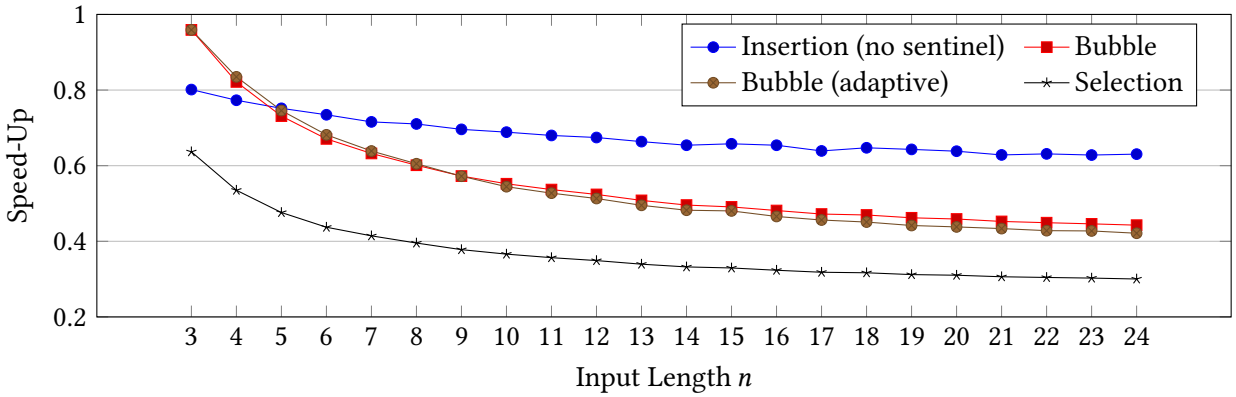

Figure 1: The speed-ups of some sorting algorithms over an InsertionSort using sentinel values with uniform input distribution. Compared to normal BubbleSort, the adaptive version terminates prematurely if no changes were made to the input array during an iteration. Speed-ups below 1 indicate slow-downs.

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 displays 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.

When moving an element to the left, two checks are needed: Does the left neighbour exist and is it smaller than the element? 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% (Fig. 1).

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

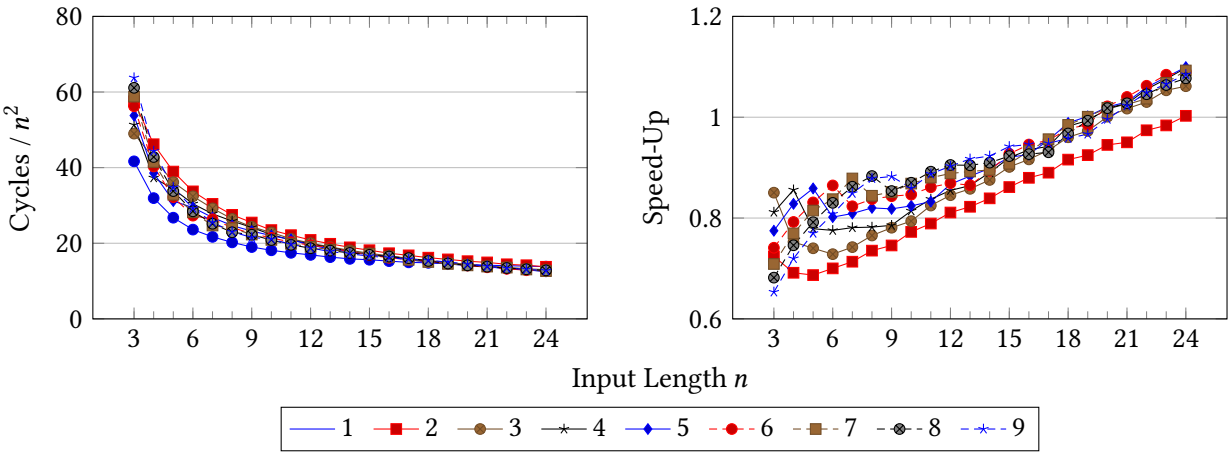

Figure 2: The runtime of InsertionSort and various ShellSorts as well as the speed-ups of the ShellSorts over InsertionSort with uniform input distribution. Each ShellSort does one InsertionSort pass with a step size between 2 and 9. Speed-ups below 1 indicate slow-downs.

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.
```