Walkthrough (TA said after topic 6)

The sorting algorithms

Insertion sort

Insertion sort is good only for sorting small arrays (usually less than 100 items). In fact, the smaller the array, the faster insertion sort is compared to any other sorting algorithm. However, being an $O(n^2)$ algorithm, it becomes very slow very quick when the size of the array increases. It was used in the tests with arrays of size 100.

Implementation.

Shell sort

Shell sort is a rather curious algorithm, quite different from other fast sorting algorithms. It's actually so different that it even isn't an $O(n\log n)$ algorithm like the others, but instead it's something between $O(n\log^2 n)$ and $O(n^{1.5})$ depending on implementation details. Given that it's an *in-place* non-recursive algorithm and it compares very well to the other algorithms, shell sort is a very good alternative to consider.

Implementation.

Heap sort

Heap sort is the other (and by far the most popular) *in-place* non-recursive sorting algorithm used in this test. Heap sort is not the fastest possible in all (nor in most) cases, but it's the *de-facto* sorting algorithm when one wants to make *sure* that the sorting will not take longer than $O(n\log n)$. Heap sort is generally appreciated

because it is trustworthy: There aren't any "pathological" cases which would cause it to be unacceptably slow. Besides that, sorting in-place and in a non-recursive way also makes sure that it will not take extra memory, which is often a nice feature.

One interesting thing I wanted to test in this project was whether heap sort was considerably faster or slower than shell sort when sorting arrays.

Implementation.

Merge sort

The virtue of merge sort is that it's a truely O(*n*log*n*) algorithm (like heap sort) and that it's stable (it doesn't change the order of equal items like eg. heap sort often does). Its main problem is that it requires a second array with the same size as the array to be sorted, thus doubling the memory requirements.

In this test I helped merge sort a bit by giving it the second array as parameter so that it wouldn't have to allocate and deallocate it each time it was called (which would have probably slowed it down somewhat, especially with arrays of the bigger items). Also, instead of doing the basic "merge to the second array, copy the second array to the main array" procedure like the basic algorithm description suggests, I simply merged from one array to the other alternatively (and in the end copied the final result to the main array only if it was necessary).

Implementation.

Quicksort

Quicksort is the most popular sorting algorithm. Its virtue is that it sorts *in-place* (even though it's a recursive algorithm) and that it usually is very fast. One reason for its speed is that its inner loop is very short and can be optimized very well.

The main problem with quicksort is that it's not trustworthy: Its worse-case scenario is $O(n^2)$ (in the worst case it's as slow, if not even a bit slower than insertion sort) and the pathological cases tend to appear too unexpectedly and

without warning, even if an optimized version of quicksort is used (as I noticed myself in this project).

I used two versions of quicksort in this project: A plain vanilla quicksort, implemented as the most basic algorithm descriptions tell, and an optimized version (called `MedianHybridQuickSort` in the source code below). The optimized version chooses the median of the first, last and middle items of the partition as its pivot, and it stops partitioning when the partition size is less than 16. In the end it runs insertion sort to finish the job.

The optimized version of quicksort is called "Quick2" in the bar charts.

The fourth number distribution (array is sorted, except for the last 256 items which are random) is very expectedly a pathological case for the vanilla quicksort and thus was skipped with the larger arrays. Very unexpectedly this distribution was pathological for the optimized quicksort implementation too, with larger arrays, and thus I also skipped it in the worst cases (because else it would have affected negatively the scale of the bar charts). I don't have any explanation of why this happened.

Implementation.