Reinhardt Implementation:

Reinhardt‘s in-place mergesort algorithm consists of two main procedures:

In the first step, 2/3 of the unsorted elements are sorted by using the other 1/3 of the unsorted elements as gap (reinhardt\_gapsort). In the second step, the sorted elements of the first step (called “short” list) are merged with the previous sorted elements (called “long” list) (reinhardt\_merge).

Therefore in a single iteration, we can merge 2/3 of the unsorted elements into the sorted list. Despite this is a full description of an O(n\*log(n)) in-place-mergesort, we need a lot of time to swap all elements after one iteration as well as in the asymmetric merge in step two. Therefore we implemented some improvements, most of them also described in the paper:

In the whole algorithm we regard our elements as a ring list to skip the moves after one iteration (\*\_ring). Hereby we “overloaded” the dereference operator so that every iterator points on the shifted element. But because of the worse results and the non-compatibility with the following improvements, we just continued without the ring list and leave a more detailed explanation.

We also implemented two other procedures that can be executed instead the “usual” step (quick\_steps). They perform a Quickselect- or Quicksort-step on the unsorted elements and then merge them with a part of the long list. After this step, the merged elements are on their correct position and do not need to be considered any more.

In the following, we give a more detailed description of the stated procedures:

**In\_place\_mergesort basic routine** (inplace\_mergesort\_[qsel / qsort])

Firstly, we sort 4/5 elements of the whole list by using 1/5 as gap. This is possible in the first iteration because we do not need to consider the gap for step two. After that, we call the recursive procedure which executes the two steps mentioned above. We implemented an integer-flag where the user can choose the frequency of quicksort or quickselect iteration compared to the usual iteration. A quicksort step can leave the new gap on the right side, therefore we needed to implement an analogous second procedure for this case. When the gap size undercuts a short number, we execute one parallel insertion sort step for the left block by use of binary sort for efficiency reasons.

**Step one** (reinhardt\_gapsort)

We sort the elements between the passed iterators by using a gap of size / 4 on the left or on the right side. Hereby we firstly sort four quarters of the list with the usual recursive mergesort (rec\_mergesort\_iter) and the gap as “extra storage”. Therefore we permanently have to swap the gap elements instead of just assign to the extra memory.

Afterwards we merge the four quarters in a way quite similar to the symmetric merge procedure described below (step two).

Step one can also be used as a full sort algorithm with extra space size / 4 (reinhardt\_extrasort). The number of assignments is obviously bigger for the in-place-mergesort caused of swapping instead just moving. But extra space of size / 4 is only just enough to avoid any additional costs for swapping or shifting.

**Step two** (reinhardt\_merge)

The merge procedure expects the longer list being between the gap and the shorter list and a minimal gap size of half of the shorter list. The merge direction changes after a “collision” with the longer list, exactly as described in the paper (chapter two).

To reduce the number of comparisions, we also implemented an asymmetric merge variant. Hereby we compare the first element of the shorter list with the 2n’th element of the longer list whereas n depends on the relation of the list sizes. Afterwards the whole block of the longer list can be moved or a binary search in the block is needed. The case 2n = 1 is equal to the usual symmetric merge.

**Quickselect and Quicksort iteration** (quick\_steps)

The quickselect iteration extracts the 2/3- smallest element of the gap and partitions the gap elements around. Note that this step should not exceed O(n\*log(n)) in worse case to remain O(n\*log(n)) as worse case runtime for the whole sort. Afterwards, the small-elements partition is sorted with step one by using the big-elements partition as gap. Now we determine the fitting small-elements list of the long list by binary sort and then the two small-elements lists are merged (after a possibly necessary swap of the lists).

Although there is a non-negligible overhead for the quickselect procedure, all the merged elements are on their correct position and the gap size is reduced same as in the usual step.

The quicksort iteration as described in chapter five does not guarantee a specific reduction of the gap size, but it moves half of the long list on the correct position. Therefore the size of the remaining “long” list could get very small and in this case we just continue to execute the usual iteration in the basic routine for efficiency reasons. The quicksort step works as follows:

After quicksort on the unsorted elements with the middle element of the long list as pivot, we sort the partition of the bigger or the smaller elements depending on the size of the partitions. Before the following merge with the bigger or the smaller half of the long list, we possibly need some swaps to get the lists to be merged in the right order. After the merge, the remaining gap may has changed on the other side and in this case the other recursive basic routine is called.