Computer Science Department
VU Amsterdam, 1081HV Amsterdam

Bachelor Thesis

Sorting Evolving Data: An Experimental Study

Author: Yasin Kerim Karyağdi (2745681)

1st supervisor: Ali Mehrabi
2nd reader: Yasamin Nazari

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

June 30, 2025

Contents

1 Abstract

We perform an experimental study on sorting algorithm within the evolving data model. In this model, the algorithms maintain a solution that is close to the true order at each time step. To this end, the algorithm is allowed to perform one comparison at each time step, where after each comparison a constant number α of random swaps of consecutive elements occur in the true order, so the true order changes with time. Previous work studies two variants of quicksort and two variants of insertion sort. We build upon this work and see whether the theoretical claims regarding the algorithms hold in practice. Additionally, we see whether the claims regarding algorithm convergence and behavior hold with our implementation. Furthermore, we study the effect of the block size on the maintained Kendall tau distance. Lastly, we introduce a new variant of insertion sort and study how it compares with regards to quicksort and insertion sort.

We find that all algorithms converge within $O(n^2)$, even when starting from a maximal Kendall tau distance, and that the maintained Kendall tau and the steady behavior coincide with previous practical findings. Additionally, we find that for $\alpha \in \{1, 2, 10\}$ the theoretical bounds clearly hold in practice for most algorithms, and that it is highly likely that the theoretical bound holds in practice for blocksort. Furthermore, we find that the block size does have an effect on the maintained Kendall tau distance and that this effect increases for higher values of the input size and the change rate. Moreover, we find that smaller block sizes perform better in general, but that it does not necessarily hold that smaller is better. There seems to be a trade-off between performance and robustness. Lastly, we find that our variant of insertion sort exhibits the expected behavior of both quicksort and insertion sort, and that it seems to be viable for cases where the change rate changes significantly throughout the run.

Keywords: Sorting, Evolving data, Quicksort, Insertion sort, Experimental

2 Introduction

In the classical paradigm, an algorithm is given an input, performs some computations, and returns an output within some time and space. We can reason about this time and space complexity in various ways, such as the best case, worst case and average case, which allows us to select the best algorithm for our use case. Though not all real world settings reflect this paradigm of input, computation, and output, which gives rise to new computational models such as the **evolving data** model introduced by Anagnostopoulos *et al.*[1].

2.1 General Framework

In the evolving data model, the given input changes while the algorithm performs the computations. At each time step, the algorithm accesses this changing input through **probes**, which is implemented in various ways depending on the problem setting. Furthermore, at each time step the algorithm gives some approximate output based on the information it received from probing the input and the goal of the algorithm. The output is approximately close to the "true" output, which is the output at a certain time given that the true input would have been given statically to the algorithm at that time. For example, for sorting algorithms, which we focus on in this paper, the goal is to return at each time step a sorted list that is approximately sorted according to the true ranking. Lastly, due to the nature of this model, an algorithm does not have a certain runtime, it runs indefinitely until it is terminated.

2.2 Related Work

This thesis builds on the previous theoretical and experimental work performed by Anagnostopoulos *et al.*[1] and by Besa Vial *et al.* [3][2] regarding sorting problems in the evolving data model.

The evolving data model was introduced by Anagnostopoulos *et al.*[1], who study sorting and selection with respect to a total ordered list that slowly changes. This means that the input does not have a static total order given to the algorithm but rather a dynamic total order that changes at each time step. This change is modeled by means of a small constant number of uniformly **random swaps** of pairwise adjacent elements, where we call the mechanism that performs the swaps the **uniform adversary**. The algorithm has access to the true total order through a constant number of **probes**, which is modeled by means of comparisons between pairwise adjacent elements in the true ordering. For this setting, they prove a lower bound of $\Omega(n)$ on the expected Kendall tau distance between the estimated order computed by any algorithm and the actual order at any time t. Furthermore, they prove that repeatedly running quicksort maintains a Kendall tau distance of $O(n \ln n)$, in expectation and with high probability, with respect to the true total order. Lastly, they prove that alternating steps of quicksort with that of blocksort, a variant of quicksort, can maintain a Kendall tau distance of at most $O(n \ln \ln n)$ in expectation and with high probability.

Besa Vial *et al.* [3][2] performed an experimental study to see how these algorithms perform in practice [3], and how they perform against sorting algorithms that have a quadratic runtime in the classical paradigm, specifically insertion sort, cocktail sort and

bubble sort. They found that the algorithms empirically reach a steady behavior within $O(n^2)$. Furthermore, they found that quadratic runtime algorithms maintain a lower Kendall tau distance in the steady behavior; however, they take longer to reach this state. Additionally, they introduce the hot spot adversary to see how the algorithms perform under such a setting. They provide evidence that these sorting algorithms have similar robustness against the **hot spot adversary** as they do against the uniform adversary.

Besa *et al.* [2] expanded on these results with a theoretical paper where they prove that repeatedly running insertion sort in this setting maintains a Kendall tau distance of at most $O(n)$ in expectation and with high probability. That is, after an initialization period of $\Theta(n^2)$ steps. Additionally, they show that running an initial run of quicksort improves this initialization time to $\Theta(n \ln n)$ steps, with high probability, by first performing a run of quicksort and then following that with repeated insertion sort.

There are several other related works, but due to space limitations, we refer an interested reader to the related work sections found in the mentioned works of Anagnostopoulos *et al.*[1] and of Besa Vial *et al.* [3][2] for more related works.

2.3 Our contribution

In this thesis, we conduct an experimental investigation of sorting in the evolving data model.

We start by verifying the claim that all algorithms converge within $O(n^2)$[3] by investigating if this holds when we start from a maximal Kendall tau distance and run the algorithms for n^2 time steps. Then we compare our findings regarding the steady behavior of quicksort, blocksort[1] and insertion sort [2] with the findings of Besa Vial *et al.* [3]. To see whether we still find similar results to those of Besa Vial *et al.* [3], even when starting from a maximal Kendall.

Afterwards, we verify whether the theoretical claims regarding the maintained Kendall tau distance hold in practice.

Then, we will experimentally investigate how the algorithm that alternates between steps of quicksort and of blocksort[1] performs for different block sizes under varying amounts for the input size and the change rate. This is done to see whether we can improve the practical results found by Besa Vial *et al.* [3] regarding blocksort.

Lastly, we investigate the new algorithm introduced in section 3.6 that expands on quicksort then repeated insertion sort [2], namely **repeated quicksort then insertion sort**. We investigate how it compares to the regular variant of insertion sort and the regular variant of quicksort for varying amounts for the input size and the change rate.

To this end, we formally introduce the setting, the relevant algorithms and their behavior in Section 3. In Section 4 we describe the experiments and present the results.

Plagiarism declaration. I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

3 Sorting Evolving Data

3.1 Formal Setting

We now formally introduce the setting for sorting on a true total ordered list in the evolving data model formulated by Anagnostopoulos *et al.*[1]. From this point onward, when we mention an algorithm, we specifically refer to sorting algorithms that are run in the evolving data model.

Input and Output

The initial input of an algorithm is a finite domain U of n distinct elements. At each time step t, there exists a true total order π_t on U, and this total order changes over time. At each time step π_t is accessible to the algorithm through **probes**, which are comparisons between pairwise adjacent elements.

The algorithm uses probes to maintain an **approximation** of the order, denoted $\tilde{\pi}_t$. At each t the algorithm gives as output the current estimated ordering $\tilde{\pi}_t$, where $\tilde{\pi}_t$ is a possible permutation over the n elements. The goal of the algorithm is to maintain an approximation $\tilde{\pi}_t$ that is close to π_t at each t. To measure this closeness between $\tilde{\pi}_t$ and π_t, we use the **Kendall tau** distance:

$$\mathrm{KT}(\pi_1, \pi_2) = |\{(x, y) \mid x <_{\pi_1} y \wedge y <_{\pi_2} x\}|$$

This is the number of pairwise inversions between $\tilde{\pi}_t$ and π_t, so in our case that is the number of pairs that are incorrectly ordered with respect to the true order. The maximal Kendall tau distance is reached when one list is reverse of the other, where we would have the Kendall tau distance equal to $n \cdot (n-1)/2$, so it is tightly bound from above by $\Theta(n^2)$.

Discrete Time Model

It is important to note that time is **discretized**, meaning we have steps $t = 0, 1, 2, \ldots$. At each time step, an algorithm is allowed to perform a fixed number of **probes** p. We will only consider the case where $p = 1$, and therefore take the number of comparisons made as an indication for the time.

After each set of comparisons, the true order π_t changes to π_{t+1} through a sequence of **swaps**, depending on the adversary model. We are interested in the initial setting introduced by Anagnostopoulos *et al.*[1], therefore, we will only consider the **uniform adversary** [3]. The uniform adversary performs a constant number of **random swaps** depending on the **change rate** $\alpha \geq 1$. Each time, it chooses a pair of pairwise adjacent elements in π_t uniformly at random and swaps them.

Metrics for Analyzing Algorithms

We do not have a definitive terminal state because an algorithm runs indefinitely in the evolving data model. Therefore, it is important to note the following two metrics to analyze the behavior of the algorithms running in the model: the **steady behavior** [3] and the **convergence time** [3]. For each algorithm the Kendall tau distance reaches a final repetitive behavior after a certain amount of time. This repetitive behavior is called the steady behavior, and the time required to reach it is called the convergence time. Every algorithm's empirical convergence time is at most $O(n^2)$ steps[3].

Additionally, note that we can use the running time of algorithms in the classical paradigm in order to reason about its behavior and properties in the evolving data model.

3.2 Randomized Quicksort

In the classical paradigm, quicksort takes as an input an array of n elements and outputs the array sorted increasingly. To do so, it divides the list into two parts depending on the pivot, which is a chosen element of the array. Everything to the left of the pivot is smaller and everything to the right is bigger. Afterwards, it recursively runs quicksort on the left and right parts until the whole array is sorted. The base case for quicksort is a call in which the number of elements being sorted is less than two because there is nothing to sort in such a case. The worst-case time complexity of quicksort is $O(n^2)$. This occurs when the partitioning algorithm picks the smallest or largest element as the pivot in each step.

Algorithm 1 Partition(A, p, r)

Input: Array A, indices p and r

```
1: x ← A[r]
2: i ← p − 1
3: for j ← p to r − 1
4:     if A[j] ≤ x then
5:         i ← i + 1
6:         Exchange A[i] with A[j]
7:     end if
8: end for
9: Exchange A[i + 1] with A[r]
10: return i + 1
```

Algorithm 2 Randomized_Partition(A, p, r)

Input: Array A, indices p and r

```
1: i ← RANDOM(p, r)
2: Exchange A[r] with A[i]
3: return PARTITION(A, p, r)
```

Algorithm 3 Randomized_Quicksort(A, p, r)

Input: Array A, indices p and r

```
1: if p < r then
2:     q ← RANDOMIZED_PARTITION(A, p, r)
3:     RANDOMIZED_QUICKSORT(A, p, q − 1)
4:     RANDOMIZED_QUICKSORT(A, q + 1, r)
5: end if
```

In the evolving data model, we run quicksort repeatedly. Furthermore, the result of the comparisons are done using probes, whereby the result is based on the current true order. We update our current approximation when a quicksort run on the whole array terminates. We start a new run after updating the approximation. Given our setting, it is important that we use randomized quicksort, which is a variant of quicksort whereby the pivot element is randomly picked and moved into the pivot position, which is at the end of the current part of the array on which we called partition on. Randomized quicksort has an average case of $O(n \ln n)$ and due to the random selection of the pivot we avoid the worst-case time complexity of $O(n^2)$ with high probability. Furthermore, given our setting, it is not permissible that the recursive run of the algorithm on the left part of the array is interleaved with the recursive run on the right part. Anagnostopoulos *et al.* [1] proof that repeatedly running quicksort guarantees that for every time step t the distance between the orderings π_t and $\tilde{\pi}_t$ is $O(n \ln n)$, with high probability.

3.3 Blocksort

We now introduce an algorithm that can maintain an error of $O(n \ln \ln n)$, with high probability [1]. This algorithm builds on the results obtained by quicksort and exploits its properties. Specifically, that after the quicksort run terminates, which due to its running time results in an error of $\Omega(n \ln n)$, the rank of each element in the approximation is within $O(\ln n)$ of its actual rank. Therefore, we do not need to run an $O(n \ln n)$ algorithm from scratch. Instead, we can run several ($O(n/\ln n)$) local quicksorts on **blocks** of size $m = \Theta(\ln n)$ to correct the ordering. We interleave steps of these local quicksorts with steps of a full quicksort in order to obtain the resulting algorithm that can maintain a distance of $O(n \ln \ln n)$. The blocksort algorithm is responsible for performing the local quicksorts.

Algorithm 4 BlockSort(ρ)

Input: Permutation ρ = output of full Quicksort at time t_0

1: B : array of size m
2: **for** $j = 1$ to $m/2$
3: $\quad B(j) \leftarrow \rho^{-1}(j)$
4: **end for**
5: **for** $i = 1$ to $2n/m - 1$
6: $\quad$ **for** $j = 1$ to $m/2$
7: $\quad\quad B(m/2 + j) \leftarrow \rho^{-1}(i \cdot m/2 + j)$
8: $\quad$ **end for**
9: $\quad$ QUICKSORT(B)
10: $\quad$ **for** $j = 1$ to $m/2$
11: $\quad\quad \sigma^{-1}((i-1) \cdot m/2 + j) \leftarrow B(j)$
12: $\quad\quad B(j) \leftarrow B(m/2 + j)$
13: $\quad$ **end for**
14: **end for**
15: **for** $j = 1$ to $m/2$
16: $\quad \sigma^{-1}(n - m/2 + j) \leftarrow B(j)$
17: **end for**

Blocksort takes as input the last full run of quicksort, so an approximation where each element is within $O(\ln n)$ of its actual rank. The algorithm divides the array into overlapping blocks of size m, which is $O(\ln n)$, and sorts them from left to right. After sorting a block, we store the smallest $m/2$ elements to the result and create a new block with the largest $m/2$ elements of the current block and the next $m/2$ elements of the array. Thus. the blocks overlap with $m/2$ elements, this is done to counteract the possible case where elements move towards neighboring blocks during execution.

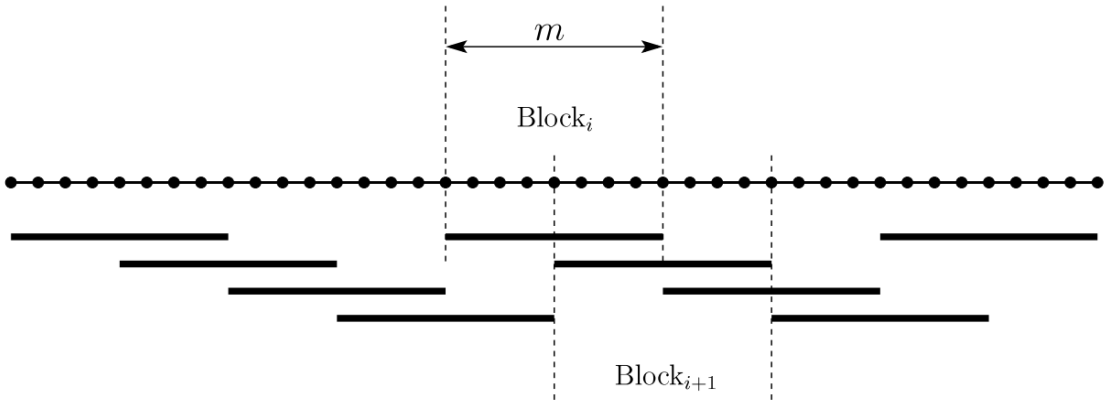


Figure 1: The overlap of blocks with their neighboring blocks.

Another possible issue is that elements eventually move far from their actual rank when we repeatedly run blocksort. To solve this, we interleave executions of blocksort with executions of regular quicksort. The two processes are executed independently with their own data structures. On even time steps we execute steps of blocksort and on odd steps those of quicksort. After each terminated run of blocksort we update our current approximation with the result obtained, and after that we start a new blocksort run with the last output of a full quicksort. Note that we start with an initial full quicksort run before being able to run blocksort.

3.4 Repeated Insertion Sort

In the classical paradigm insertion sort takes as an input an array of n elements and outputs the array sorted increasingly. To do so, it compares each element to the elements on its left and insert it in its correct position. We start this process from left to right and thus slowly sort the array in that order. The current element that is inserted is the element at position j. Inserting into the right position is done by means of swapping when the element to the left of it is bigger than the current element and by deducting j, until either there is nothing to the left of the element or the element to the left of it is smaller. The running time of insertion sort is $O(n + I)$, where I is the number of initial inversions in the list. Therefore, insertion sort runs $O(n)$ in its best case, which is an already sorted array, and $O(n^2)$ in its worst and average cases.

Algorithm 5 Repeated Insertion Sort

Input: Array A of n elements

```
while true
    for i ← 1 to n − 1
        j ← i
        while j > 0 and A[j] < A[j − 1]
            Swap A[j] and A[j − 1]
            j ← j − 1
        end while
    end for
end while
```

In the evolving data model we run insertion sort repeatedly. Furthermore, the result of the comparisons are done using probes, whereby the result is based on the current true order. The swaps are performed in the current approximation. So we update the current approximation directly, instead of waiting for a run of insertion sort to terminate. Note that we no longer have the property that everything to the left of an inserted element is smaller during sorting. This is because the true order changes each time step and therefore an already sorted part can become unsorted by some swaps in the true order.

Besa Vial *et al.* proof that repeatedly running insertion sort can for every time step t maintain a distance of $O(n)$ between the orderings π_t and $\tilde{\pi}_t$, with high probability. This is the asymptotically optimal distance as proven by Anagnostopoulos *et al.*[1]. Furthermore, this linear upper bound seems to hold for higher rates of $\alpha > 1$[2], where Besa Vial *et al.*[3] show that it holds for $\alpha \in \{1, 2, 10\}$.

3.5 Improved Repeated Insertion Sort

Besa Vial *et al.*[2] show that the convergence time of repeated insertion sort can be improved from $\Theta(n^2)$ to $\Theta(n \ln n)$ by starting with a run of quicksort. This is because quicksort takes $\Theta(n \ln n)$ comparisons to finish and because of this the number of inversions left after a run is $O(n \ln n)$.

Algorithm 6 Quicksort Followed by Repeated Insertion Sort

Input: Array A of n elements

```
Quicksort(A)
while true
    for i ← 1 to n − 1
        j ← i
        while j > 0 and A[j] < A[j − 1]
            Swap A[j] and A[j − 1]
            j ← j − 1
        end while
    end for
end while
```

3.6 Possible Further Improved Repeated Insertion Sort

We now introduce our attempt at an improvement. Besa Vial *et al.*[3] found that for higher change rates α quicksort performs better, this is most likely due to the insertion sort fixing at most one inversion each probe, whereby we are limited to one probe each time step, while α swaps occur, each of which can either introduce or fix an inversion. Therefore, it is more dependent on the swaps that occur instead of the inversion fixed by the insertion sort. The algorithm attempts to improve the performance of Algorithm 6 for higher values of α by running quicksort after a certain amount of runs of insertion sort, namely k runs. This could result in the average behavior of the two algorithms, where we can pick a higher k if we want the insertion sort behavior to dominate.

Algorithm 7 Repeated Quicksort Followed by Repeated Insertion Sort

Input: Array A of n elements and an integer k

```
1:  while true
2:      QUICKSORT(A)
3:      m ← 0
4:      while m < k
5:          for i ← 1 to n − 1
6:              j ← i
7:              while j > 0 and A[j] < A[j − 1]
8:                  Swap A[j] and A[j − 1]
9:                  j ← j − 1
10:             end while
11:         end for
12:         m ← m + 1
13:     end while
14: end while
```

We may have that the algorithm performs worse than repeated insertion sort for low values of α, while performing worse than repeated quicksort for high values of α. But it could still be a viable option if it performs better than repeated quicksort for low values of α and better than repeated insertion sort for high values of α. This would make it perform relatively well in systems where α fluctuates constantly and where we are unsure of the current α or where it costs resources to switch from one algorithm to the other. Our hypothesis is that higher values of k should maintain a lower Kendall tau distance compared to lower values of k for lower values of α, while the reverse should also hold.

3.7 Brief Overview Algorithmic Behavior

For clarity we summarize the different possible ways of updating, the effect it has on the steady behavior, and the theoretical bounds of each algorithm.

In terms of updating, the algorithms we consider can be divided into three categories: **periodic**, **instant** and **mixed**.

Periodic algorithms update the approximation when the current run is terminated, after which they start a new run. At each time step, the algorithm gives as output the most recently updated approximation. So, while the algorithms are computing the Kendall tau distance is determined by the random swaps performed by the uniform adversary, because the output is the same while the true order changes. In the steady behavior, this leads to a slow increase in the Kendall tau distance, until the current run terminates, and the Kendall tau distance drops. This results in an oscillating behavior.

Instant algorithms can update the approximation at each time step, depending on whether an inversion has been found with the probe. The maintained Kendall tau distance at each time step is determined by the inversions fixed by the algorithm and by the random swaps performed by the uniform adversary. In the steady behavior, this leads to a somewhat straight line around a certain Kendall tau distance, which is determined by the probability that inversions are fixed or introduced.

Mixed algorithms make use of both types of updating. This is because they alternate between two different algorithms where one updates periodically and the other updates instantly. Repeated Quicksort Followed by Repeated Insertion Sort falls under this category because it repeatedly runs one quicksort run followed by a certain number of insertion sort runs. In the steady behavior, this should result in phases where it displays the behavior of quicksort and phases where it displays the behavior of insertion sort.

Algorithm	**Update type**	**Theoretical bounds**
Repeated Quicksort	Periodic	$O(n \ln n)$
Repeated Blocksort	Periodic	$O(n \ln \ln n)$
Repeated Insertion sort	Instant	$O(n)$
Quicksort Followed by Repeated Insertion Sort	Mix	$O(n)$
Repeated Quicksort Followed by Repeated Insertion Sort	Mix	?

Table 1: Summary of algorithms, update types, and theoretical bounds.

4 Experiments

The main goal of our experimental framework is to address the following questions:

1. Do all the algorithms converge within $O(n^2)$ when starting from a maximal Kendall tau distance? If so, do they converge to Kendall tau distances similar to those found by Besa Vial *et al.*[3]?
2. Do the various theoretical claims regarding the maintained Kendall tau distance hold in practice?
3. What is the effect of the block size on the maintained Kendall tau distance in the steady behavior for varying input sizes and change rates?
4. How does **Repeated Quicksort Followed by Repeated Insertion Sort** perform in the steady behavior compared to repeated quicksort and repeated insertion sort for varying input sizes and change rates?

We present results below that address each of these questions.

4.1 Experimental Setup

We implemented the various algorithms in Python [1]. Randomness for experiments was generated using randint from the Python library random. The code was run in Python 3.10.12, and the computer running it was a remote machine with the following CPU: AMD EPYC-Milan (48) @ 2.399GHz.

We take the time measurement as the number of comparisons performed because the probe rate is equal to one in all experiments. For each configuration, the different algorithms ran on the same seeds. This is so that the changes in the true ordering were the same for each algorithm given a certain configuration.

To evaluate algorithm performance, we computed the average Kendall tau distance across multiple runs. Specifically, at each time step, we calculated the average distance over all runs with different seeds. This averaging process provides a clearer view of the general performance trends of each algorithm.

Due to time constraints we used 30 seeds for the experiment performed in 4.3, while using 100 seeds in all other experiments. We terminate a run after n^2 comparisons and during a run the Kendall tau distance is measured every $n/20$ comparisons for the experiment in Section 4.2 and $n/10$ for the rest of the experiments.

In Section 4.2 the choice of a sampling rate of $n/20$ and a running time of n^2 is the same as that of Besa et al. [3]. We chose to do so to compare our results with theirs. Note that while Besa *et al.* [3] evaluate their results on a single seed, we averaged 100 seeds to better capture the overall behavior.

[1] The code can be found on Github on the following link: https://github.com/YasinKaryagdi/Thesis

4.2 Validating convergence time and behavior

We begin by answering the first question, with the additional goal of comparing our findings with those of Besa Vial *et al.*[3]. To that end, we choose the same block size, that is, the block size is the first even number larger than $10 \ln n$ that divides n [3]. To validate the algorithmic behavior, we chose a reverse-sorted list as input, so that the Kendall tau distance is maximal at the start. We do so because this is the worst starting point in terms of the distance between the true order and the approximation. We want to see if the algorithms still reach the steady behavior within $O(n^2)$[3]. Note that this differs from the choice of Besa et al.[3]. They performed this experiment with a uniformly random shuffled list, resulting in a Kendall tau distance that is on average half the maximal Kendall tau distance (in our case of $n = 1000$, it is equal to 250000). Additionally, we want to see how Algorithm 7 behaves compared to the rest and if it will also reach the steady behavior within $O(n^2)$, therefore we also run it for this setup.

We start with a reverse-sorted list and run the algorithms for the case where $n = 1000$ and $\alpha = 1$ for n^2 time steps. Figure 2 shows the results of the whole run and Figure 3 shows the behavior at the endpoint, which is the last 5%. Lastly, Figure 4 shows the results obtained by Besa Vial *et al.*[3].

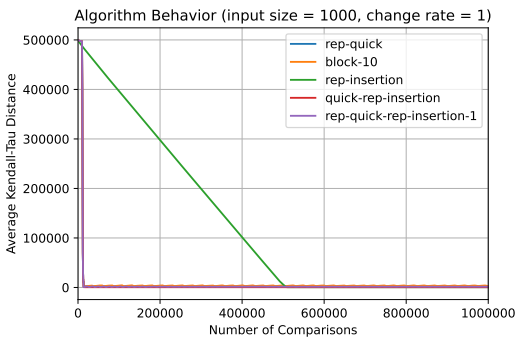


Figure 2: Algorithm behavior for n^2 time steps, for $n = 1000$ and $\alpha = 1$

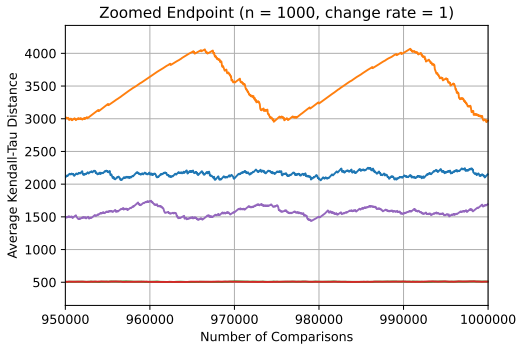


Figure 3: Algorithm behavior at the endpoint for $n = 1000$ and $\alpha = 1$

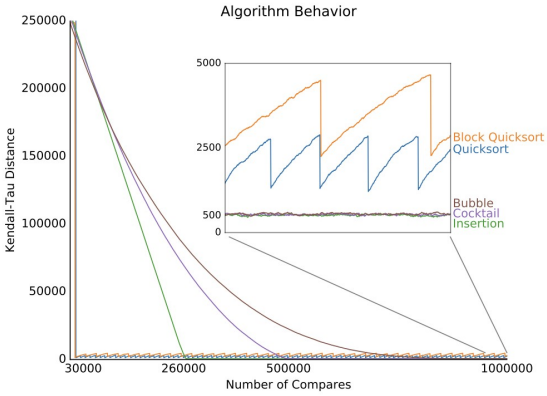


Figure 4: Algorithm behavior at the for $n = 1000$ and $\alpha = 1$ found by Besa Vial *et al.*[3] starting from a randomly shuffled list.

We find that all algorithms reach the steady state within $O(n^2)$, even when starting from the maximal Kendall tau distance. Additionally, we find that running quicksort before repeatedly running insertion sort improves the convergence time, while maintaining the same Kendall tau distance in the steady state. Furthermore, we find that the quicksort variants all oscillate, while insertion sort maintains a constant Kendall tau distance. Lastly, we find that Algorithm 7 falls between quicksort and insertion sort in terms of maintained Kendall tau distance, while displaying oscillating behavior similar to quicksort.

The steady behavior and the Kendall tau distance maintained in this state are similar to the results obtained by Besa Vial *et al.*[3]. Note that we differ in the amount of time that it takes insertion sort to reach the steady behavior; we find that it takes longer. This can be explained by the fact that we start with a different Kendall tau distance.

The second difference is in the behavior itself. Their plot shows the drop that occurs when a blocksort or quicksort run terminates, which looks like a straight line from a peak to a valley. We also see these peaks and valleys, but they are less distinct and the distance between a peak and a valley is smaller than what Besa Vial *et al.*[3] find. This is most likely because we plot the Kendall tau distance averaged over 100 runs, whereas they plot it for only one run. This explains the difference in behavior and in the exact Kendall tau distances maintained within the steady state.

From all these findings, we conclude that all algorithms converge within $O(n^2)$ when starting from a maximal Kendall tau distance, and that the Kendall tau distances found in this case are similar to those found by Besa Vial *et al.*[3], who started from a randomly shuffled list.

4.3 Validating Theoretical Bounds

We are now interested in validating the theoretical bounds mentioned in Section 3.7 by looking at whether they hold in practice. To this end, we run the same algorithms from the previous experiment for input sizes between 100 and 10000. However, note that we exclude Algorithm 7 because it does not have a theoretical bound. From the previous experiment, we find that all algorithms converge within $O(n^2)$ and therefore we only consider the Kendall tau distance maintained in the last 10% in each run, which we consider to be the endpoint. Furthermore, we only consider $\alpha \in \{1, 2, 10\}$ because Besa Vial *et al.* [3] have shown that the theoretical bounds hold for these values of the change rates. Lastly, we start with a reverse-sorted list in order to ensure that we start from the maximal Kendall tau distance.

In the following, we plot the average Kendall tau distance over the endpoint divided by the input size against the input size. We rescale the average Kendall tau distance because the theoretical bounds are all divisible by n, and dividing by n makes it easier to identify the bounds.

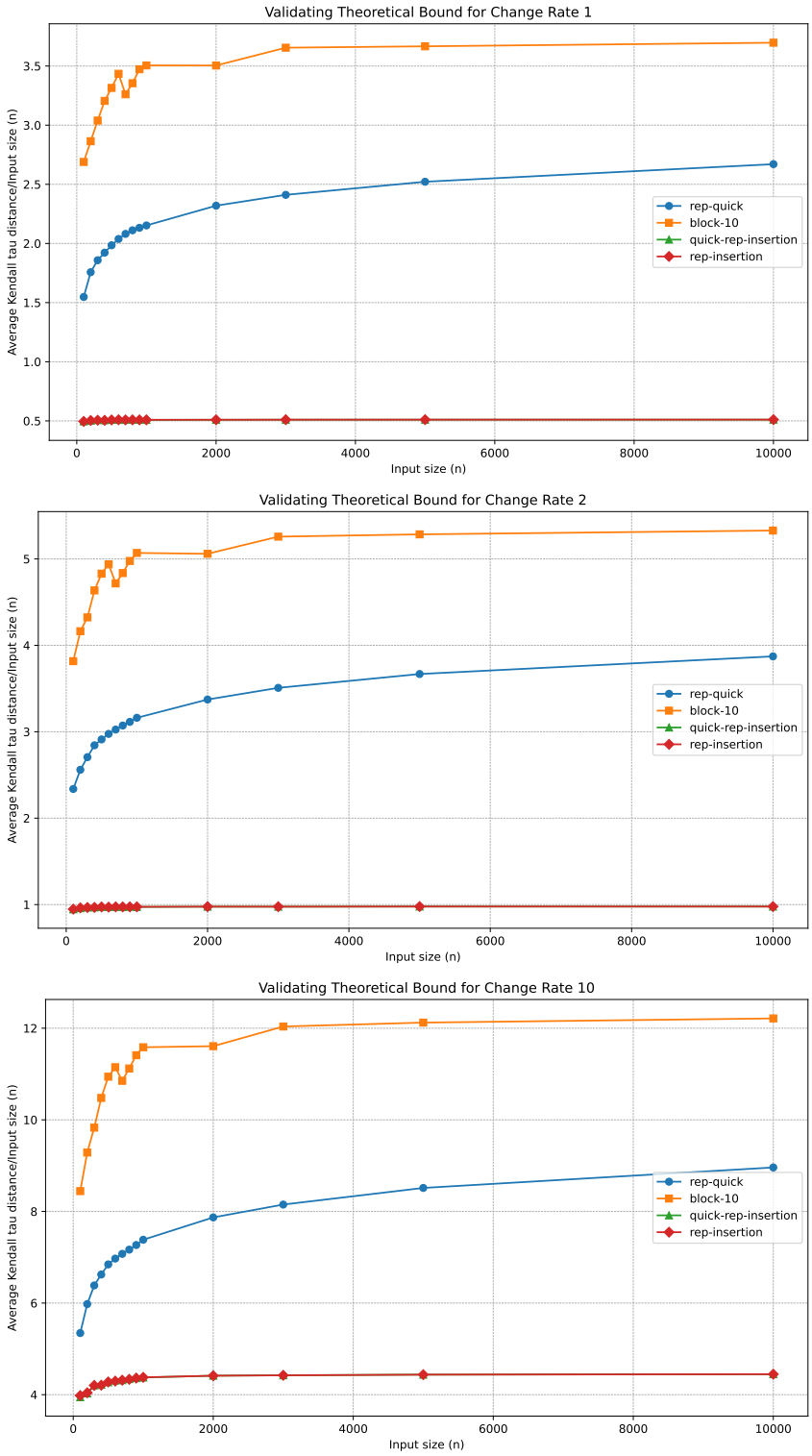


Figure 5: The attained average maintained Kendall tau distance over the endpoint divided by the input size for various input sizes and for increasing change rates.

We see that for all $\alpha \in \{1, 2, 10\}$ we have that both repeated insertion sort, as well as quicksort followed by repeated insertion sort, maintain a rescaled average Kendall tau distance of $O(1)$ in the endpoint. This means that they maintain an average Kendall tau distance of $O(n)$. So, we find that the theoretical bound holds in practice for these algorithms.

Additionally, we see that for all $\alpha \in \{1, 2, 10\}$ we have that repeated quicksort maintains a rescaled average Kendall tau distance of $O(\ln n)$ in the endpoint. This means that it maintains an average Kendall tau distance of $O(n \ln n)$, so we again find that the theoretical bound holds in practice.

We now look at blocksort. First we note that there seems to be an initial drop after $n = 600$, this is most likely due to the way we pick block sizes, which causes the block sizes to be inconsistent over the various input sizes. This leads to this case where the block size of $n = 600$ is 100, while that of $n = 700$, $n = 800$ and $n = 900$ are respectively 70, 80 and 90. This affects our ability to clearly distinguish whether the theoretical bound holds in practice. However, despite this we see that blocksort appears to be a logarithmic algorithm for all $\alpha \in \{1, 2, 10\}$. It is unclear from these results whether it is double logarithmic. This is because double logarithmic algorithm are supposed to have a slower increase in the average maintained Kendall tau distance compared to quicksort, which is a logarithmic algorithm. However, we find that the increase is faster compared to a logarithmic algorithm until an input size of $n = 3000$, after which it seems to have a slower increase. So, from $n = 3000$ onward it appears to be double logarithmic, meaning $O(\ln \ln n)$, which would imply that the theoretical bound of $O(n \ln \ln n)$ holds in practice, but this does not seem to hold for the input sizes smaller than $n = 3000$.

There could be multiple reasons for this; the most likely candidate is the way we pick the block sizes. Moreover, we could also have that blocksort requires the input to be at least of a certain size before it maintains a Kendall tau distance of $O(n \ln \ln n)$. Another reason could be the way we average, where we take the average at the endpoint of 10%. We could have that taking the endpoint to be 10% for these input sizes does not accurately depict the average. Lastly, we could have that running for 30 seeds is not sufficient to accurately depict the average behavior. In any case, from these results we can not conclude for certain whether the theoretical bound of $O(n \ln \ln n)$ holds in practice, but it seems highly likely because we see that from $n = 3000$ onward the increase is lower than that of quicksort, which we identified to be logarithmic.

From these results, we conclude that for $\alpha \in \{1, 2, 10\}$ the theoretical bounds clearly hold in practice for most algorithms, and that it is highly likely that the theoretical bound holds in practice for blocksort.

4.4 Effect of Block Size on Performance

As mentioned previously, Anagnostopoulos *et al.*[1] do not specify a specific constant for the block size and only mention that it should be of size $O(\ln n)$. We are interested in what effect the block size has on the performance of the algorithm in terms of the Kendall tau distance measured in the steady behavior.

To investigate this, we introduce the block constant b, where the block size depends on the block constant. We chose the block size for blocksort to be the first even number larger than $b \cdot ln(n)$ that divides n, where b is the block constant specified in the experiment.

We run the algorithm mentioned in section 3.3 for $b \in \{1, 5, 10, 20, 40\}$ and investigate its behavior at the endpoint. Each will be run for various input sizes, for $n \in \{100, 500, 1000\}$, and for various change rates, for $\alpha \in \{1, 5, 10, 20\}$. Note that it is possible that the block constants may resolve into the same block size or into block sizes close to each other due to our choice for the block size. Furthermore, note that we are only interested in the performance of the algorithms in their steady behavior and relative to each other, and less in their convergence time, so we start with a sorted array as the initial input and only consider the endpoints. The endpoints for the various values of n are as follows: the last 40% for $n = 100$, the last 10% for $n = 500$ and the last 5% for $n = 1000$.

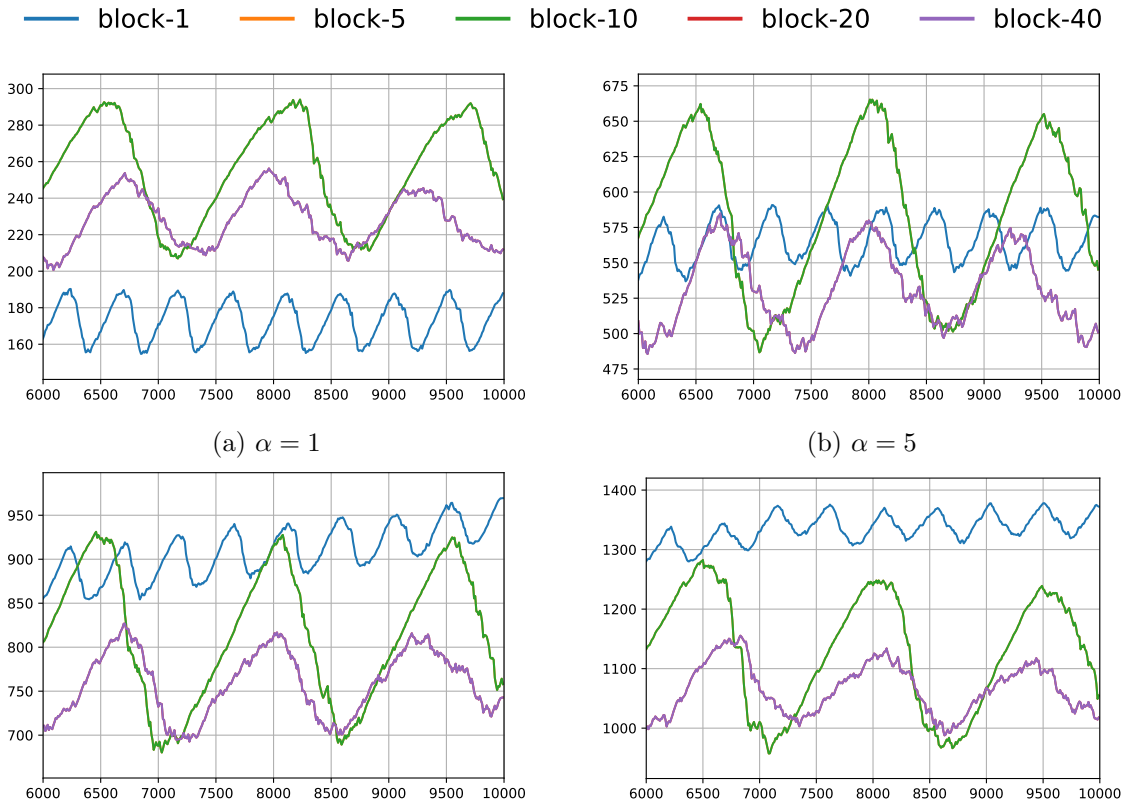
In the following, we present the results of the experiment using figures to show the average Kendall tau distance at each time step, and tables to show the average Kendall tau distance over the endpoint.

Note that not all lines seem to be plotted; this is due to some block constants that result in the same exact block size and thus the same behavior as others. This happens for $b = 5$ and $b = 10$ given that $n = 100$, and for $b = 20$ and $b = 40$ in the case where $n = 100$ and $n = 500$.

	100	**500**	**1000**
block-1	4	10	8
block-5	50	50	40
block-10	50	100	100
block-20	100	250	200
block-40	100	250	500

Table 2: The exact block size given a block constant for the various input sizes n.

Results for $n = 100$



(a) $\alpha = 1$

(b) $\alpha = 5$

(c) $\alpha = 10$

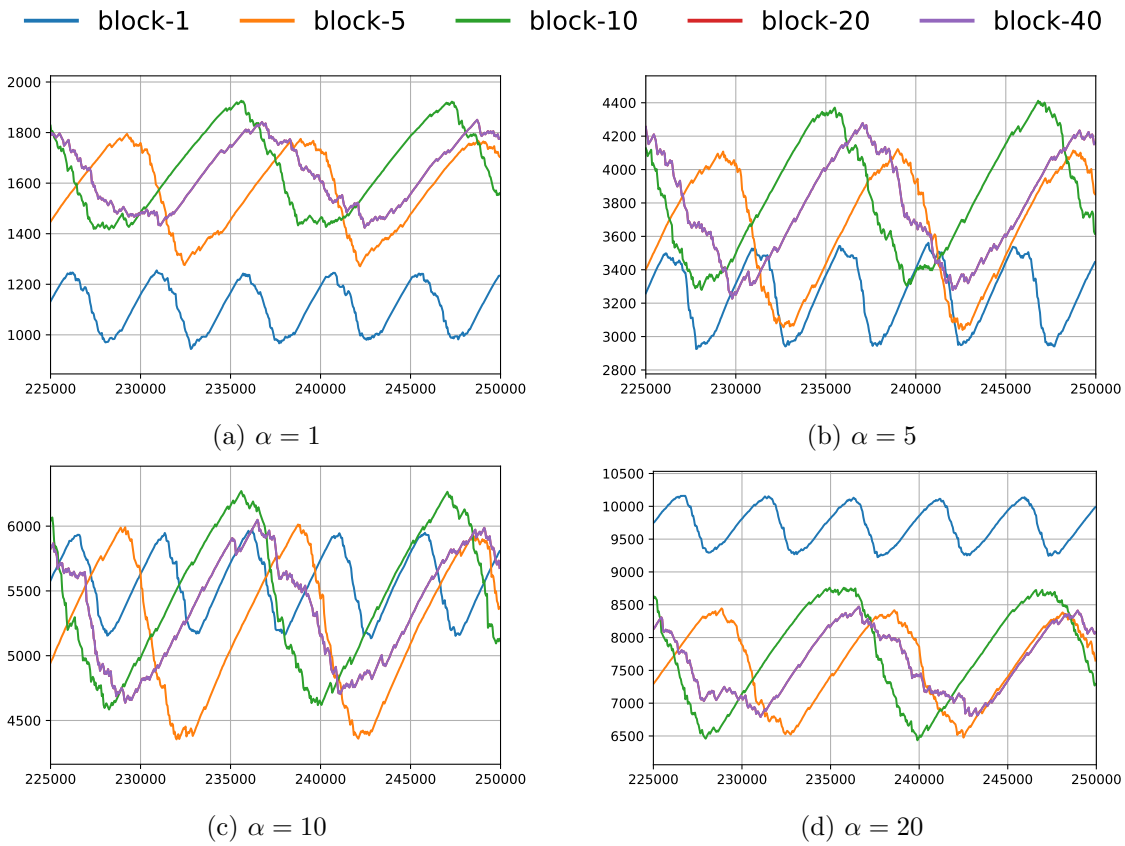
(d) $\alpha = 20$

Figure 6: Comparison between blocksort performance across input sizes and alpha values for $n = 100$.

	1	**5**	**10**	**20**
block-1	172.79	565.78	909.19	1335.69
block-5	256.42	584.14	816.05	1130.99
block-10	256.42	584.14	816.05	1130.99
block-20	227.99	531.45	758.70	1066.19
block-40	227.99	531.45	758.70	1066.19

Table 3: The average Kendall tau distance for $n = 100$ over the endpoint. Color coding: 1st, 2nd, 3rd.

Results for $n = 500$



(a) $\alpha = 1$

(b) $\alpha = 5$

(c) $\alpha = 10$

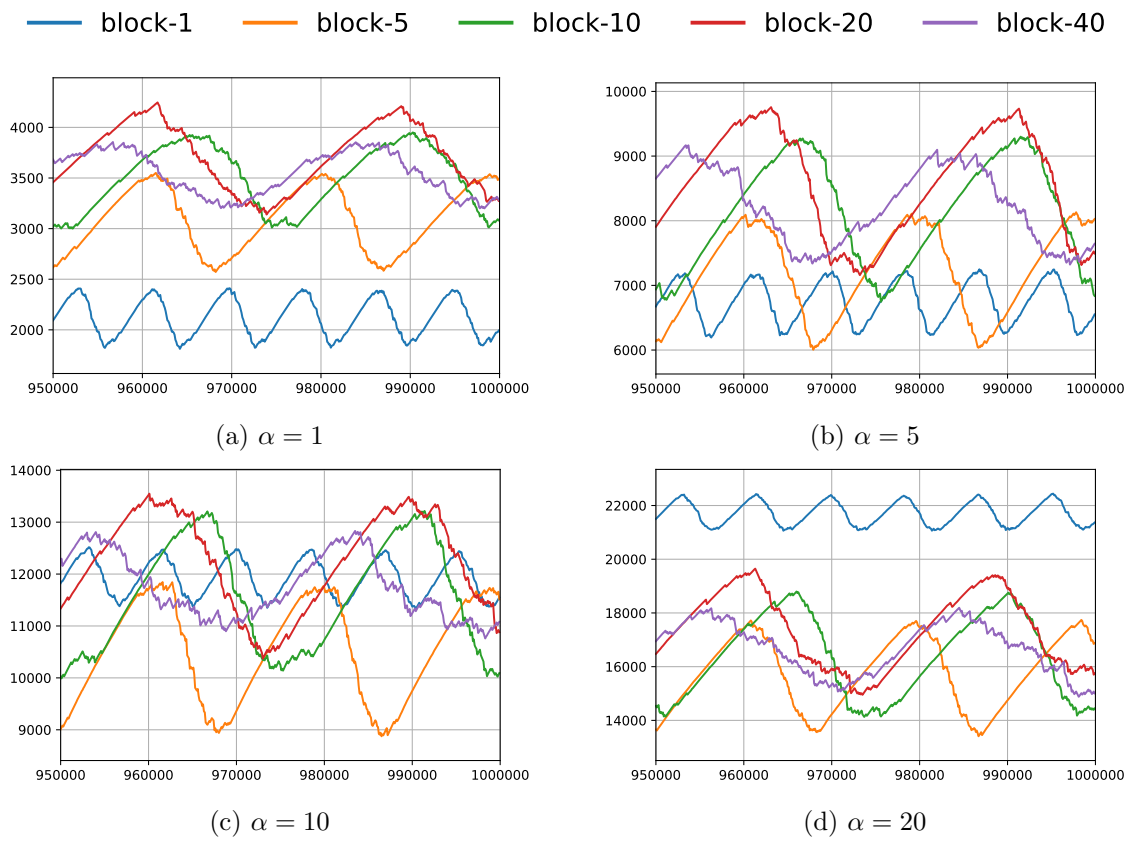
(d) $\alpha = 20$

Figure 7: Comparison between blocksort performance across input sizes and alpha values for $n = 500$.

	1	**5**	**10**	**20**
block-1	1111.52	3254.90	5563.13	9708.67
block-5	1573.06	3659.18	5253.59	7599.51
block-10	1656.07	3841.70	5442.17	7748.60
block-20	1633.09	3773.40	5372.05	7614.21
block-40	1633.09	3773.40	5372.05	7614.21

Table 4: The average Kendall tau distance for $n = 500$ over the endpoint. Color coding: 1st, 2nd, 3rd, 4th.

Results for $n = 1000$



(a) $\alpha = 1$

(b) $\alpha = 5$

(c) $\alpha = 10$

(d) $\alpha = 20$

Figure 8: Comparison between blocksort performance across input sizes and alpha values for $n = 1000$.

	1	5	10	20
block-1	2123.13	6751.11	11926.26	**21708.86**
block-5	3095.18	7215.01	10531.16	15780.96
block-10	3499.70	8118.51	11550.61	16360.95
block-20	**3720.99**	**8601.65**	**12252.52**	17377.69
block-40	3547.91	8241.86	11747.07	16693.90

Table 5: The average Kendall tau distance for $n = 1000$ over the endpoint. Color coding by rank: 1st, 2nd, 3rd, 4th, **5th**. The differences between the different ranking at $\alpha = 10$ is very small, this is the turning point for block-1. Block-20 consistently performs worse than the rest.

From Tables 3, 4 and 5 we see that the smallest block constant of 1 results in a better average Kendall tau distance maintained within the steady behavior for lower values of α, namely $\alpha = 1$ and $\alpha = 10$. Although it performs worse for the higher change rates, where it performs considerably worse than the other block constants. This trend is consistent over all the input sizes, and we have that the difference in performance between it and the others is more significant for higher input sizes. For example, in the case of $\alpha = 1$ and $n = 100$ it ranks first and there is a difference of 50 with the second ranking, while in the case of $\alpha = 1$ and $n = 1000$ this difference is 900, which is a significant improvement. This increase in effect also applies when it ranks last in the cases of $\alpha = 10$ and $\alpha = 20$, where the differences become greater for larger input sizes. We can clearly see this difference in performance in the figures, where we see that it initially oscillates between Kendall tau distances below the other lines, but slowly starts to rise for higher α, until it clearly oscillates above the other lines.

Moreover, interestingly, we find that in these cases of $\alpha = 10$ and $\alpha = 20$ the first ranking block constant is that of 5, our second smallest block constant, which ranked second in the other cases. From this we conclude that, in general, smaller block constants perform better given these low change rates, but that this increase in performance is less robust if the block constant is too small. This is most likely due to the elements shifting further from their true rank for higher values of α. The purpose of blocksort is to counteract the $O(\ln n)$ shift in ranking mentioned in Section 3.3. We could have that certain block sizes are better at counteracting this shift in ranking caused by the higher change rates. A small block size is most likely advantageous for a low change rate because the elements are less likely to shift too far, while it becomes a major disadvantage for higher change rates because the elements most likely constantly move to neighboring blocks, and thus are not properly moved back to their true rank.

Surprisingly, we find that a lower block constant does not always lead to better performance. This can be seen in Table 3, where the two largest block constants rank first or second, and Table 4, where they rank second or third. Furthermore, this is also the case in Table 5 where we can see that a block constant of 40 performs better than that of 20. From this we can see that a lower block size does not necessarily correspond to an improvement, though it is unclear currently why this is the case.

Lastly, we note that the differences in the average Kendall tau distance maintained are quite close in most cases. The only exception to this is a block constant of 1 in the case where $\alpha = 1$, where it performs significantly better than the rest, and $\alpha = 20$, where it performs significantly worse than the rest. Interestingly, from the figures we see that the different block constants exhibit similar but different behaviors, even though their averages are close to each other. The larger block constants oscillate less compared to the smaller ones, and they also have different maxima and minima.

From these results, we conclude that the block size has an effect on the Kendall tau distance maintained and that this possible improvement increases for higher values of n. Although there does seem to be a trade-off between performance and robustness, where a lower block constant performs better for lower change rates but is not robust for higher change rates. Lastly, a smaller block size does not always perform better compared to a bigger block size.

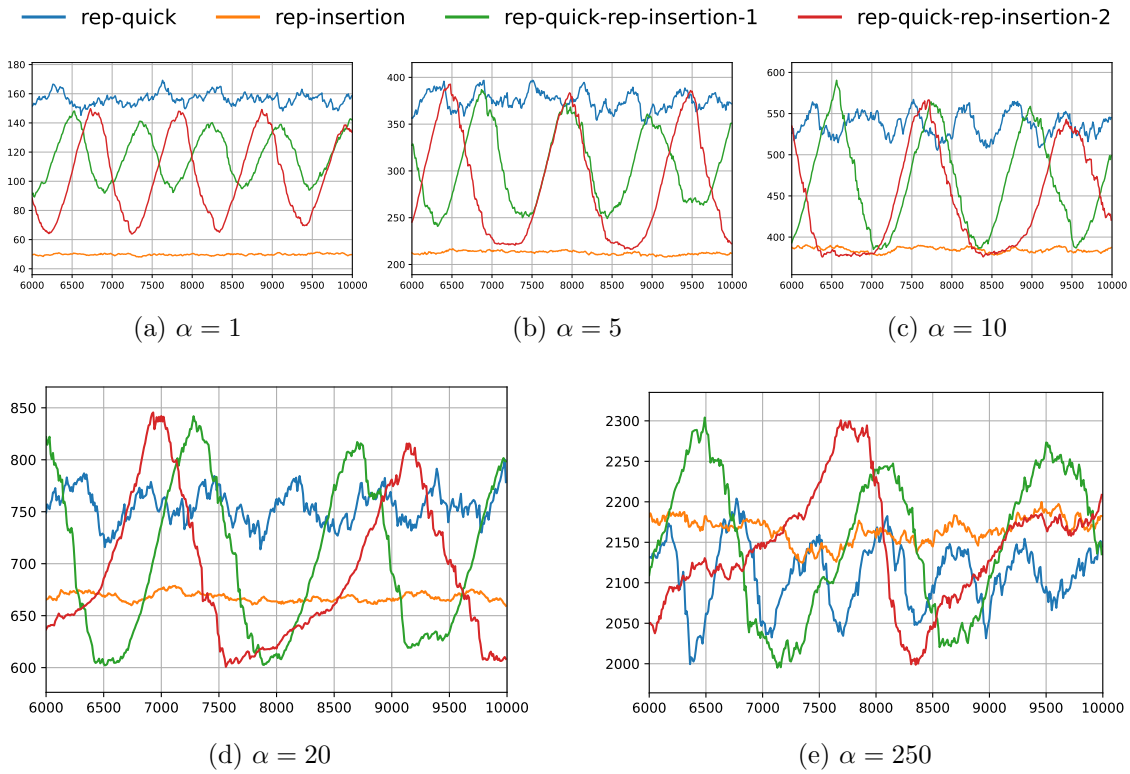
4.5 Performance of Repeated Quicksort Followed by Repeated Insertion Sort

Recall the potential improvement mentioned in Section 3.6. We are interested in how it compares to repeated quicksort and to repeated insertion sort in the steady behavior for differing values of the input size and the change rate. Furthermore, we are interested in how different amounts for k affect this attained behavior. Therefore, we consider running Algorithm 7 for $k \in \{1, 2\}$, where k is the number of insertion sort runs that we perform after a quicksort run. Once more, we are only interested in the performance of the algorithms in their steady behavior and relative to each other, so we start with a sorted array as the initial input and only consider the endpoints. The endpoints for the various values of n are as follows: the last 40% for $n = 100$, the last 10% for $n = 500$ and the last 5% for $n = 1000$. The only exception is that we consider the last 40% in the cases where $\alpha = 250$ to properly display the steady behavior. Furthermore, note that we only consider Algorithm 5, as opposed to both Algorithm 5 and Algorithm 6. This is because both maintain the same Kendall tau distance in the steady behavior, so either is a valid choice.

We run each algorithm for various input size n, namely $n \in \{100, 500, 1000\}$, and reasonable change rates, namely $\alpha \in \{1, 5, 10, 20\}$, plus an extreme case, namely $\alpha = 250$.

In the following, we present the results of the experiment using figures to show the average Kendall tau distance at each time step, and tables to show the average Kendall tau distance over the endpoint.

Results for $n = 100$



(a) $\alpha = 1$ (b) $\alpha = 5$ (c) $\alpha = 10$

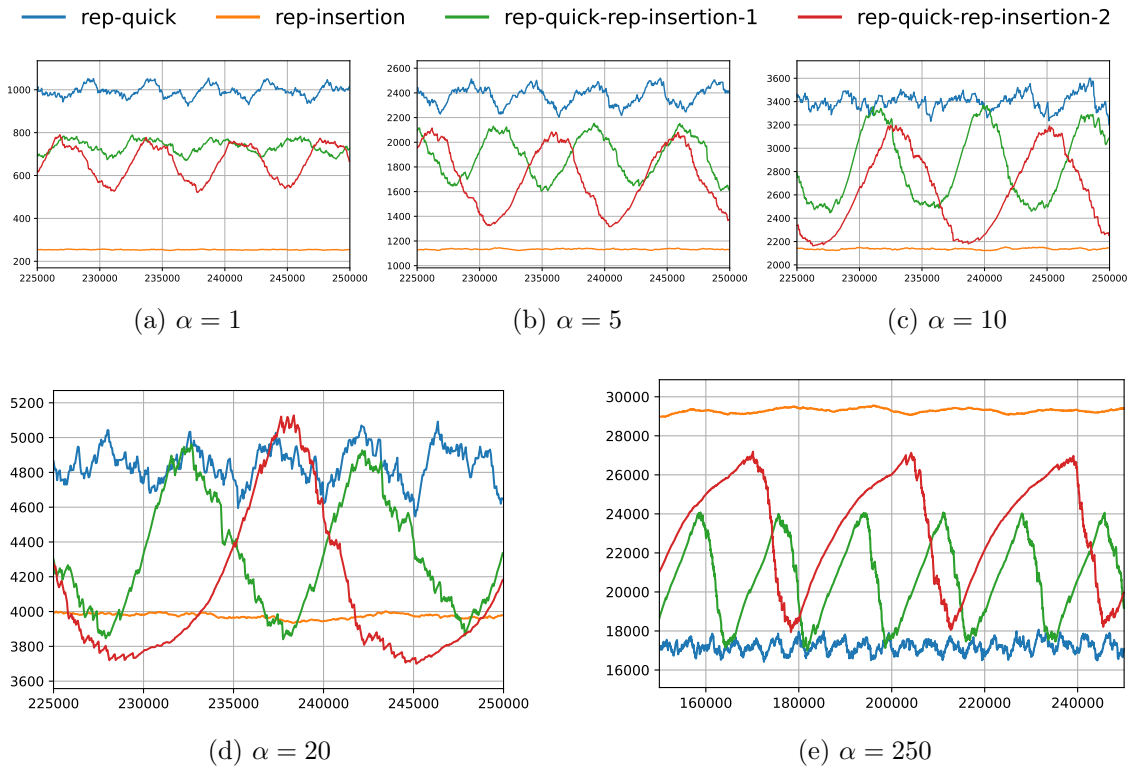
(d) $\alpha = 20$ (e) $\alpha = 250$

Figure 9: Comparison between blocksort performance across input sizes and α values for $n = 100$.

	1	**5**	**10**	**20**	**250**
rep-quick	156.44	374.99	536.17	755.55	2109.81
rep-insertion	49.80	212.52	384.73	667.87	2165.24
rep-quick-rep-insertion-1	117.17	303.53	467.05	702.63	2151.45
rep-quick-rep-insertion-2	103.68	287.97	442.00	693.11	2144.20

Table 6: The average Kendall tau distance for $n = 100$ over the endpoint. Color-coded per column: 1st, 2nd, 3rd, 4th.

Results for $n = 500$



(a) $\alpha = 1$ (b) $\alpha = 5$ (c) $\alpha = 10$

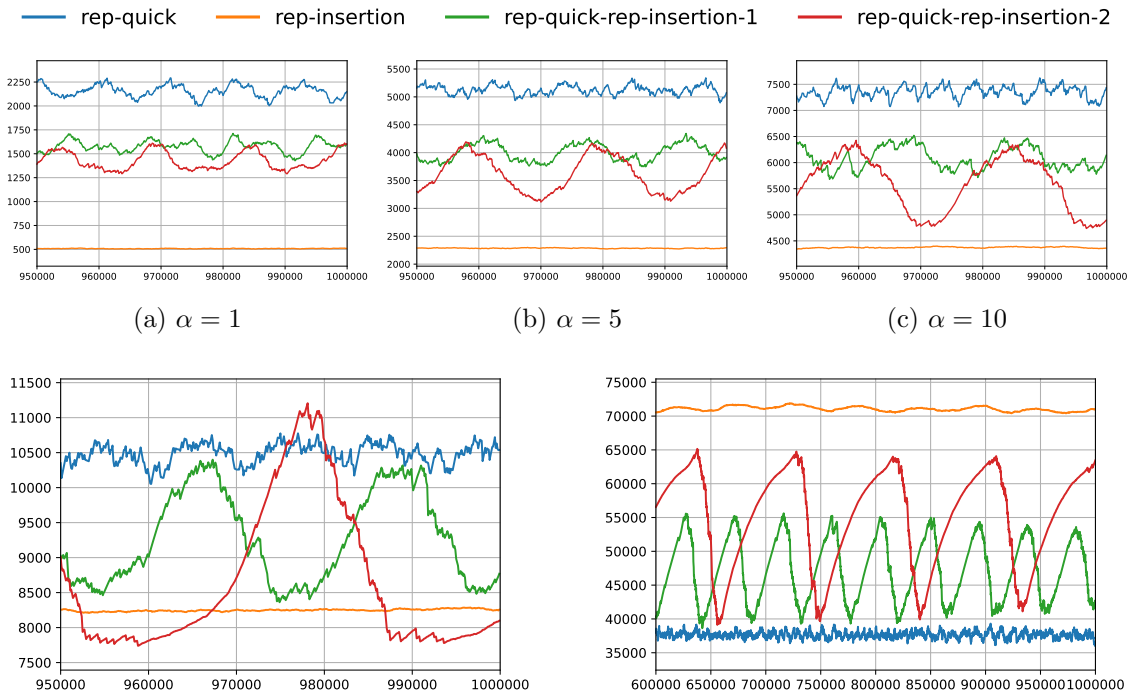
(d) $\alpha = 20$ (e) $\alpha = 250$

Figure 10: Comparison between blocksort performance across input sizes and α values for $n = 500$.

	1	**5**	**10**	**20**	**250**
rep-quick	993.54	2375.23	3400.54	4846.21	17202.95
rep-insertion	254.00	1133.52	2136.98	3974.20	29285.92
rep-quick-rep-insertion-1	734.25	1881.40	2850.26	4320.68	20781.91
rep-quick-rep-insertion-2	665.83	1717.16	2631.10	4101.72	23389.68

Table 7: The average Kendall tau distance for $n = 500$ over the endpoint. Color-coded per column: 1st, 2nd, 3rd, 4th.

Results for $n = 1000$



(a) $\alpha = 1$ (b) $\alpha = 5$ (c) $\alpha = 10$

(d) $\alpha = 20$ (e) $\alpha = 250$

Figure 11: Comparison between blocksort performance across input sizes and α values for $n = 1000$.

	1	**5**	**10**	**20**	**250**
rep-quick	2156.34	5126.42	7353.53	10486.82	37658.42
rep-insertion	508.41	2285.63	4374.12	8247.88	71064.09
rep-quick-rep-insertion-1	1576.71	4001.89	6082.36	9262.29	47305.99
rep-quick-rep-insertion-2	1427.56	3621.57	5596.82	8631.01	55274.90

Table 8: The average Kendall tau distance for $n = 1000$ over the endpoint. Color-coded per column: 1st, 2nd, 3rd, 4th.

We start by looking at the performance of repeated quicksort and of insertion sort, and reason about how those affect Algorithm 7. Tables 6, 7, and 8 show that for all $\alpha \in \{1, 5, 10, 20, 250\}$ there is a gap between the maintained Kendall tau distance of repeated quicksort and that of repeated insertion sort, and that this gap is larger for higher values of n. We find that for all $\alpha \in \{1, 5, 10, 20\}$ repeated insertion sort performs best, while repeated quicksort performs worst. The reverse holds for $\alpha = 250$. Additionally, we see that the gap between the performance of these two algorithms is slowly narrowed, and we find that there is most likely a certain point between $20 < \alpha < 250$ where repeated quicksort performs better than repeated insertion sort, and most likely continues doing so from that point onward.

We now look at how this affects the ranking and the averages of Algorithm 7, namely for $k \in \{1, 2\}$. We see that they consistently rank third and fourth. We have that running with $k = 2$ outperforms $k = 1$ in all the cases where repeated insertion sort ranks first, while $k = 1$ outperforms $k = 2$ in the case that repeated insertion sort ranks last. This is to be expected; as mentioned in Section 3.7 we slowly converge towards a certain Kendall tau distance when we repeatedly run insertion sort. Although in the case of Algorithm 7 we most likely do not reach that Kendall tau distance because we switch over to quicksort at a certain point, which in most cases seems to be too soon for insertion sort to have converged. However, $k = 2$ spends more time running insertion sort and therefore gets closer to that Kendall tau distance. This is a benefit when this Kendall tau distance is lower than that maintained by quicksort, while it is a disadvantage when it is higher, which is the case when $\alpha = 250$. This explains how $k = 1$ and $k = 2$ obtain their rankings and why they switch at $\alpha = 250$.

From Figures 9, 10 and 11 we clearly see that for $\alpha \in \{1, 5, 10\}$ both $k = 1$ and $k = 2$ slowly reach the Kendall tau distance maintained by repeated insertion sort, while we see that $k = 2$ gets closer in all cases. From Figure 9c, we can clearly see what happens when they do reach this Kendall tau distance. Here we find that both $k = 1$ and $k = 2$ reach the maintained Kendall tau distance of insertion sort, namely ≈ 400, and that $k = 2$ maintains this longer than $k = 1$.

Furthermore, from Figures 9d, 10d and 11d we see that both $k = 1$ and $k = 2$ maintain a Kendall tau distance below that of insertion sort at certain time periods, but do not do so at all times and therefore do not reach a lower average maintained Kendall tau distance over the endpoint.

Lastly, we see that, in terms of the average Kendall tau distance maintained over the endpoint $k = 1$ and $k = 2$ seem to fall right in between repeated insertion sort and repeated quicksort, so around the mean of these two algorithms. Additionally, we see that, if we exclude $\alpha = 250$, this holds more closely for higher α, most likely due to the narrowing of the gap between the performance of repeated insertion sort and repeated quicksort.

From these results, we conclude the following; Algorithm 7 can be viable in cases where the α is unknown or changes continuously. This is because it performs better than repeated quicksort for low α, while performing better than repeated insertion sort for high α. Additionally, performing more insertion sort runs, so higher values for k, is beneficial in the cases when repeated insertion sort is the best performing algorithm, while it is disadvantageous when it is the worst performing algorithm. So, depending on how quickly α changes or on how high α is, we can select an ideal k to maintain a lower average Kendall tau distance over the endpoint.

5 Conclusions

In this thesis, we performed an experimental study regarding sorting in the evolving data model. To this end, we introduced the setting and algorithms before we perform the experiments.

Our initial experiment shows that all algorithms converge within $O(n^2)$ when starting from a maximal Kendall tau distance, and that the Kendall tau distances found in this case are similar to those found by Besa Vial *et al.*[3], who started from a randomly shuffled list.

In our second experiment we validate the theoretical bounds by seeing if they hold in practice. We find that for $\alpha \in \{1, 2, 10\}$ the theoretical bounds clearly hold in practice for most algorithms, and that it is highly likely that the theoretical bound holds in practice for blocksort.

In our third experiment, we explore the effect of the block size on the maintained Kendall tau distance. Our results show that the block size does have an effect on the maintained Kendall tau distance and that this effect increases for higher values of n. We find that smaller block constants seem to perform better in general, but that if the resulting block constant becomes too small to account for the changes it performs considerably worse. So, there seems to be a trade-off between performance and robustness, where a lower block constant performs better for lower change rates but is not robust for higher change rates. Lastly, we find that smaller block sizes do not necessarily result in a lower maintained Kendall tau distance in the steady behavior.

In our last experiment, we looked at the performance and behavior of our new introduced algorithm, Repeated Quicksort Followed by Repeated Insertion Sort, compared to repeated quicksort and repeated insertion sort. Here we find that our new variant exhibits the expected behavior of both quicksort and insertion sort, and falls between the two in terms of maintained Kendall tau distance in the steady behavior. Furthermore, we find that this algorithm can be viable in cases where the α is unknown or changes continuously. This is because it performs better than repeated quicksort for low α, while performing better than repeated insertion sort for high α. Additionally, performing more insertion sort runs, so higher values for k, is beneficial in the cases when repeated insertion sort is the best performing algorithm, while it is disadvantageous when it is the worst performing algorithm. So, depending on how quickly α changes or on how high α is, we can select an ideal k to maintain a lower average Kendall tau distance over the endpoint.

Given our results, it would be interesting to explore the exact nature of this effect of the block size on the maintained Kendall tau distance. It would be interesting to answer the question on why exactly this occurs and what an ideal block size would be given the input size and the change rate. Additionally, it would be interesting to see how the current block constants perform for larger input sizes to see whether they perform better, worse, or the same as they did.

Additionally, it would be interesting to further explore our variant of insertion sort. Our results seem to imply that after a certain value for α repeated quicksort performs better than repeated insertion sort, which is the turning point for our algorithm. It would be interesting to try the same experiment for higher values of α, where $20 < \alpha < 250$, to see when this turning point is and if it is reasonable to expect it to occur. Lastly, it would be interesting to see how the algorithm performs in practice for higher rates of k.

References

[1] Aris Anagnostopoulos et al. "Sorting and selection on dynamic data". In: *Theoretical Computer Science* 412.24 (2011). Selected Papers from 36th International Colloquium on Automata, Languages and Programming (ICALP 2009), pp. 2564–2576. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2010.10.003. URL: https://www.sciencedirect.com/science/article/pii/S0304397510005475.

[2] Juan José Besa Vial et al. "Optimally Sorting Evolving Data". In: *CoRR* abs/1805.03350 (2018). arXiv: 1805.03350. URL: http://arxiv.org/abs/1805.03350.

[3] Juan José Besa Vial et al. "Quadratic Time Algorithms Appear to be Optimal for Sorting Evolving Data". In: *CoRR* abs/1805.05443 (2018). arXiv: 1805.05443. URL: http://arxiv.org/abs/1805.05443.

A Model Implementation

We start with the implementation of our DynamicList class and the relevant Stats class. We give an instance of the DynamicList class to an sorting algorithm at the start of a run. The true order and our approximation are stored in the DynamicList. Furthermore, probing within the sorting algorithm is done through the instance with the *probe_with_swap* method. Within this method we implement the random swaps that occur in the true ranking by means of the *random_swap* method. Additionally, within this method we store the current Kendall tau distance every so often depending on the time, which is the amount of probes performed, and the sample rate. To store the current distance we call the *add_curr_distance* with the real and approx lists as arguments.

```
from model.statistics import Stats
import random


class DynamicList:
    real: list[int]
    curr_approx: list[int]
    change_rate: int
    stats: Stats
    start_time: int

    # Is currently unused but it seems to be too late to remove it without
    # potentially breaking something, can be used in the future but would most likely require
    # you to change the way the probe method works.
    probe_rate: int

    def __init__(
        self,
        rand_seed: int,
        probe_rate: int,
        change_rate: int,
        n: int,
        sample_rate: int,
        time_limit: int,
        start_time: int
    ):
        random.seed(rand_seed)
        self.stats = Stats()
        self.probe_rate = probe_rate
        self.change_rate = change_rate
        self.sample_rate = sample_rate
        self.time_limit = time_limit
        self.start_time = start_time

        # We start with a real that goes from 0, to n
        self.real = []
        for i in range(0, n):
            self.real.append(i)

        # We start with an approximation that goes from 0, to n
        self.curr_approx = []
        for i in range(0, n):
```

```
            self.curr_approx.append(i)

    # Performs change_rate amount of uniformly random swaps before probe,
    # sorts based on the index of int i and j in the real list, so based on
     their positions in the real
    def probe_with_swap(self, i: int, j: int):
        self.stats.add_probe(i, j)

        if (self.get_time() % self.sample_rate == 0) and (self.get_time()
    >= self.start_time) and (self.get_time() <= self.time_limit):
            self.stats.add_curr_distance(self.real, self.curr_approx)

        self.random_swap(self.change_rate)

        index_i = self.real.index(i)
        index_j = self.real.index(j)
        return index_i < index_j

    # Used to perform random swaps in the real
    def random_swap(self, change_rate):
        n = len(self.real)

        # Perform change rate amount of random swaps in the real, swapping
    i with i + 1
        for x in range(0, change_rate):
            i = random.randint(0, n - 2)
            temp = self.real[i]
            self.real[i] = self.real[i + 1]
            self.real[i + 1] = temp

    # Used to update the current approximation
    def permute_answer(self, ans_perm: list[int]):
        self.curr_approx = ans_perm.copy()

    # Used to get the size of the dynamic list
    def size(self):
        return len(self.real)

    # Used to get the current number of probes performed,
    # we only consider the case where the probe rate = 1,
    # so this is the same as taking the len of the list keeping track of
    all the probes
    def get_time(self):
        return len(self.stats.probes)

    # Used to start with a real list that is reverse sorted,
    # so it is sorted decreasingly
    def reverse_order(self):
        n = self.size()
        for i in range(0, n):
            self.real[i] = (n - 1) - i
```

In the Stats class we store the probes and the calculated Kendall tau distances. Note that we calculate the distance between two lists with the *merge_sort* method, which we call from within the *add_curr_distance* method. The *merge_sort* method has access to the real list by means of the *probe*, which does not apply any changes in the true order, which allows merge sort to make as many probes as it requires to count all the inversions.

The probes are stored in the probes list, while the distances get stored in the distances list. When we call *get_time* in the DynamicList we return the length of the probes list because we only consider the case where the probe rate is equal to one. After a full execution, we store the distances list as an excel sheet as the result of that specific run.

```
# Used for tracking the time by means of storing the probes,
# and for tracking the Kendall tau distances at certain time steps,
# which depends on the sampling rate
class Stats:
    probes: list[list[int]]
    distances: list[int]

    def __init__(self):
        self.distances = []
        self.probes = []

    # Adds the current probe and the probes indexes to the list
    def add_probe(self, i, j):
        self.probes.append([i, j])

    # Allows comparison based on the index of int i and j in the real list,
    # so based on their positions in the real list,
    # no changes are made to the real list
    def probe(self, real: dict, i: int, j: int):
        index_i = real.get(i)
        index_j = real.get(j)
        return index_i < index_j

    # Merge sort that counts the number of inversions in the list,
    # which is the number of discordant pairs in the current approximation,
    # to this end we give access to the real list as a dictionary,
    # where the keys are the integers and the values are their indices in
    the real list
    def merge_sort(self, real: dict, temp: list[int]):
        if len(temp) <= 1:
            return 0

        # Splitting the list into the left and right halves
        # and recursively sorting them
        left_size = len(temp) // 2
        right_size = len(temp) - left_size


        left = []
        for i in range(0, left_size):
            left.append(temp[i])

        right = []
        for i in range(left_size, len(temp)):
            right.append(temp[i])
```

```
45
46          left_invs = self.merge_sort(real, left)
47          right_invs = self.merge_sort(real, right)
48          between_invs = 0
49          i = 0
50          j = 0
51          k = 0
52          while i < left_size and j < right_size:
53
54              if self.probe(real, left[i], right[j]):
55                  temp[k] = left[i]
56                  k += 1
57                  i += 1
58              else:
59                  temp[k] = right[j]
60
61                  # Something on the right being smaller means that there are
     a certain number
62                  # of inversions, namely (left_size - i), where i is the
    index of the current element on the left,
63                  # so we essentially have that this right side element
    agrees with i elements but is discordant with the rest of the left
    array
64                  between_invs += (left_size - i)
65                  k += 1
66                  j += 1
67
68          # Copy over the remaining elements in either left or right,
69          # the while loop will fail for at least one
70          while i < left_size:
71              temp[k] = left[i]
72              k += 1
73              i += 1
74
75          while j < right_size:
76              temp[k] = right[j]
77              k += 1
78              j += 1
79
80          return left_invs + right_invs + between_invs
81
82
83      # Adds the current distance between the real order and our
    approximation
84      def add_curr_distance(self, real: list[int], approx: list[int]):
85          temp = approx.copy()
86          real_dict = dict(zip(real, range(0, len(real))))
87
88          distance = self.merge_sort(real_dict, temp)
89          self.distances.append(distance)
```

B Algorithms Implementation

We implemented each algorithm according to the pseudocode, where we implemented the comparisons using *probe_with_swap*. Note how insertion sort makes changes directly to the approximation list, while the other sorting algorithms only update the answer after "termination", so they exchange the whole current approximation for a new one, while insertion sort gradually updates it. Lastly, we want to mention that we looked at the implementation of blocksort of Besa Vial *et al.*[3] and implemented it in Python, in order to have an accurate comparison of algorithm behavior. For our stack implementation of blocksort we use the QSortState in order to fully store a sort state during blocksort.

```
from model.dynamic_list import DynamicList
from model.q_sort_state import QSortState
import random
import math


def partition(list: DynamicList, to_sort, low: int, high: int):
    ranges = high - low + 1

    # We randomly pick a pivot, probably should have done random.randint(low, high)
    pivotChoice = low + random.randint(0, 32767) % ranges

    temp = to_sort[high]
    to_sort[high] = to_sort[pivotChoice]
    to_sort[pivotChoice] = temp

    i = low - 1
    for j in range(low, high):
        # Extra check that makes sure that we don't go over time
        if list.get_time() >= list.time_limit:
            return -1

        if list.probe_with_swap(to_sort[j], to_sort[high]):
            i += 1
            temp = to_sort[i]
            to_sort[i] = to_sort[j]
            to_sort[j] = temp
    temp = to_sort[i + 1]
    to_sort[i + 1] = to_sort[high]
    to_sort[high] = temp
    return i + 1


def quicksort(list: DynamicList, to_sort, low: int, high: int):
    if low < high:
        pivot = partition(list, to_sort, low, high)

        # Pivot of -1 means that list.get_time() >= list.time_limit so we return,
        # introduced in order to stop running the algorithm after the time limit has passed
        if pivot == -1:
            return
```

```
42
43          quicksort(list, to_sort, low, pivot - 1)
44          quicksort(list, to_sort, pivot + 1, high)
45
46
47  # Given a random list that goes from 0 to n-1 sort it based on the order in
        the real list,
48  # afterwards update the curr_approx to the result,
49  # this is the initial quicksort called when a run of quicksort is performed
50  def quicksort_base(list: DynamicList, n: int):
51      to_sort = []
52      for i in range(0, n):
53          to_sort.append(i)
54
55      quicksort(list, to_sort, 0, n - 1)
56      list.permute_answer(to_sort)
57
58
59  def repeated_insertion_sort(list: DynamicList, time_limit: int):
60      n = list.size()
61      while list.get_time() < time_limit:
62          for i in range(1, n):
63              j = i
64              # Probe the ints at approx[j] with approx[j - 1]
65              while j > 0 and list.probe_with_swap(
66                  list.curr_approx[j], list.curr_approx[j - 1]
67              ):
68                  temp = list.curr_approx[j - 1]
69                  list.curr_approx[j - 1] = list.curr_approx[j]
70                  list.curr_approx[j] = temp
71                  j -= 1
72
73                  # Extra check that makes sure that we don't go over time
74                  if list.get_time() >= time_limit:
75                      return
76
77
78  # Perform one run of quicksort, followed by repeated insertion sort
79  def quick_then_insertion_sort(list: DynamicList, time_limit: int):
80      n = list.size()
81      quicksort_base(list, n)
82      repeated_insertion_sort(list, time_limit)
83
84
85  # Perform repeated runs of one quicksort, followed by a certain amount of
        insertion sort runs
86  def rep_quick_then_insertion_sort(list: DynamicList, time_limit: int,
        iterations: int):
87      n = list.size()
88
89      while list.get_time() < time_limit:
90          quicksort_base(list, n)
91
92          k = 0
93          while (list.get_time() < time_limit) and (k < iterations):
94              for i in range(1, n):
```

```
                j = i
                # Probe the ints at approx[j] with approx[j - 1]
                while j > 0 and list.probe_with_swap(
                    list.curr_approx[j], list.curr_approx[j - 1]
                ):
                    temp = list.curr_approx[j - 1]
                    list.curr_approx[j - 1] = list.curr_approx[j]
                    list.curr_approx[j] = temp
                    j -= 1

                    # Extra check that makes sure that we don't go over time
                    if list.get_time() >= time_limit:
                        return
            k += 1


# Repeatedly perform quicksort until the time limit is reached
def repeated_quicksort(list: DynamicList, time_limit: int):
    n = list.size()
    while list.get_time() < time_limit:
        quicksort_base(list, n)


def start_new_quicksort_call(
    to_sort: list[int], call_stack: list[QSortState], low: int, high: int
):
    if low < high:
        range = high - low + 1
        pivotChoice = low + random.randint(0, 32767) % range
        temp = to_sort[high]
        to_sort[high] = to_sort[pivotChoice]
        to_sort[pivotChoice] = temp
        newCall = QSortState(low, high)
        call_stack.append(newCall)


def stack_quicksort_run_step(
    list: DynamicList, to_sort: list[int], call_stack: list[QSortState]
):
    step_performed = False

    while not step_performed:

        # If there are no quicksort calls on the stack, terminate
        if len(call_stack) == 0:
            return False

        # Otherwise attempt to run a step of the top call
        curr_call = call_stack[-1]
        low = curr_call.low
        high = curr_call.high
        i = curr_call.i
        j = curr_call.j

        if j < high:
```

```
            if list.probe_with_swap(to_sort[j], to_sort[high]):
                curr_call.i += 1
                temp = to_sort[i + 1]
                to_sort[i + 1] = to_sort[j]
                to_sort[j] = temp
            curr_call.j += 1
            step_performed = True
        else:
            call_stack.pop()
            # If the quicksort call finished,
            # then move the pivot into the correct position
            temp = to_sort[i + 1]
            to_sort[i + 1] = to_sort[high]
            to_sort[high] = temp

            # And start the two new recursive calls
            low_left = low
            high_left = i
            low_right = i + 2
            high_right = high
            start_new_quicksort_call(to_sort, call_stack, low_left, high_left)
            start_new_quicksort_call(to_sort, call_stack, low_right, high_right)
    return True


def blocked_quicksort(list: DynamicList, time_limit: int, block_constant: int):
    n = list.size()

    # Determine block size m close to block_constant * ln(n)
    m = int(block_constant * math.log(n))
    if m % 2 == 1:
        m += 1
    if m > n:
        m = n
    while n % m != 0:
        m += 2

    # Initial full quicksort
    full_stack = []
    quicksort_base(list, n)
    to_full_sort = list.curr_approx.copy()

    last_full_answer = list.curr_approx.copy()
    next_full_answer = list.curr_approx.copy()
    new_full_answer = False

    start_new_quicksort_call(to_full_sort, full_stack, 0, n - 1)
    block_stack = []

    while list.get_time() < time_limit:
        to_blocksort = [None] * m
        block_answer = [None] * n

```

```
        # Switch to new full answer if available
        if new_full_answer:
            new_full_answer = False
            last_full_answer = next_full_answer.copy()

        # Fill first m/2 items
        for i in range(m // 2):
            temp = last_full_answer[i]
            to_blocksort[i] = temp

        for i in range(1, (2 * n) // m):
            # Fill next m/2 items
            for j in range(m // 2):
                temp = last_full_answer[i * (m // 2) + j]
                to_blocksort[m // 2 + j] = temp

            # Sort current block
            start_new_quicksort_call(to_blocksort, block_stack, 0, m - 1)
            while stack_quicksort_run_step(list, to_blocksort, block_stack)
    :
                full_sort_done = not stack_quicksort_run_step(
                    list, to_full_sort, full_stack
                )
                if full_sort_done:
                    new_full_answer = True
                    next_full_answer = to_full_sort.copy()
                    start_new_quicksort_call(to_full_sort, full_stack, 0, n
     - 1)

            # Copy smallest half to block answer, shift larger half forward
            for j in range(m // 2):
                temp1 = to_blocksort[j]
                block_answer[(i - 1) * (m // 2) + j] = temp1
                temp2 = to_blocksort[m // 2 + j]
                to_blocksort[j] = temp2

        # Handle last m/2 elements
        for j in range(m // 2):
            temp = to_blocksort[j]
            block_answer[n - (m // 2) + j] = temp

        list.permute_answer(block_answer)

        # Update all working answers
        last_full_answer = block_answer.copy()
        next_full_answer = block_answer.copy()
        to_full_sort = block_answer.copy()
```

```
# Used in order keep track of the sort states during Block Sort
class QSortState:
    low: int
    high: int
    i: int
    j: int

    def __init__(self, l, h):
        self.low = l
```

```
        self.high = h
        self.i = self.low - 1
        self.j = self.low
```

C Running Experiments

We use the Runner class to run an algorithm in a specific configuration for different seeds. We created instances of the Runner class within a nested for loop, where the first for loop went over the input size, the second over the algorithm, the third over the change rate and the last over the seeds. This allowed us to construct a whole experiment by specifying the lists and running the nested for loop. Most important is to note the possible starting ordering for the real, which are "reverse-sorted" and "sorted". It is also important to initialize the Runner instance with a valid algorithm name, which can be found in the *run* method.

```
from model.dynamic_list import DynamicList
from algorithms.algorithms import *
import pandas as pd


class Runner:
    alg: str
    curr_list: DynamicList
    time_limit: int

    def __init__(
        self,
        rand_seed: int,
        probe_rate: int,
        change_rate: int,
        algorithm: str,
        input_size: int,
        time_limit: int,
        sample_rate: int,
        config: str,
        start_time: int = 0
    ):
        self.alg = algorithm
        self.time_limit = time_limit
        self.curr_list = DynamicList(
            rand_seed, probe_rate, change_rate, input_size, sample_rate,
    time_limit, start_time
        )

        # Currently only valid options are sorted and reverse-sorted
        if config == "reverse-sorted":
            self.curr_list.reverse_order()
        elif config != "sorted":
            raise Exception("Invalid config")

    # Runs the relevant algorithm with the current configuration
    def run(self):
        if self.alg == "rep-quick":
            repeated_quicksort(self.curr_list, self.time_limit)
```

```
        elif self.alg.startswith("block"):
            temp = self.alg.split("-")
            block_constant = temp[len(temp) - 1]
            blocked_quicksort(self.curr_list, self.time_limit, int(
    block_constant))
        elif self.alg == "rep-insertion":
            repeated_insertion_sort(self.curr_list, self.time_limit)
        elif self.alg == "quick-rep-insertion":
            quick_then_insertion_sort(self.curr_list, self.time_limit)
        elif self.alg.startswith("rep-quick-rep-insertion"):
            temp = self.alg.split("-")
            i = temp[len(temp) - 1]
            rep_quick_then_insertion_sort(self.curr_list, self.time_limit,
    int(i))
        else:
            raise Exception("No correct alg given as input")

    # Saves the results of the current run in the for of an excel sheet
    def store_results(self, file_name):
        results = pd.DataFrame(self.curr_list.stats.distances)
        datatoexcel = pd.ExcelWriter(path=file_name)
        results.to_excel(datatoexcel)
        datatoexcel.close()
```