Programming assignment 3 - Emil Collén

Problem 2

For the hash table for the vehicles, we decide when we create the hash table what size it should have by passing an integer as an argument, for example:

```
MyHash h = new MyHash(31);
```

What size we decide to make our hash table is important to our performance. We don´t want to choose a number that is often repeated or divisible many numbers, since then we will get a lot of collisions. That rules out many even numbers and round numbers. We also don´t want to use number that are a powers of two (such as 4, 8, 16, … 256, 512, 1024..). By just avoiding these we can get a decent table size. We can make it even better by choosing a prime number, preferably close to a power of two. In the case above, we use 31. It is close to 32, and not divisible by anything but 1 and itself (a prime). This is a rather small hash table, but we can make it as big as we want – as long as we keep the above mentioned in mind.

Firstly, we want to make sure that the hash code gives an even distribution. When we run 1 million runs of randomizing license plate, year, color and passenger seat, we do get an even distribution on a hash table of size 31. Figure 1 below shows the distribution. We can therefore determine that the randomization is even for following parts.

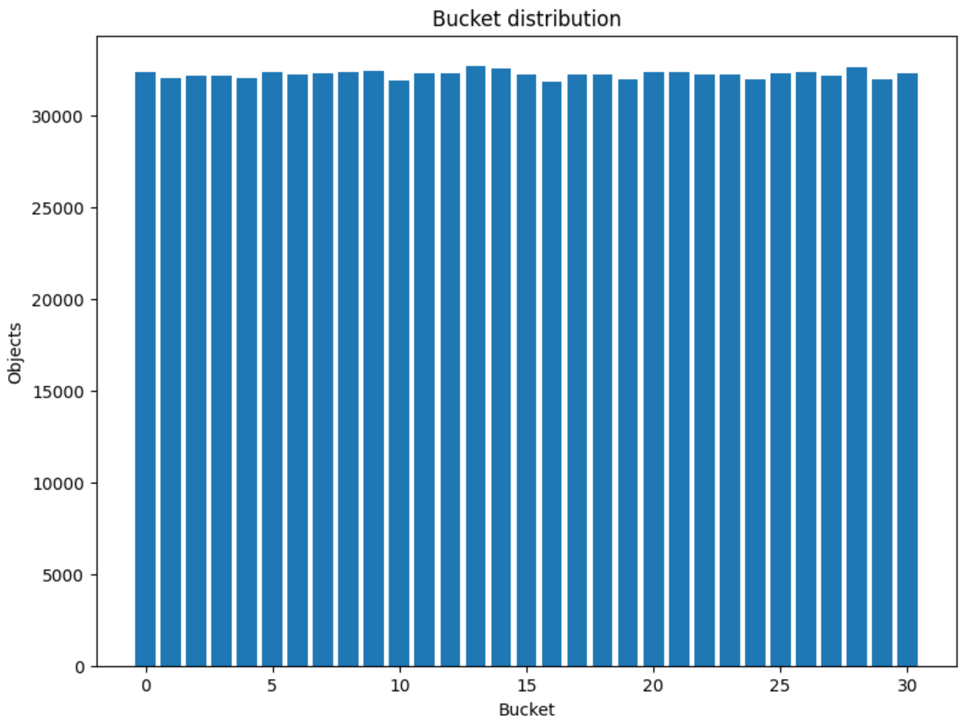

Figure 1 - Bucket distribution for 1million objects on a size 31 hash table.

When we want to analyze the hash table, we can start by stating that the odds of getting an empty bucket (no collision) is reduced the more objects it already contains, starting with $\frac{1}{Hash\ table\ size}$, in this case $\frac{1}{31}$. Since we use quadratic probing, even similar hash values won´t result in a clustered table – at least not with primary clustering, only secondary clustering. It is important that the table size is at least twice the number of entries, as it can potentially fail to insert otherwise.

In figure 2 below, we can see how the number of collisions and the total offset grows. Both are averages of 10k runs of populating a hash table of size 257. Naturally, when we only insert a few objects (cars), there are rarely any collisions, and therefore few offsets. As the number of occupied buckets grows, so does collisions and thereby the offset.

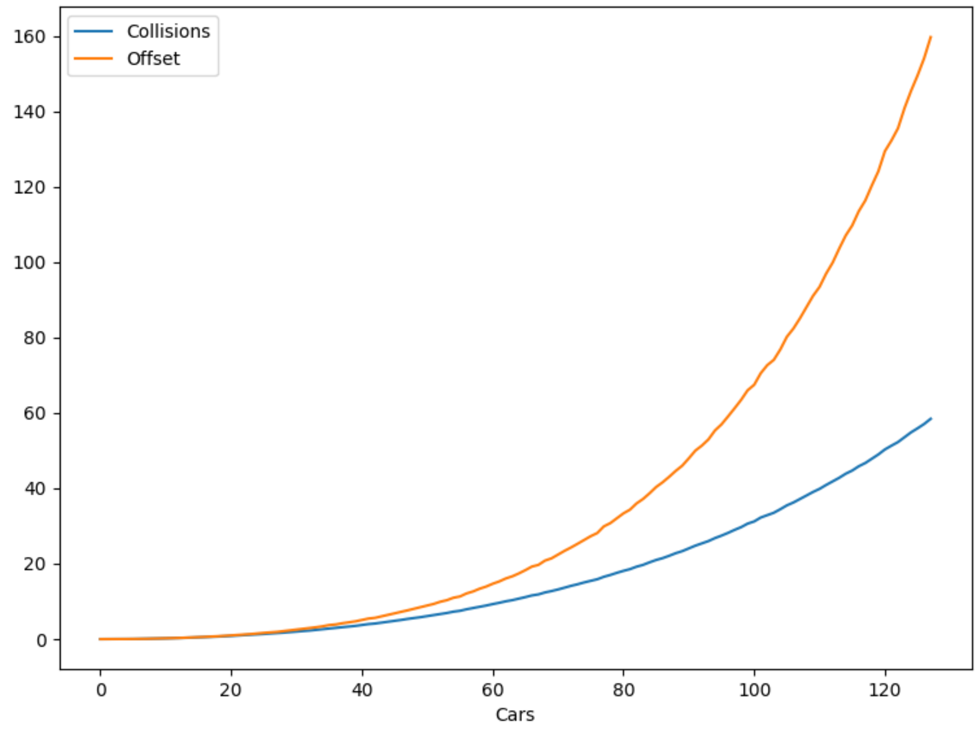

Figure 2 – 127 Inserts on 257 sized table, using quadratic probing

The relation between the offset and the number of collisions is presented below in figure 3. Before we have collisions, the difference is obviously 0, but once the collisions start to occur, the average offset in relation to the number of collisions grows. At first, the average offset is roughly one (same as collisions), but as we approach half-filled hash table, it is more than 2.5 times the number of collisions. The growth is not linear, either. In this example, if we compare the slope between cars (buckets filled), we get that the slope is roughly ~0.002 between 11-12, and roughly ~0.015 between 120-121. That means that the more our hash table grows, not only does the number of collisions grow – the offset of which the inserts are using is growing.

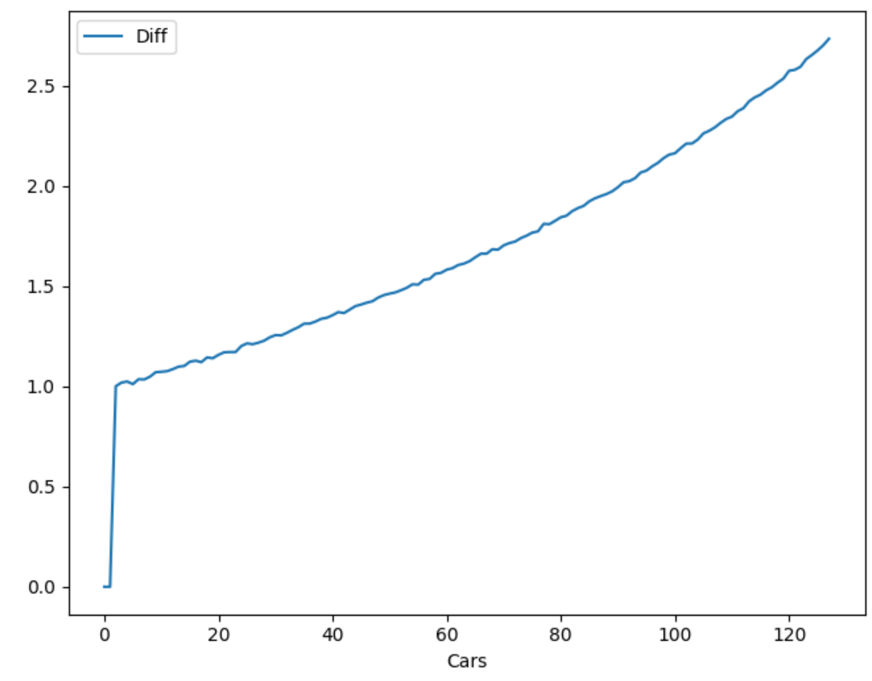

Figure 3 - The relation between offset and collision (offsets/collisions)

Presented in figure 4 below, we make 200 inserts into the same hash table. We can immediately see that the offset is skyrocketing once the table starts to fill up. Once this happens, not only is the offset getting extreme as we jump around the table time and time again to find an empty bucket, we also likely miss some of the inputs as the table is too crowded. As the average offset is roughly 1400(!!), we loop over the table more than 5 times, each time we try to insert with each iteration.

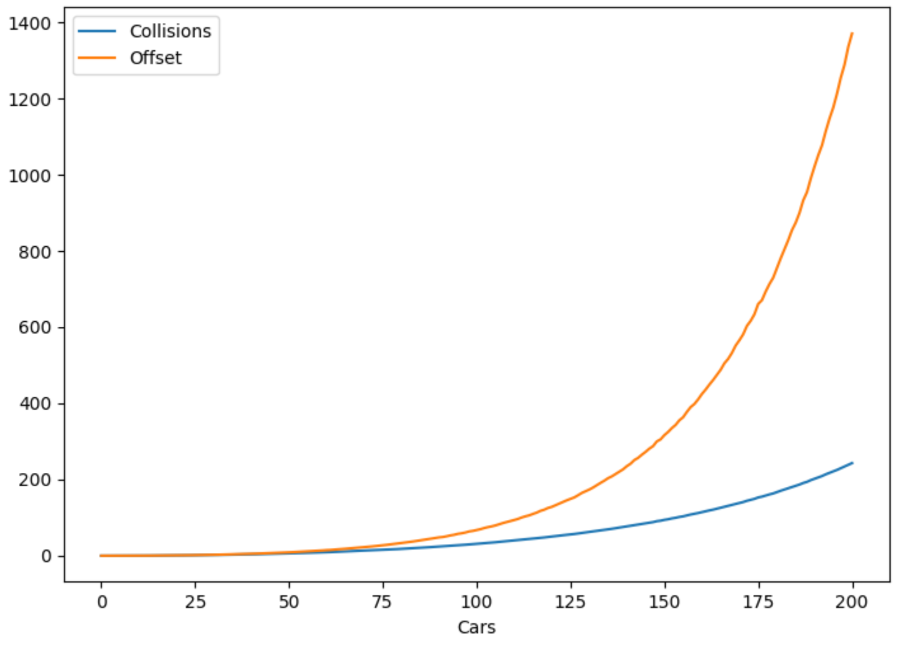

Figure 4 - 200 Inserts on 257 sized table, using quadratic probing

It is even more clear in figure 5 below that the growth in offset in relation to collisions is not linear. As we get closer to 200 objects in out 257 sized table, the average offset is above 5 times the number of collisions.

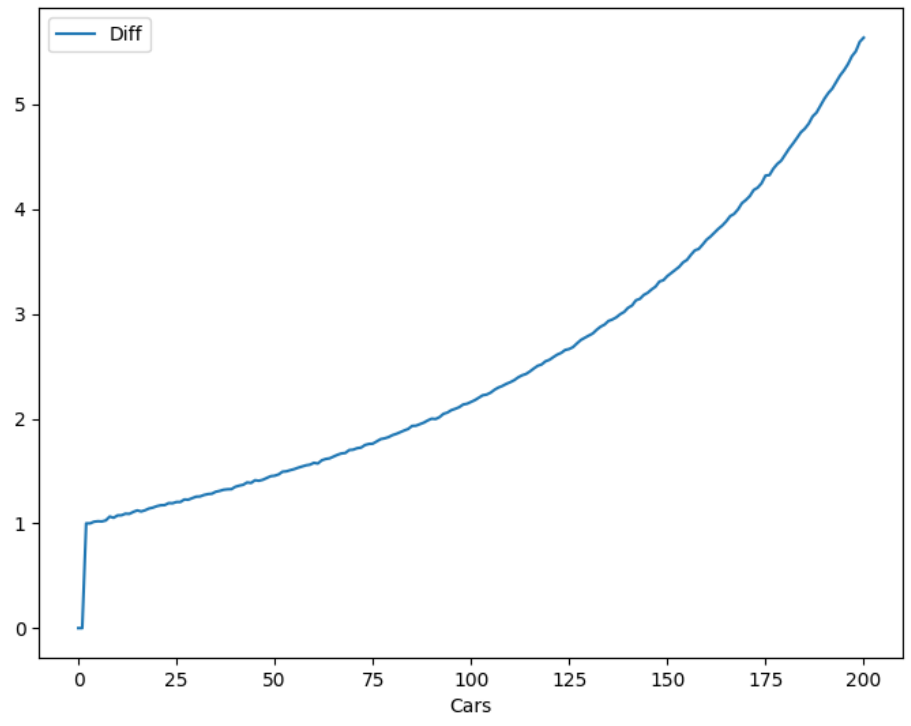

Figure 5 - The relation between offset and collision (offsets/collisions)

Comparing quadratic probing to linear, we can determine that even though both perform poorly on tables that are filled more then ½ of the table size. Linear probing, however, does not fail to insert the same way quadratic probing can. It continues to look until an empty bucket is found, whereas the quadratic probing runs the risk of looping though the table, table-size times, without finding an empty bucket. Using both probing techniques, the risk if clustering is always present – but in different ways. Linear probing risks having primary clustering (in one part of the hash table), while the quadratic probing risks secondary clustering (spread out across the table, but in quadratic intervals).

Problem 5

For this task, I used 1 million element arrays, and I ran the depths 2-40 with interval 2 (2, 4, 6… 36, 38, 40) and between 5-50 iterations to get an average time. Most ran 50 times, except from insert sorts with only very few in depth, due to time restraints. We know that on short arrays and almost sorted arrays, insert sort performs very well. On the other hand, very long unsorted lists are terrible to sort using insert sort. Keeping this in mind, the result below is not that surprising. As we can see in figure 6, when using very few in depth on our 1 million sized array, sorting the array using quicksort with insert sort was extremely slow. Each time we recurse one step deeper, the array gets significantly more sorted and the sizes of which the insert sort operates gets shorter, therefore we can see a very fast drop in the time taken for insert sort.

Compared to quick sort with insert sort, both regular quick sort and quick sort with heap sort roughly takes the same time, no matter which depth. As we can see in figure 7, at very low depths there is a difference. For depth 2 and 4 for example, the quick sort with heap sort takes roughly 40% longer, but compared to quick sort with insert sort, they are very similar. Since regular quick sort has a worst case of $O(N^2)$ (which is very unlikely, since lists are randomized), we want to remove that case to remove the risk of having such cases. By using this hybrid sorting technique, we get that the worst case is O(N*log(N)), which is the same as its average. Regular quick sort also has the average case of O(N*log(N)).

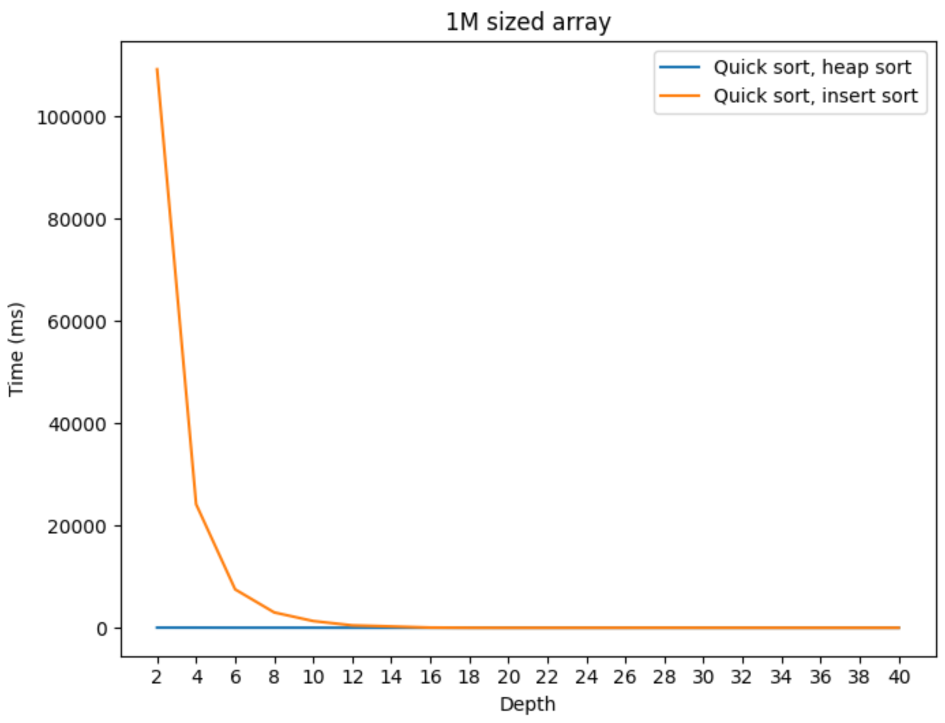

Figure 6 - Quick sort with heap- vs insert sort

If we look at figure 7 below, we compare the regular quick sort with quick sort using heap sort. In this comparison, we can see that the regular quick sort is faster than the hybrid version at very low depths. But like mentioned above, we ensure that we don´t get the worst case of O(N^2) with the hybrid version, which in most cases is a decent trade-off. If we use say log(N) as depth, that is roughly 20, we get a very good result as both the time is good as well as removed risk of O(N^2). It even appears to be slightly faster than regular quick sort.

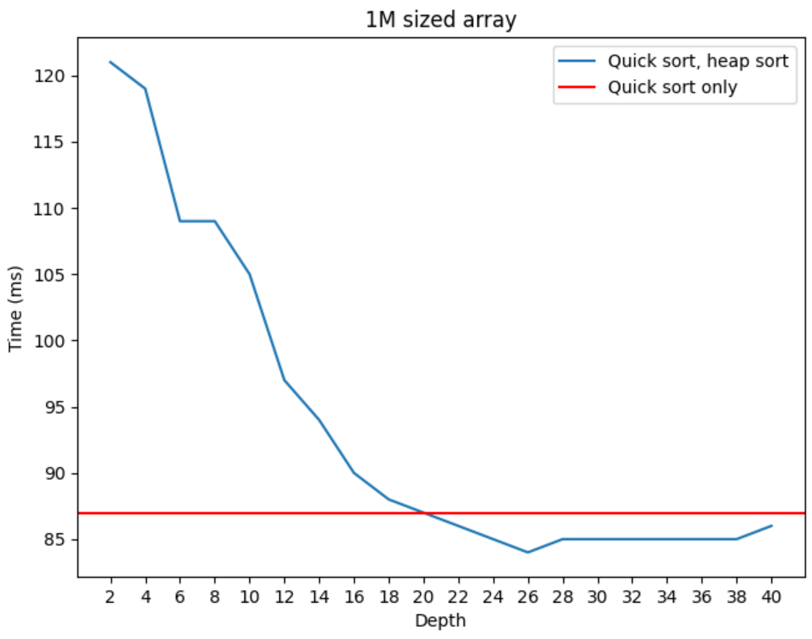

Figure 7 - Regular quick sort vs quick sort with heap sort

If we zoom in on when the insert sort and heap sort versions of the quick sort intersects, like in figure 8 below, we can see that once the depth is enough (at around log(N)), in other words, when the array is sorted enough and the sub-arrays are small enough, we get that they all take **roughly** the same amount of time.

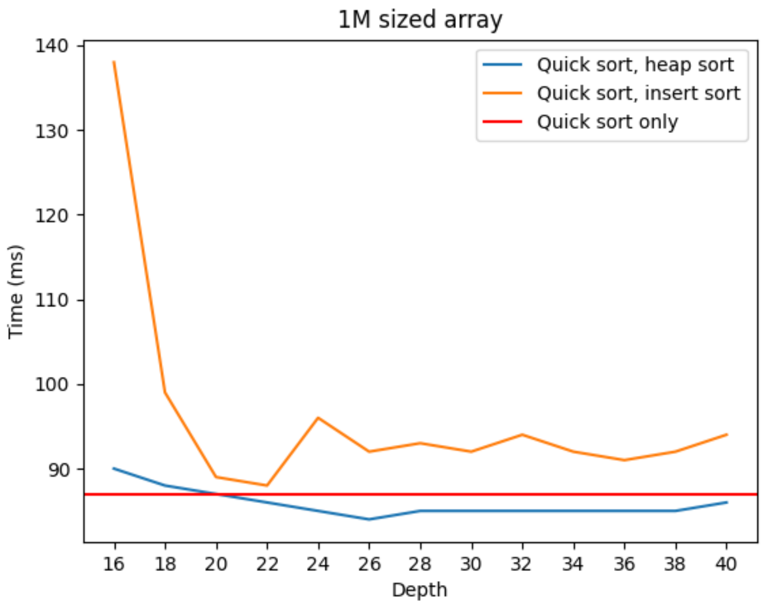

Figure 8 - Looking at deeper depths

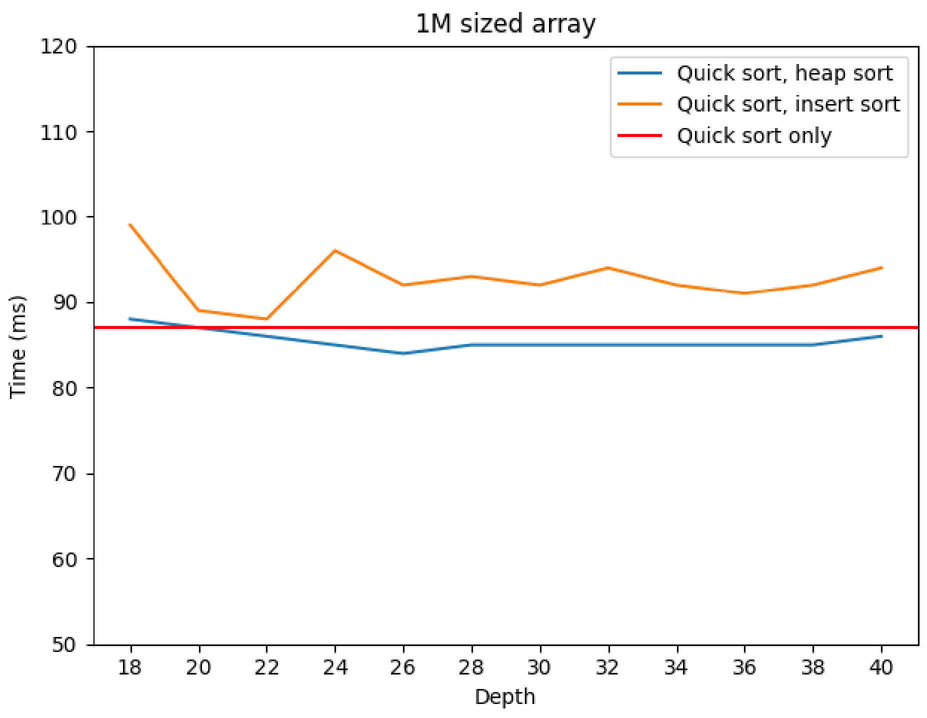

Figure 9 – From depth 18 to 40, a bit zoomed out

What we can conclude from the results, is that on average cases, regular quick sort is very fast. But by combining it with other techniques, we eliminate the risk of the worst case scenario. Since we use random input data, the regular quick sort is represented by a constant line, as it is the average case. In real scenarios however, we run the risk of degenerate input, resulting in terrible time complexity. Both types of hybrid sorting aim to avoid this case. The quick sort with heap sort appears to be faster than regular quick sort once the cut-off depth is enough. The quick sort with insert sort is terrible on long and unsorted lists, quite decent once it is sorted enough and the sub-arrays are small enough. When it is, it is basically as fast as the other types, as we can see in figure 9 above. As they all have O(N*log(N)) in the average case, I would say that quick sort using heap sort is the best choice, as we can always expect O(N*log(N)), no matter the array or depth. At depth above log(N), it appears to be the fastest way. It is the safest way as it doesn´t have bad cases like the other techniques. For quick sort with insert sort, if we set log(N) as the depth for which to switch to insert sort, we get a good result as $\log_2(N)$ is roughly 20. Using this depth, we see that it is roughly as fast as regular quick sort while we have removed the worst case. If we say that the cut off depths is just below log(N) to switch to insert sort, that would likely be a good call. This is also a good cut off-depth for quick sort with heap sort as well. It doesn´t show in these cases, since the input is random, but using log(N) as cut off is a good way to ensure that we eliminate worst cases yet don´t spend too much on overhead operations. Basically, both types of swap from quick sort is to avoid worst cases, and the preferred method depends on input data, but with, say, quick sort using heap sort, we know we always get a good N*log(N), no matter the size or order of the data.