**Test results**

In the following we will describe and analyse the results of the tests. Although it is not possible to compare all versions of the in-place merge sorts with all possible test conditions, we try to cover a broad range to figure out the main points. The raw data and further results are given in the “Test\_Diagramme” file 🡪 **File schöner strukturieren, Kommentare löschen etc.**

For all the tests given here, we sorted equally distributed integer values in multiple runs (of course more runs for smaller lists) on the server. We start by comparing our in-place merge sort implementations between each other. **Time integer Diagram**

Integers do not need much time to be assigned. Therefore the time in this test is mainly influenced by other statements like index computations. This can also be seen by a comparison with the following result. **Big Type with expensive comparison Diagram**

The second result is mainly influenced by the number of comparisons and assignments (see below). As one can see, the used algorithm or version of algorithm should be chosen depending on the type for better performance. For a universal measure, independent of type, compiler and operating system, the number of comparisons and assignments can be evaluated **(Comp/Assignm).**

The standard merge sort does roughly one assignment after one comparison and has only a small additional number of assignments (*see result extra file*). One can see that all our in-place versions need roughly just two assignments per comparison.

In a lot of diagrams, we show the results divided by n\*log(n). Therefore, time, comparisons and assignments should be nearly constant throughout different list sizes for all mergesort implementations. To conclude to other specific implications, another scale may be used. E.g. the following diagram can be used as prove that the number of comparisons does not exceed 1.3n\*log(n) + O(n). If this number is not exceed then the values do not increase for an increasing list size.

**Comparisons with another normalisation.** Although diagrams with n\*log(n) instead of 1.3\*nlog(n) also do not show a clear increase, it is impossible to use this diagram as a prove because of the fluctuating values in this case.

For a clearer comparison of Huang and Chen it is possible to just compare the merges because they can be substituted in the algorithm (*results see extra file*). But here it is important to mention that Reinhardt’s merge works completely differently because it merges to another range and though the std-inplace-merge merges to the same range, it needs additional memory to just make use of n-1 comparisons.

The runtime of the merges reflects the bad result of Chen in the first test more powerful. But it is important to remind that the runtime also depends on compiler and device. For a usual laptop and another compiler we got the following time average for multiple runs with 10 Mio elements (in the same test!):

**Chen: 148 418 ns Reinhardt: 88 717 ns** **Huang: 99 766 ns StdInpl-merge: 147 056 ns**

And if we make comparisons/assignments more relevant by use of lists with 1 million bigtype<30>, we get an entirely different relation of results for multiple runs on the laptop:

**Chen: 317 783 ns Reinhardt: 309 832 ns** **Huang: 485 642 ns StdInpl-merge: 247 364 ns**

In the above tests, we run the in-place algorithm of Reinhardt by using insertion sort in "reinhardt\_gapsort" for small cases and the asymmetric merge in "reinhardt\_merge". Furthermore we executed the quickselect step in every iteration.

But if we make specific demands to the sort, e.g. a small number of comparisons, we can switch to a more suitable mode. In the following, we will compare the different Reinhardt modes between themselves. In all runs, we use integer as type to be sorted. **Time different iterations**

We can see that Reinhardt’s algorithm is fastest for integers if we just execute quickselect or quicksort iterations. Though these iterations need a bit more comparions for the execution of quicksort or quickselect, the number of assignments is smaller. This clearly can also be seen in the two following diagrams. **Comparisons/Assignments different iterations and merges**

Merging asymmetric is a big improvement for time and number of comparisons (obviously not for number of assignments) of the usual iteration because in the usual iteration, the short list is always merged with all other sorted elements. For the time of the quickselect iteration, the asymmetric merge is, especially for the runtime, no improvement in this test case.

For a further reduction of comparisons and in the case of sorting bigtypes, it is also a good idea to replace insertion sort in the sort of the gap for small cases (here all along 50 elements) by aborting the recursive procedure later (3-smallsort). **Comparisons different iteration and smallsorts**

The result is not surprising because merge sort needs less comparisons than insertion sort. But on the other side, insertion sort is used many times as sort for small cases because it is often executed faster for integers. **Time different iteration and smallsorts**

We also compared our implementations with some existing in-place merge sorts. **Time comparisons with other inplace merge sorts.** In terms of comparisons and moves they are able to compete with the other algorithms. It must be said that some more sophisticated and specialized implementations can slightly outperform our implementations when it comes to the measured time.