Contents

Todo list

Architektur

Speicherzugriffe (memcpy, mram_read ...)

triple buffer

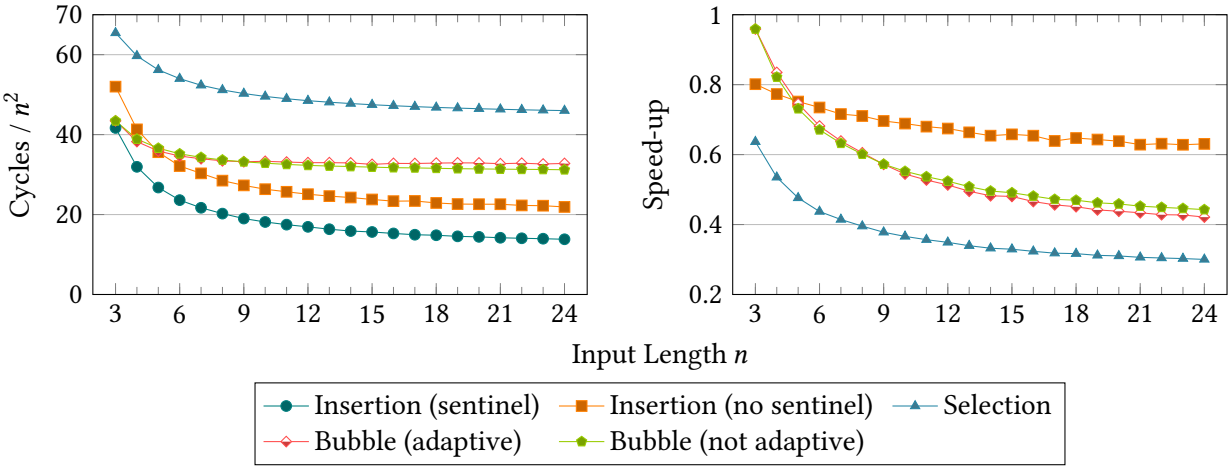

Figure 1: Comparison of sorting algorithms with a runtime in $O(n^2)$. The adaptive BubbleSort terminates prematurely if no changes were made to the input array during an iteration. The speed-ups are with respect to the InsertionSort relying on sentinel values.

1 Sorting with One Tasklet

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

InsertionSort using one sentinel value

ShellSort using h_1 sentinel values

QuickSort recursive implementation; switching to InsertionSort whenever 13 elements or less remain in a partition; median of three as pivot

1.1 InsertionSort

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

Beleg?

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

auf Kom-ex- und implizite Wächter-werte ben-ennen

auf Compilersperenzchen eingehen?

Note. Other known simple sorting algorithm with similar runtime complexity are SelectionSort

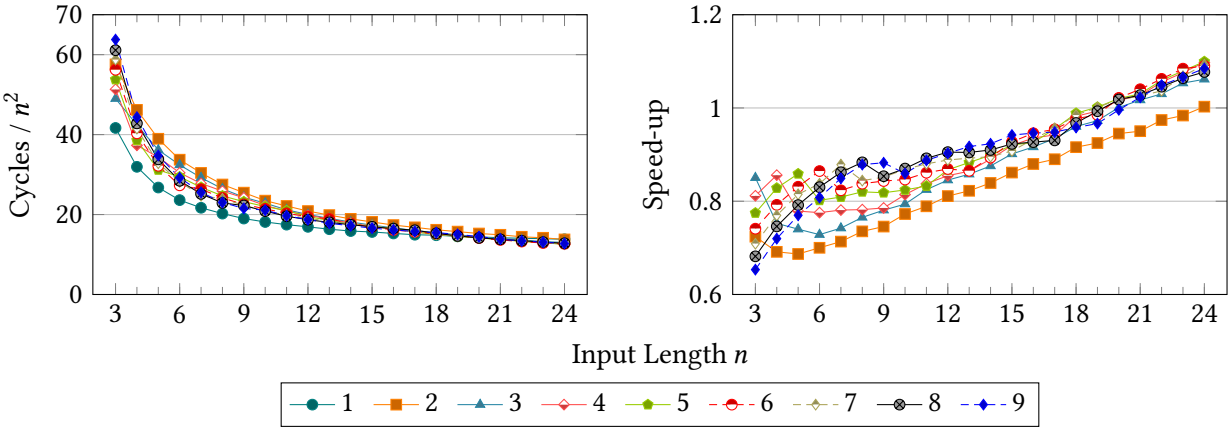

Figure 2: Comparison of InsertionSort (1) and various ShellSorts (2–9). Each ShellSort does one InsertionSort pass with a step size between 2 and 9 before doing a pass of regular InsertionSort. The speed-ups are with respect to the InsertionSort.

and BubbleSort. The asymptoticity, however, hides much higher constant factors such that even for as little as three elements InsertionSort is superior (Fig. 1) and should always be preferred.

1.2 ShellSort

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

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

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

But would pushing the limits of ShellSort even be rewarding? Two issues come up. Firstly,

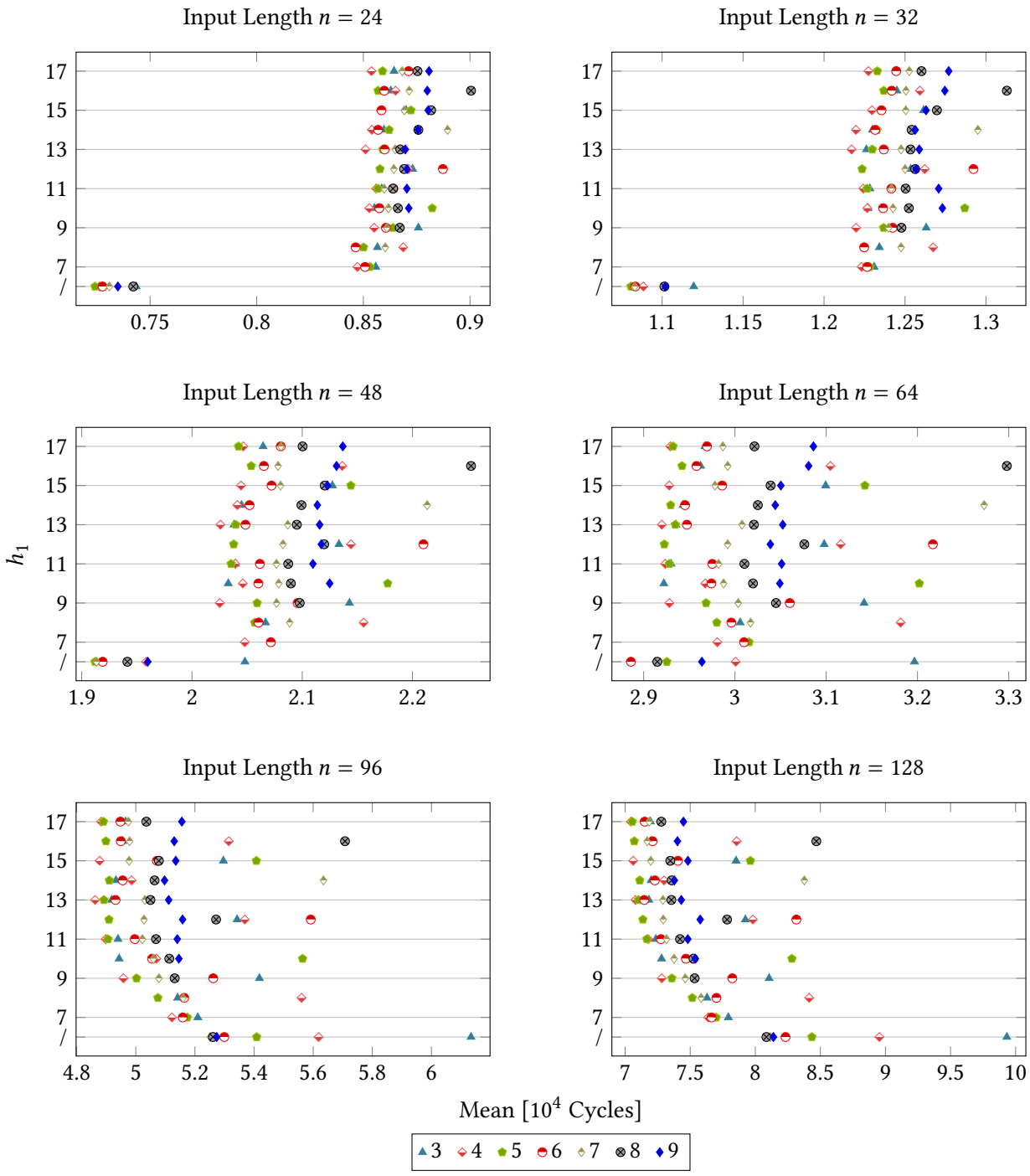

Figure 3: Runtimes of ShellSorts with two rounds (/) and three rounds (7–17). The coloured symbols encode the step size h_1 for two-round ShellSorts and the step size h_2 for three-round ShellSorts. For the latter, the step size h_1 is noted on the y-axes.

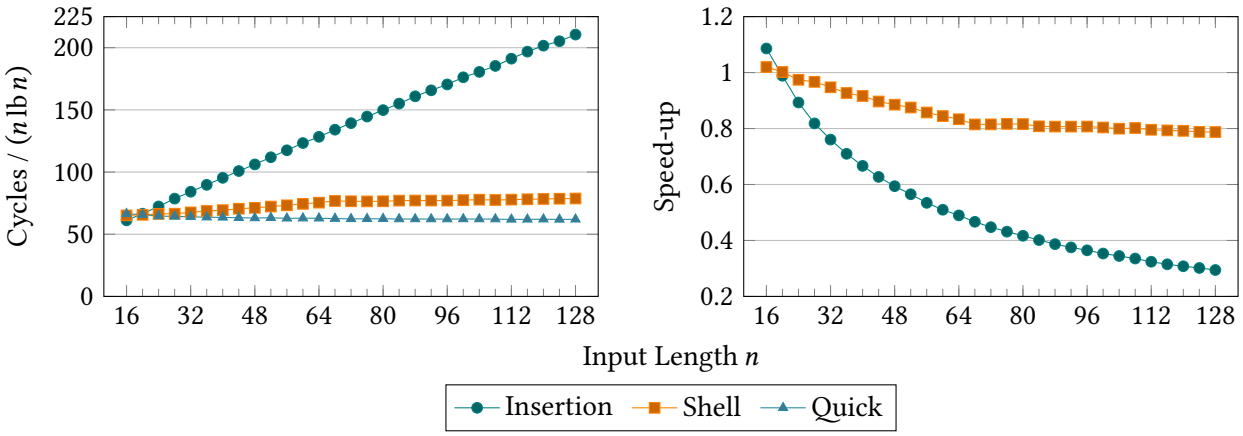

Figure 4: Comparison of InsertionSort, ShellSort and QuickSort. The ShellSort uses the steps sizes $h = (6, 1)$ for $n \leq 64$ and $h = (17, 4, 1)$ elsewise. The speed-ups are with respect to the QuickSort.

QuickSort mit Zufallspivot auch noch einbauen

verschiedene Verteilungen

greater input lengths require greater steps — well into the three digits for $n \approx 1000$ [1, 4] — and those in turn require more sentinel values. But the more sentinel values are stored, the less space is available for the actual input array, leading to smaller runs and thus hurting the overall sorting algorithm. Explorative testing suggests that falling back to bounds checking for big steps is too punishing. Secondly, there simply are better alternatives, namely QuickSort (which will be discussed in more detail in the next section). Figure 4 shows that even though ShellSort takes just a fraction of the time InsertionSort takes — apparently achieving a runtime between $\Omega(n \operatorname{lb} n)$ and $O(n \operatorname{lb}^2 n)$ —, QuickSort beats both from 20 elements onwards. Even QuickSort's standard deviation of 2237 cycles is superior to ShellSort's 2670 cycles (for 128 elements). Together with Fig. 2, this means that ShellSort is not worth using at all and will, consequently, not be improved upon in this thesis.

1.3 QuickSort

QuickSort uses partitioning to sort in an expected average runtime of $O(n \log n)$: A pivot element is chosen from the input array, then the input array gets scanned and elements bigger or smaller than the pivot are moved to the right or left of the pivot element, respectively. Finally, QuickSort is called 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 alternative. As Fig. 5 demonstrates, the optimal threshold for switching the sorting algorithm is around 13 elements, netting a speed-up of between a quarter and a half compared to 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 Section 3.1.

2 References

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

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

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

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

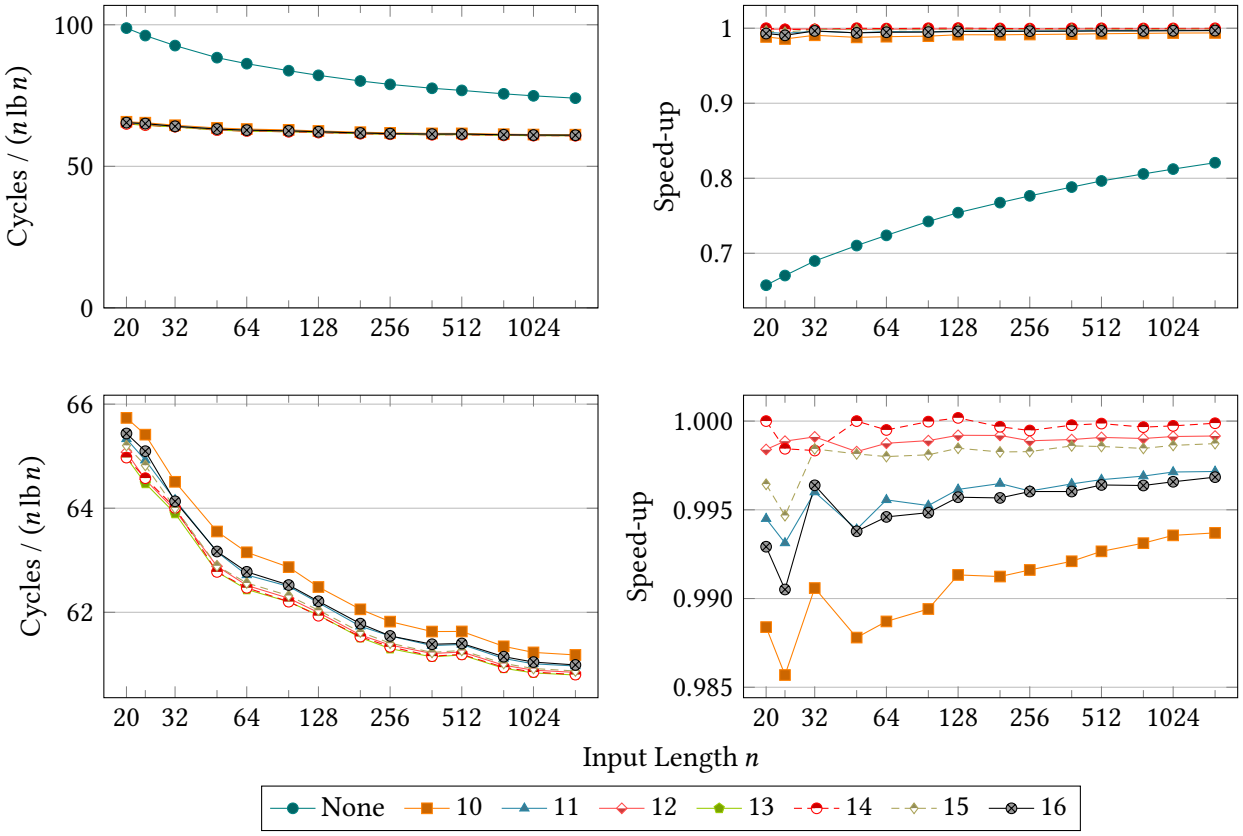

Figure 5: Comparison of QuickSort without fallback algorithm (None) and QuickSorts using InsertionSort if only a few elements remain in a partition (10–16). The bottom row is zoomed in on the latter. The speed-ups are with respect to the QuickSort without fallback algorithm.

3 Appendix

3.1 QuickSort

Two base cases exist: 1. If there are less than two elements left, terminate. 2. If there are less than 14 elements left, call InsertionSort. The first base case is not strictly necessary as the second one covers it. The following implementation takes 623 000 cycles on average for 1024 elements:

- If the partition has a length of 1 or less, terminate. If not and if the threshold is undercut, sort with InsertionSort. Otherwise, sort with QuickSort and call QuickSort on both the left and right partition.

In comparison, the following implementations take between 715 000 and 725 000 cycles, which is 15% more:

- If the threshold is undercut, sort with InsertionSort. Otherwise, sort with QuickSort and call QuickSort on both the left and right partition.
- If the threshold is undercut, check whether the partition has a length of 1 or less and either terminate or sort with InsertionSort. Otherwise, sort with QuickSort and call QuickSort on both the left and right partition.
- If the threshold is undercut, sort with an InsertionSort which terminates if the given array has a length of 1 or less. Otherwise, sort with QuickSort and call QuickSort on both the left and right partition.
- If the threshold is undercut, sort with InsertionSort. If not, terminate if the partition has a length of 1 or less. Otherwise, sort with QuickSort and call QuickSort on both the left and right

partition.

- Sort with QuickSort. Check if the partitions undercut the threshold and either call InsertionSort, QuickSort, or nothing on them.
- Sort with QuickSort. Check if the partitions have a length of 1 or less or at least undercut the threshold and either call InsertionSort, QuickSort, or nothing on them.

Even this implementation is a bit slower at 625 000 cycles:

- If the threshold is undercut, sort with InsertionSort. Otherwise, sort with QuickSort. Check if the partitions have a length of 1 or less and call QuickSort on them if not.

r0 start

r1 end

r23 return address

```
insertion_sort_nosentinel:
    jleu r0, r1, .LBB2_1  // Continue if array of positive length …
.LBB2_8:
    jump r23  // … else leave the function.
.LBB2_1:
    move r2, r0, true, .LBB2_2  // i ← start; Jump to beginning of outer loop.
.LBB2_5:
    move r4, r5  // ?
.LBB2_7:
    add r2, r2, 4  // i++
    sw r4, 0, r3  // *curr ← to_sort
    jgtu r2, r1, .LBB2_8  // If i > end, terminate.
.LBB2_2:  // Beginning of outer loop.
    lw r3, r2, 0  // to_sort ← *i;
    add r5, r2, -4  // prev ← i - 1
    move r4, r2  // curr ← i
    jltu r5, r0, .LBB2_7  // If prev < start, skip to the next iteration of the outer loop.
    move r5, r2  // (prev + 1) ← i
.LBB2_4:
    lw r6, r5, -4  // *prev
    jleu r6, r3, .LBB2_5  // If *prev > to_sort, terminate inner loop.
    add r4, r5, -4  // Store prev.
    add r7, r5, -8  // Store prev--.
    sw r5, 0, r6  // *curr ← *prev
    move r5, r4  // curr ← prev
    jgeu r7, r0, .LBB2_4  // If prev >= start, continue with the next iteration of the inner l
    jump .LBB2_7  // Continue with the next iteration of the outer loop.

insertion_sort_sentinel:
    jleu r0, r1, .LBB3_1  // Continue if array of positive length …
.LBB3_5:
    jump r23  // … else leave the function.
.LBB3_4:
    add r0, r0, 4  // i++
    sw r3, 0, r2  // *curr ← to_sort
    jgtu r0, r1, .LBB3_5  // If i > end, leave the function.
.LBB3_1:
```

```
    lw r2, r0, 0  // to_sort ← *i
    lw r4, r0, -4  // *prev
    move r3, r0  // curr ← i
    jleu r4, r2, .LBB3_4  // If *prev > to_sort, terminate inner loop.
    move r3, r0  // ???
.LBB3_3:
    sw r3, 0, r4  // *curr ← *prev
    lw r4, r3, -8  // *(prev - 1)
    add r3, r3, -4  // curr ← prev
    jgtu r4, r2, .LBB3_3  // If *(prev - 1) > to_sort, continue with the next iteration of the
    jump .LBB3_4  // Leave inner loop.
```