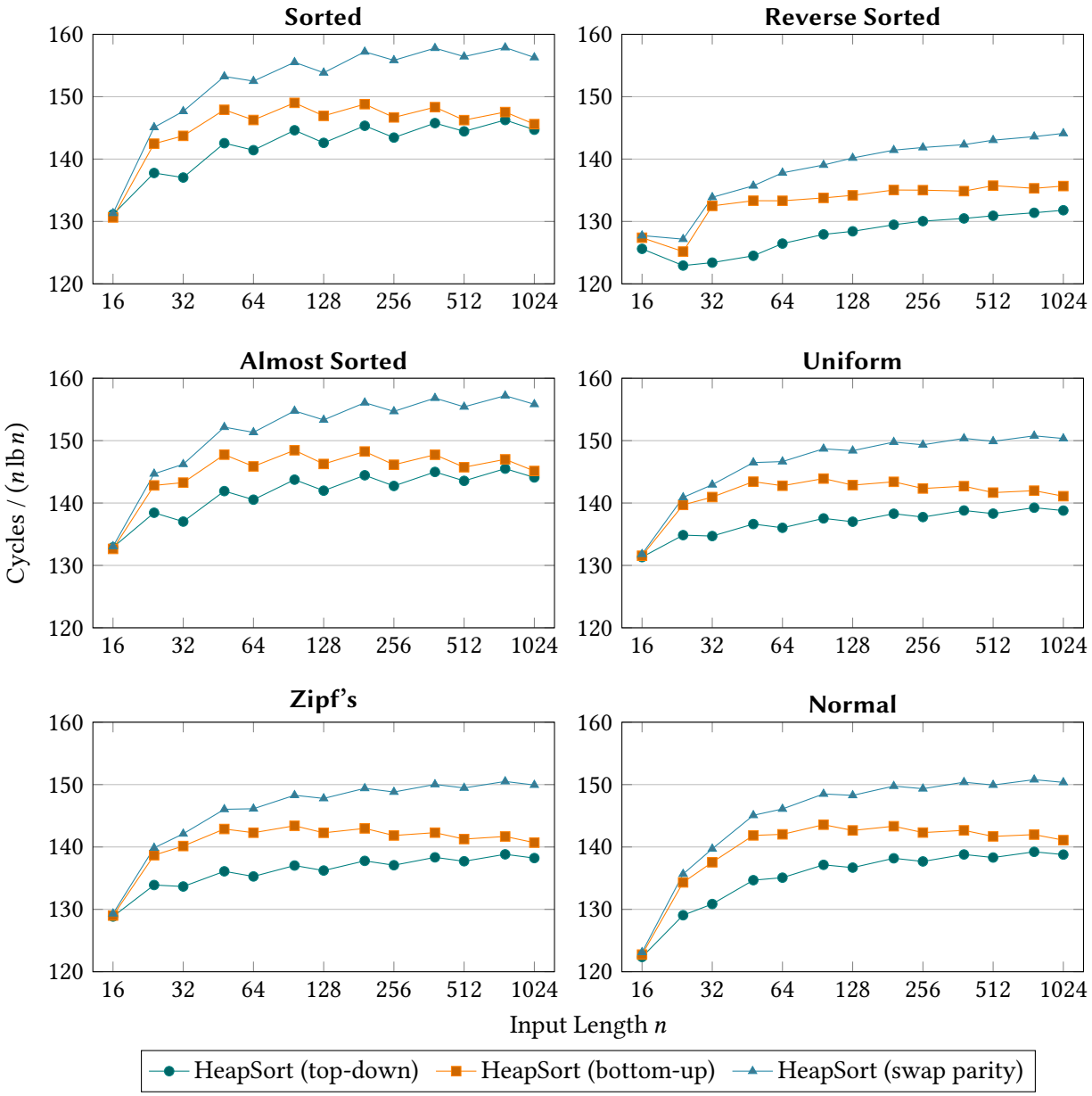


Figure 22: An extension to Fig. 9. The data type is 32-bit unsigned integers.

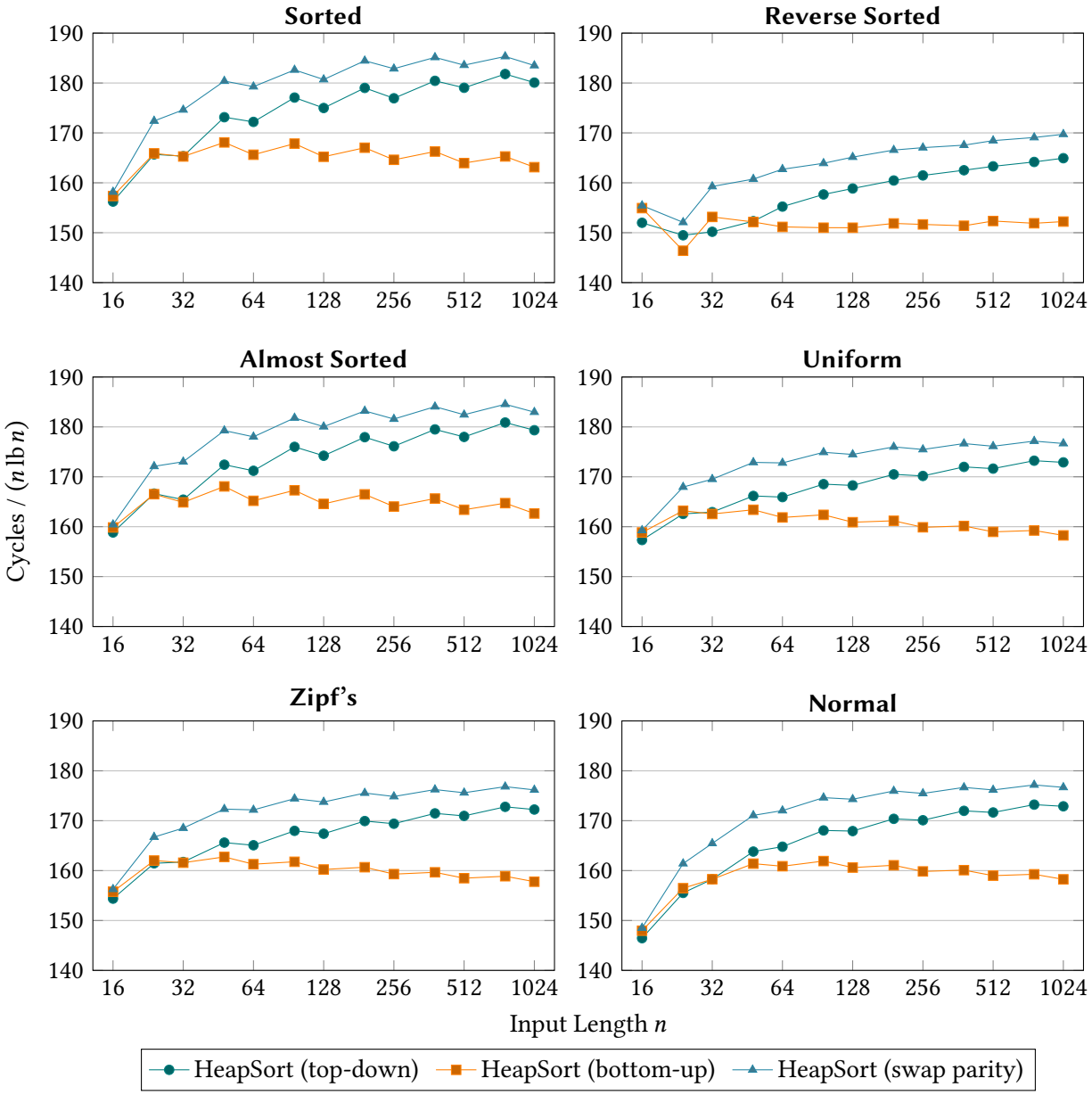


Figure 23: An extension to Fig. 9. The data type is 64-bit unsigned integers.

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.

[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].

[7] Ingo Wegener. 'BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small)'. In: *Theoretical Computer Science* 118 (1 13th Sept. 1993), pp. 81–98. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90364-Y.