Opleiding Informatica

Automatically Deriving Sorting Algorithms in tUPL

Alano Kling

Supervisors:
Dr. Kristian F.D. Rietveld & Prof. Dr. Harry A.G. Wijshoff

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/09/2022

Abstract

Sorting is a fundamental part of computing. Many different concepts in computing require either direct or indirect use of sorting algorithms. As sorting is such a broad concept, many different approaches exist on how to perform this task. The performance achieved by the different approaches also depends on the hardware architecture and size of the data set. Sorting algorithms to be used are typically selected or implemented by the programmer, which allows for non-optimal solutions to be implemented that cannot be amended by traditional compilers. We will explore sorting in tUPL, a high level program specification language, which disallows explicit execution order and dependencies to be specified. We define different transformations that can be applied to a base specification of sorting in tUPL, that when combined result in the generation of different sorting algorithms. We found that there are algorithms which perform better than the rest, but also that existing algorithms can surpass these algorithms. The gathered results can be used as the basis for future work, where the task will be to combine different algorithms to be able to surpass these existing algorithms.

Contents

1 Introduction **1**

1.1 Thesis overview 1

2 Background **2**

2.1 tUPL 2

2.2 Sorting 2

2.3 Related Work 4

3 Methods **5**

3.1 Transformations 5

3.1.1 Limiting Reservoirs 5

3.1.2 Ordering 6

3.1.3 Problem Reduction 7

3.1.4 Divide and Conquer 8

3.1.5 Interval Sorting 9

3.2 Hybrid Algorithms 10

4 Evaluation **12**

4.1 Experiment Setup 12

4.2 Results 13

4.2.1 Ascending Values 13

4.2.2 Descending Values 15

4.2.3 Random Values 19

4.3 Comparing with existing algorithms 21

5 Discussion and Conclusion **22**

References **24**

A Result Data **25**

A.1 Ascending 25

A.2 Descending 26

A.3 Unique Random 29

A.4 Non-Unique Random 32
A.5 Hybrid Algorithms 35

1 Introduction

Sorting is one of the most fundamental concepts in computing. When a programmer wants to sort a set of numbers, they are faced with many options in terms of both languages and algorithms. As far as the language goes, this heavily depends on how long the sorting operation would take. As the input gets larger, a low level language would be more beneficial, as these could process the data faster than a high level languages [PF16]. One of the drawbacks with low level languages is that the programmer is given the freedom to choose the data structure and the control flow for the algorithm. This implies that the programmer has to take optimisations and dependencies into account when developing said algorithm. Furthermore, by explicitly encoding these dependencies and fixed execution orders in the low-level program, traditional compilers are limited in their ability to further optimize the code.

tUPL is a high-level program specification language designed to disallow programmers from specifying dependencies, execution orders and data structures explicitly [RW14b, vdZ19]. The main premises through which this is achieved are the looping structures introduced and the removal of explicitly defined data structures. By doing so, the compiler is given many more opportunities to transform and optimise the program and to suit it to a particular target architecture as it is no longer restricted by (false) dependencies specified by the programmer.

There exist a vast amount of different sorting algorithms, which all have different performance with different inputs. As sorting is used in many different scenarios in computing, it is crucial to generate the algorithm which performs the best for the use case. In some cases, when a single algorithm might not be sufficient, hybrid algorithms could be generated in order to handle more diverse input sets. When using tUPL, this can be performed by the compiler instead. This allows the programmer to only write the base algorithm and therefore increase the speed at which the code is written.

1.1 Thesis overview

For this thesis, we will be exploring the possibilities of transforming the tUPL base sorting algorithm in order to find a set of transformations which performs near-optimal in most scenarios. As tUPL heavily relies on the compiler, these transformations should be implementable as modifications to the loop structure or the data structure. We will solely be focusing on the loop structure in this thesis, although the modifications to data structure can also heavily influences the resulting algorithm speed [Sed98]. We will be exploring different transformations, each affecting the flow of the visitation of tuples.

In this thesis, we will go over all steps required to test and compare different sorting transformations and combinations in tUPL. Section 2 will be focused on discussing the background of the thesis subject and previous works related to tUPL; Section 3 will consist of setting up all preliminaries for the evaluation; Section 4 will discuss both the setup of the tests and the gathered findings; Section 5 will conclude the thesis and provide a final verdict. This thesis was written as part of the Computer Science bachelor's degree at the Leiden Institute of Advanced Computer Science (LIACS) and supervised by Dr. Kristian F.D. Rietveld and Prof. Dr. Harry A.G. Wijshoff.

2 Background

2.1 tUPL

We will look into the premise of tUPL and why this programming specification language gives the compiler many more optimisation opportunities. We will first look at the introduced looping structures, the forelem and whilelem loops. These loops iterate through a tuple reservoir, which contains a set of n-dimensional tuples. The forelem loop visits each tuple in the tuple reservoir exactly once in an undefined order. The whilelem loop continues to visit tuples in undefined order, visiting each tuple any number of times, until no more operations are executed inside the loop body. This allows the whilelem loop to visit tuples more than once, but also to not execute certain tuples for an infinite amount of iterations. Both loop structures are inherently parallel. Each iteration of both loops executes the loop body atomically, which nullifies the risk of data dependencies.

Instead of defining data structures in the code, tUPL handles the generation of optimal data structures during compile time. As data structures are not explicitly defined, lists such as arrays must be stored in some abstract way. This is done using a shared space. A shared space is a storage location which is indexed by any-dimensional integers. To access this shared space, we use an address function. An address function F_A of shared space A takes a tuple as input and returns an index inside the shared space. This mapping is simplified to $A[\texttt{t}]$, where $A[\texttt{t}]= n$ implies that n is stored in A at $[F_{\texttt{A}}(\texttt{t})]$. This does not imply that the shared space is defined as an array, as the structure of the shared space is defined during compile time. The order at which the shared space is stored into memory is also not explicitly defined, which provides more optimisations in terms of locality.

2.2 Sorting

As we will be discussing sorting in this thesis, we will look at the base sorting algorithm for tUPL in Listing 2.1. Sorting in tUPL is based on the swapping of elements. After an arbitrary number of swaps, any given list can be sorted. The tUPL algorithm achieves this by swapping elements based on tuples provided in an undefined order. When the shared space is eventually sorted, no more actions are performed inside the loop body as no more tuples refer to an unsorted pair of positions, which terminates the whilelem loop.

The first line denotes the tuple reservoir. This reservoir contains all tuples where $i < j$ and where both values are constrained between $[0, |\texttt{X}|)$. If we take $|\texttt{X}| = 4$ for example, we would get a tuple reservoir consisting of:

$\texttt{T} = \{ <0,1>, <0,2>, <0,3>, <1,2>, <1,3>, <2,3> \}$

The whilelem loop defines a single tuple `t` of the tuple reservoir `T`. Due to the whilelem loop, the order in which the tuples are iterated is undefined. The loop body attempts to swap the elements at positions `t.i` and `t.j` in `X` if these are not in the correct order. Once no elements are out of order, no tuple can be visited that has any influence on the shared space `X`, which can be used as definition to terminate the whilelem loop. As the whilelem loop body is executed atomically, it is possible to parallelise the loop, provided no overlapping tuples are executed. This implies that with the prior example of $|\mathtt{X}| = 4$, tuples $< 0, 1 >$ and $< 2, 3 >$, $< 0, 2 >$ and $< 1, 3 >$ or $< 0, 3 >$ and $< 1, 2 >$ could be iterated at the same time.

```
T = { < i,j > | 0 ≤ i < j < |X| }

whilelem ( t; t ∈ T ) {
    if (X[t.i] > X[t.j]) {
        swap(X[t.i], X[t.j]);
    }
}
```

Listing 2.1: Base sorting algorithm in tUPL

At the start of the algorithm, the cardinality of the inversion set, also known as the inversion number [Man85], will be any value $\mathtt{inv}(\mathtt{X}) \in \{x \mid x \in \mathbb{N}_0 \text{ and } 0 \leq x \leq \sum_{n=1}^{|\mathtt{X}|-1} n\}$. The task of the algorithm is to reduce this value to $\mathtt{inv}(\mathtt{X}) = 0$. We first declare the set of tuples which can be executed:

$\mathtt{T_a} = \{< \mathtt{i}, \mathtt{j} > \mid < \mathtt{i}, \mathtt{j} > \in \mathtt{T} \text{ and } \mathtt{X[t.i]} > \mathtt{X[t.j]}\}$

We also declare the inversion set $\mathtt{I}(\mathtt{X})$, containing all inversions in $\mathtt{X}$. A tuple in $\mathtt{T_a}$ is always in $\mathtt{I}(\mathtt{X})$, as a tuple which can be executed is an inversion, which gives that $\mathtt{T_a} \subseteq \mathtt{I}(\mathtt{X})$. An inversion in $\mathtt{I(X)}$ is also always a tuple in $\mathtt{T_a}$ that can be executed, as it is a pair of values that can be swapped by the algorithm, which gives that $\mathtt{I}(\mathtt{X}) \subseteq \mathtt{T_a}$. We can therefore say that $\mathtt{T_a} = \mathtt{I}(\mathtt{X})$.

We assume we perform a swap between a tuple $\mathtt{t_a} \in \mathtt{T_a}$. We will look at how the amount of inversions changes after this tuple is executed. For each inversion which contains both any position of $\mathtt{t_a}$ and a position outside the range between $\mathtt{t_a}$, we can say that this tuple will still be present after the swap, except with $\mathtt{t_a.i}$ replaced by $\mathtt{t_a.j}$ and vice versa. This is because for these values, the relative positions have not changed. We now assume that there are 3 possible values which can exist between the positions of $\mathtt{t_a}$, which can be denoted as $x < \mathtt{X}[\mathtt{t_a.j}] < y < \mathtt{X}[\mathtt{t_a.i}] < z$. An example of this would be the list $\{3, x = 0, y = 2, z = 4, 1\}$ where $\mathtt{t_a} =< 0, 4 >$. If we swap these positions, tuples $< 0, 1 >$ and $< 3, 4 >$ both remain enabled, as either value at the positions of $\mathtt{t_a}$ will always be in the incorrect order. The tuples $< 0, 2 >$ and $< 2, 4 >$ are both removed from $\mathtt{T_a}$, as after the swap these pairs are both in the correct order. In addition, the original inversion also disappears from $\mathtt{T_a}$. We get that the amount of inversions will decrease with $2n + 1$ inversions, where $n = |\{p \mid \mathtt{t_a.i} < p < \mathtt{t_a.j} \text{ and } \mathtt{X}[\mathtt{t_a.i}] > \mathtt{X}[p] > \mathtt{X}[\mathtt{t_a.j}]\}|$. This implies that the inversion number will always decrease when performing a swap between a valid inversion.

After a finite amount of swaps, we will reach a point where $\mathtt{inv}(\mathtt{X}) = 0$. At this point, we know that no more tuples can be executed, as the shared space is fully sorted. We can therefore conclude that the base sorting algorithm in tUPL will, after a finite amount of swaps, sort the full shared space. We can also conclude that the maximum number of swaps required to sort a shared space will always be equal to the inversion number of the shared space.

2.3 Related Work

As the tUPL framework is a relatively new framework, not much research has been performed on the possibilities that this framework poses. The initial concept for the framwork was described in 2013 as the forelem framework [RW13, Rie14]. This framework did not yet contain the whilelem loop, as this was later described by Prof. Dr. H.A.G. Wijshoff in unreleased slides.

Over time, algorithms such as K-means clustering [Hom17, HRW19], PageRank [vSRW17] and sparse matrix-vector multiplication [RW22] have been constructed in and optimised using tUPL. The principle of the forelem framework has also been used in the optimisation of database queries [RW14a, RW15]. To compile and run tUPL code, van der Zwaan created a compiler and frontend for tUPL named libtupl and Tython respectively [vdZ19]. This setup will not be used in the upcoming experiments, as we require more control over the order in which the loops visit the tuples.

Similar to this thesis, testing transformations on an algorithm in tUPL has been performed before. The sparse matrix-vector multiplication has been extensively transformed and tested in tUPL in a case study [RW22]. We attempt to take a similar approach in this thesis, with the exception that all transformations are loosely based on existing sorting algorithms, such as bubble sort and insertion sort.

3 Methods

In this section, we will discuss all methods and techniques that have been used in order to perform the experiments. In Section 3.1, we will discuss the different transformations that can be applied to the base algorithm. In Section 3.2, we will look at possibilities of combining these derivations in order to create an even more optimised algorithm.

3.1 Transformations

The first step to evaluating implementations of sorting algorithms, is defining a set of transformations which we can implement and combine in order to generate different algorithms. We will be looking at five different transformations. Sections 3.1.1 and 3.1.2 will focus on mandatory transformations, where these have to be applied in order to run reproducible experiments. These transformations remove the randomness of tUPL, which ensures the results are always consistent. Sections 3.1.3 to 3.1.5 will define optional transformations, which can be applied to the base algorithm in order to change the execution time of the algorithm. In all sections, we will review an implementation of the transformation in tUPL. It is however possible to perform these transformations during compilation solely on the visitation order of tuples. As this is not straightforward to display, we will instead look at code examples which achieve the same objective.

3.1.1 Limiting Reservoirs

In the elementary sorting algorithm in tUPL, the tuples in the tuple reservoirs are able to swap any two elements of the shared space as long as the elements in the given positions are not sorted. This opens up the possibility for a possible path where the minimal amount of swaps are performed to sort the shared space. In spite of this, as the selection of tuples is randomized, a high probability exists that a substantial amount of tuples which are selected will not be inversions. This, in turn, increases the execution time of the sorting algorithm. By implementing limiting of the tuple reservoirs, the algorithm will have less tuples to select from, which consecutively reduces the probability that a randomly selected finite chain of tuples will already be sorted.

As there exist $(|\mathtt{X}|!/2) \cdot 2^{((|\mathtt{X}|-1)(|\mathtt{X}|-2))/2}$ valid subsets of a given reservoir, it would be unfeasible to experiment with all possibilities. Instead, we will focus on a single minimal derivation, only allowing neighbouring elements to swap. Let $\mathtt{T'}$ be a tuple reservoir defined by:

$\mathtt{T'} = \{< \mathtt{i}, \mathtt{j} > \mid 0 \leq \mathtt{i} < |\mathtt{X}| - 1,\ \mathtt{j} = \mathtt{i} + 1\}$

Here, the reservoir $\mathtt{T'}$ will consist of only tuples where i and j are adjacent numbers, which can also be denoted as:

$\mathtt{T'} = \{< 0, 1 >, < 1, 2 >, ..., < |\mathtt{X}| - 2, |\mathtt{X}| - 1 >\}.$

This subset will allow any value to shift to any position, as long as it reduces the amount of inversions in the shared space. As it is a minimal derivation, only a single path exists which a value can take to move to the sorted position. This ensures, given perfect visitation of tuples, that the algorithm will have a consistent amount of swaps with each run for any given shared space.

3.1.2 Ordering

The order at which tuples are visited in whilelem and forelem loops is inherently random. This implies that any two runs of a non optimised tUPL algorithm containing either of the loops could differ in time elapsed or number of swaps. Even if the compiler performs notable optimisations, the statistics would still be dependent on which tuples are visited in which order. In theory, this randomness presents many opportunities for optimal runs; however, these randomised statistics would be impractical during testing. Test results can differ greatly depending on which tuples are visited in which order. To combat this, we introduce the ability to control the order at which tuples are visited. We attempt to eliminate the use of the whilelem loop, which creates a single sequential path of tuples.

As there are infinite different paths which can be traversed, we will be focusing on four paths: neighboring tuples and selections in both forward and backward direction. Neighboring tuples is equivalent to the optimisation discussed in Section 3.1.1, where only neighboring elements are allowed to swap. This would be beneficial for the cache as the principle of spatial locality is strongly adhered. For selections, we will look at the principle of selection sort. With selection sort, we sort the array by moving the minimum value of the unsorted part to the end of the sorted part. This can be simulated in tUPL, by traversing the reservoir in a specified order. In Listing 3.1, we can see an example of this. `T.i[curr]` denotes a subset of the reservoir `T` containing only tuples whose field `i` value equals `curr`. At the end of each iteration of the for loop, the value at the position `curr` will be the minimum value of the unsorted part of the shared space, which will in turn be the maximum value of the sorted part. This also has potential to benefit the cache, as here the principle of temporal locality is strongly adhered at position `curr`.

```
T = { < i,j > | 0 ≤ i < j < |X| }

for (int curr = 0; curr < |X| - 1; curr++) {
    forelem ( t; t ∈ T.i[curr] ) {
        if (X[t.i] > X[t.j]) {
            swap(X[t.i], X[t.j]);
        }
    }
}
```

Listing 3.1: Sorting algorithm in tUPL with selection

Both paths can be traversed in the forwards and backwards directions. With the forward direction, for each incremental step of i, we traverse all steps of j where i > j in order from $j = i + 1$ to $j = |X| - 1$. The backward direction is similar, except we take decremental steps for j and traverse all steps of i in decrementing order from $i = j - 1$ to $i = 0$. With neighboring tuples, this would result in only a single tuple for each step, whereas selections allows us to pick all possible tuples.

All four paths have a single sequential path defined for the visitation of tuples. We can therefore transform the whilelem loop of the base sorting algorithm to plain C code with for loops instead, displayed in Listing 3.2. By replacing the whilelem loops, it is possible to achieve consistent results throughout the testing of the transformations.

```
// Forward Neighboring
for (i = 0; i < |X| - 1; i++) {
    if (X[i] > X[i + 1]) {
        swap(X[i], X[i + 1]);
    }
}

// Backward Neighboring
for (i = |X| - 1; i > 0; i--) {
    if (X[i - 1] > X[i]) {
        swap(X[i - 1], X[i]);
    }
}

// Forward Selection
for (i = 0; i < |X| - 1; i++) {
    for (j = i + 1; j < |X|; j++) {
        if (X[i] > X[j]) {
            swap(X[i], X[j]);
        }
    }
}

// Backward Selection
for (j = |X| - 1; j > 0; j--) {
    for (i = j - 1; i >= 0; i--) {
        if (X[i] > X[j]) {
            swap(X[i], X[j]);
        }
    }
}
```

Listing 3.2: Ordering for sorting algorithms in C

3.1.3 Problem Reduction

The task of a sorting algorithm is to sort any input list of numbers. The task enforces the algorithm to sort the whole list, which can be a major task when working with large lists. With reducing the size of the problem, where the list is sorted in increasingly larger parts, we attempt to reduce the complexity as at each step the list is already partially sorted. An implementation of this is given in Listing 3.3. Here, `lim` denotes the size of the part of the shared space which will be sorted in the given iteration. This ensures we sort the shared space in single steps, rather than as a whole immediately. The starting point is `lim` = 1, to ensure the first pass of the shared space will consist only of the first two elements. To ensure that all tuples reside between the 0 and `lim`, we check that field `j` is less than or equal to `lim`. As field `i` is always lower than field `j`, it is guaranteed that the tuples able to be visited are all in the range $[0, \texttt{lim})$.

To validate whether this transformation still allows sorting, we will first setup a base case. If $|\texttt{X}| = 1$, the shared space is always sorted as it only consists of a single element. If we take $|\texttt{X}| \geq 2$ and $\texttt{lim} = 2$, the shared space is either sorted or can be sorted by performing the swap $< 0, 1 >$. The algorithm will continuously increment `lim` until it reaches the size of the full shared space. If we next assume $\texttt{lim} = n$ with $2 < n \leq |\texttt{X}|$, we assume that the part of the shared space in range $[0, n-2]$ is already sorted. As deduced earlier in Section 2.2, with the base sorting algorithm the shared space can be sorted in a finite amount of steps. This implies that this new range $[0, n-1]$ can also be sorted in a finite amount of steps. By induction this implies that this transformation is valid and allows the full shared space to be sorted.

```
T = { < i,j > | 0 ≤ i < j < |X| }

for (int lim = 2; lim ≤ |X|; lim++) {
    whilelem ( t; t ∈ T ) {
        if (t.j < lim && X[t.i] > X[t.j]) {
            swap(X[t.i], X[t.j]);
        }
    }
}
```

Listing 3.3: Sorting algorithm in tUPL with problem reduction

3.1.4 Divide and Conquer

As discussed in Section 3.1.3, it is possible to reduce the complexity of sorting a list by sorting said list in increasingly larger parts, compared to sorting the full list at once. A similar way to decrease the complexity of the problem, is to instead apply divide and conquer to the algorithm. The basic principle of this transformation is similar to the functionality of iterative merge sort. In iterative merge sort, the list which has to be sorted gets divided into chunks of size 1. Each two neighboring chunks are combined, or conquered, and afterwards sorted. This process gets repeated for steps of powers of 2, with the last step being the full list. This is visualised in Figure 3.1 as a full binary tree, with each leaf consisting of a single chunk of size 1. As the last chunk will not always be an exact power of 2, the tree is not perfect. Listing 3.4 displays a sorting function which simulates a similar path. Instead of breaking the shared space into small chunks, boundaries are set for which part has to be sorted. Here, `l` and `r` represent the respectively inclusive and exclusive boundaries of the current chunk of the shared space. Each chunk is sorted and combined. We make use of `size` to keep track of the size of a single chunk, hence why size gets multiplied by 2 at each step. As we make use of the forelem loop, we have the option to sort these chunks in parallel. This would increase the load on the processor, but in turn decrease the execution time.

To validate this transformation, we will follow the steps which the algorithm will take. The algorithm will split the shared space into chunks of length 1. Two chunks are combined and sorted, creating a single sorted chunk. From Section 2.2, we know that these chunks can be sorted, as these can be seen as individual shared spaces. At the root of the tree, the algorithm will behave exactly like the base algorithm, sorting the full shared space without any restrictions. The only difference here is that the shared space has already been sorted in two parts. This implies that an algorithm with only divide and conquer implemented will, at the root, sort the full shared space.

```
T = { < i,j > | 0 ≤ i < j < |X| }

for (int size = 1; size < |X|; size *= 2) {
    forelem ( l; l ∈ [0, 2 * size, 4 * size...] ) {
        whilelem ( t; t ∈ T ) {
            if (t.i ≥ l && t.j < l + 2 * size && X[t.i] > X[t.j]) {
                swap(X[t.i], X[t.j]);
            }
        }
    }
}
```

Listing 3.4: Sorting algorithm in tUPL with iterative divide and conquer

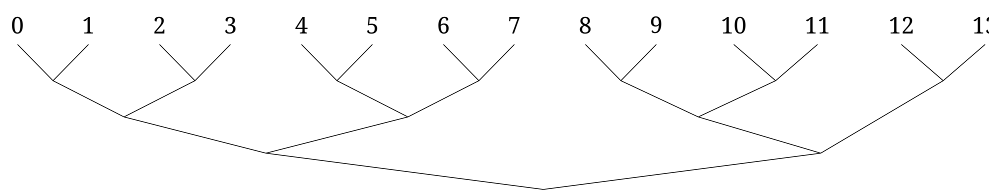

Figure 3.1: Full binary tree for iterative divide and conquer for 14 elements

3.1.5 Interval Sorting

In Sections 3.1.3 and 3.1.4, we looked at splitting a list into smaller chunks to sorted. Each chunk was defined as a short list of locally connected values. With interval sorting, we take another possible route where we look at chunks with intervals between each value. Listing 3.5 shows the algorithm which can be used to depict interval sorting. The `interval` contains the current interval at which the shared space will be sorted. Here the interval is based on floor divisions by 2, however this could also consist of powers of 2 or any other arbitrary values, as long as the last value will always be $\texttt{interval} = 1$. By making use of the forelem loop, all offsets of the interval could potentially be sorted in parallel, which would reduce the execution time of the algorithm.

Validating the transformation is similar to Section 3.1.4. We will sort each interval of increasingly larger chunks. Each interval can be sorted, as this is essentially sorting a chunk with a step size between the values. The reduction of the interval continues until we reach `interval` = 1. At this point, the only value for `offset` will be 0. This suggests that the shared space will be sorted with an offset of 0 from the start and a step size of 1, which is the same as sorting the full shared space.

```
T = { < i,j > | 0 ≤ i < j < |X| }

for (int interval = |X| / 2; interval > 0; interval /= 2) {
    forelem ( int offset; offset ∈ [0,interval-1] ) {
        whilelem ( t; t ∈ T ) {
            if (t.i % interval == offset &&
                t.j % interval == offset &&
                X[t.i] > X[t.j]) {
                swap(X[t.i], X[t.j]);
            }
        }
    }
}
```

Listing 3.5: Sorting algorithm in tUPL with intervals

3.2 Hybrid Algorithms

We have explored the possibilities of combining transformations in order to generate sorting algorithms in tUPL. As the transformations from Sections 3.1.3 to 3.1.5 are based on dividing the shared space into smaller chunks, we could also choose to combine certain algorithms based on the chunk size, creating a hybrid sorting algorithm. Hybrid sorting algorithms implement multiple algorithms and change the functionality based on the input list. One of these existing hybrid sorting algorithms is Timsort created by Tim Peters [Pet02]. Timsort uses a combination of merge sort and insertion sort to sort any given list.

A tUPL implementation, similar to the basis of Timsort but derived from the tUPL base specification using the above described transformations, is displayed in Listing 3.6. This algorithm consist of a combination of the divide and conquer and problem reduction transformations. With divide and conquer, the shared space is divided into chunks of size 1. In the example however, we split the shared space into chunks of size `CHUNK_SIZE` = 32. The first forelem loop goes over each chunk, sorting these individually using problem reduction. We encapsulate the problem reduction algorithm with a forelem loop, which indicates that the chunks can theoretically be sorted in any order and in parallel. The chunks are the basis for the second part, where we combine these chunks using the divide and conquer transformation.

```
T = { < i,j > | 0 ≤ i < j < |X| }

const int CHUNK_SIZE = 32;

// Divide into chunks
forelem ( l; l ∈ [0, CHUNK_SIZE, 2 * CHUNK_SIZE...] ) {
    // Problem reduction
    for (int lim = 2; lim ≤ CHUNK_SIZE; lim++) {
        whilelem ( t; t ∈ T ) {
            if (t.i ≥ l && t.j < l + lim && X[t.i] > X[t.j]) {
                swap(X[t.i], X[t.j]);
            }
        }
    }
}

// Conquer chunks
for (int size = CHUNK_SIZE; size < |X|; size *= 2) {
    forelem ( l; l ∈ [0, 2 * size, 4 * size...] ) {
        whilelem ( t; t ∈ T ) {
            if (t.i ≥ l && t.j < l + 2 * size && X[t.i] > X[t.j]) {
                swap(X[t.i], X[t.j]);
            }
        }
    }
}
```

Listing 3.6: Hybrid sorting algorithm based on Timsort in tUPL

4 Evaluation

We will now look at the performed tests and the findings gathered from these. In Section 4.1, we will look at the setup of the experiments and the input sets used. In Section 4.2, we will discuss the results of the algorithms and compare these against each other. In Section 4.3, we will compare these results against existing algorithms.

4.1 Experiment Setup

For these tests, a single architecture consisting of an Intel Core i7-8700 CPU at 3.20Ghz running Ubuntu Linux 18.04.6 has been used. The sizes of the L1 through L3 cache on this CPU are 32K, 256K and 12M. The compiler used is gcc 7.5.0.

To perform tests on the generated algorithms, we need a substantial set of sample data with different configurations. We will be testing each algorithm with 7 different data set sizes. Of the 7 sizes, 3 are used for testing the functionality of the algorithm and the output, while the other 4 are used to perform the actual tests. The first two sizes consist of 8 and 13 values. This is to test whether the algorithm is indeed capable of sorting a given list. The other functionality test consists of 256 values, which is used to check for any problems with larger inputs. The first true test set, labeled `L0`, is designed to be larger than the test cases, but still able to fit inside the L1 cache. For this reason, the `L0` set consist of 2 048 values, with a file size of 4K. The remaining test sets, labeled `L1` through `L3`, are sized to overflow the respective cache. This results in the inputs respectively containing 9 216 (36K), 81 920 (320K) and 4 194 304 (16M) values.

As we have 7 different input sizes, we have to stay within reasonable margin with the actual tests in each level. For all input sizes, we will be testing:

- values in ascending order,
- values in descending order,
- values in random order without duplicates twice,
- values in random order with duplicates twice.

With this setup, we attempt to capture the best and worst case time complexity of each algorithm, while also approximating an average case time complexity by using 4 random input lists.

Each level has been assigned a time limit, as it is not feasible to run all 2016 tests completely in a regular time span. The time limits for the `L0` through `L3` tests have been set at 1 minute, 5 minutes, 30 minutes and 2 hours. For this reason also, any algorithm that fails on a given input data set, will not run the larger version of that same input set on the next tests. If algorithm A is unable finish the descending data set on L1 in the given time, algorithm A will not run the descending data set on L2 or L3, as this can not be an optimal solution. This reduces the amount of tests that have to be executed and considerably reduces the execution time, as inadequate algorithms are filtered out in the early stages of the tests.

The naming system of the tests will follow $\texttt{sort\_}X_1X_2\texttt{\_}X_3$. X_1 and X_2 represents the path taken by the algorithm, defined in Section 3.1.2. These combinations consist of forwards neighboring (`fn`), backwards neighboring (bn), forwards selection (fs) and backwards selection (bs). X_3 represents which transformations from Sections 3.1.3 to 3.1.5 have been implemented and in which order. For example, sort_fs_23 uses selection in the forwards direction, where the shared space is first divided with divide and conquer and each chunk is sorted using interval sorting.

4.2 Results

We will discuss all results from the tests performed on all algorithms. In Section 4.2.1, we will look at the ascending data sets, where only validation of the list determines the execution time. In Section 4.2.2, we look at the descending data set, which will in most cases generate the worst case complexity. In Section 4.2.3, we look at both the unique and non-unique random data sets, which give an average case for all algorithms. All results can be found in Tables A.1 to A.4 of Appendix A.

4.2.1 Ascending Values

The most unambiguous test is the ascending list of values. The algorithm has to validate whether a given list is sorted. For the base algorithm in tUPL, this is done by validating whether no tuple can be visited that does not refer to two positions which are not sorted. This would imply that the full shared space is sorted. Each optional transformation introduces an extra step to the algorithm, which is wrapped around the whilelem loop. As the algorithm checks each chunk of shared space whether or not it is sorted, this will increase the execution time of the algorithm.

In Table A.1 of Appendix A, we can see the results of the algorithms. As the execution time of the algorithms varies heavily, we can not denote this in a proper graph. We instead plot Figure 4.1, which denotes which ascending input data sets have been finished within the set time limits for each algorithm. As an example, sort_Xn_1 was able to finish L0, L1 and L2, but not L3. All sorting algorithms with the same transformations and tuple reservoir are combined in this plot, as validation is not based on the direction of the validation, rather the contents of the tuple reservoir, the surrounding transformation and how validation is handled by these.

To validate whether a list is sorted, we visit each tuple and validate whether the elements at the positions of each tuple are sorted. For a full shared space, this would take $|X| - 1$ comparisons with only neighboring swaps, as we have to check $\{(0,1), (1,2), ..., (|X|-2, |X|-1)\}$. With selection swaps however, this would take $\sum_{n=1}^{|X|-1} n = (|X| \cdot (|X|-1))/2$ comparisons. The base algorithm will check the full shared space a single time, which takes $\mathcal{O}(n)$ time with neighboring swaps and $\mathcal{O}(n^2)$ time with selection swaps to perform. The base algorithm with neighboring swaps is in turn the fastest best case algorithm. The difference between neighboring and selection swaps is also visible in Figure 4.1. We can see that for each pair of algorithms, the selection swap performs equal to or worse than the neighboring swap counterpart.

Algorithms with problem reduction implemented take the longest time to execute. This can be explained by checking how many validations have to be performed. In problem reduction, we start by reducing the problem to two elements. This is validated by a single comparison, denoted by $\{(0,1)\}$. After introducing a third value, we introduce an extra comparison between $(1,2)$. As we have to validate the full chunk of shared space, we have to perform the original comparison $(0,1)$ again in this

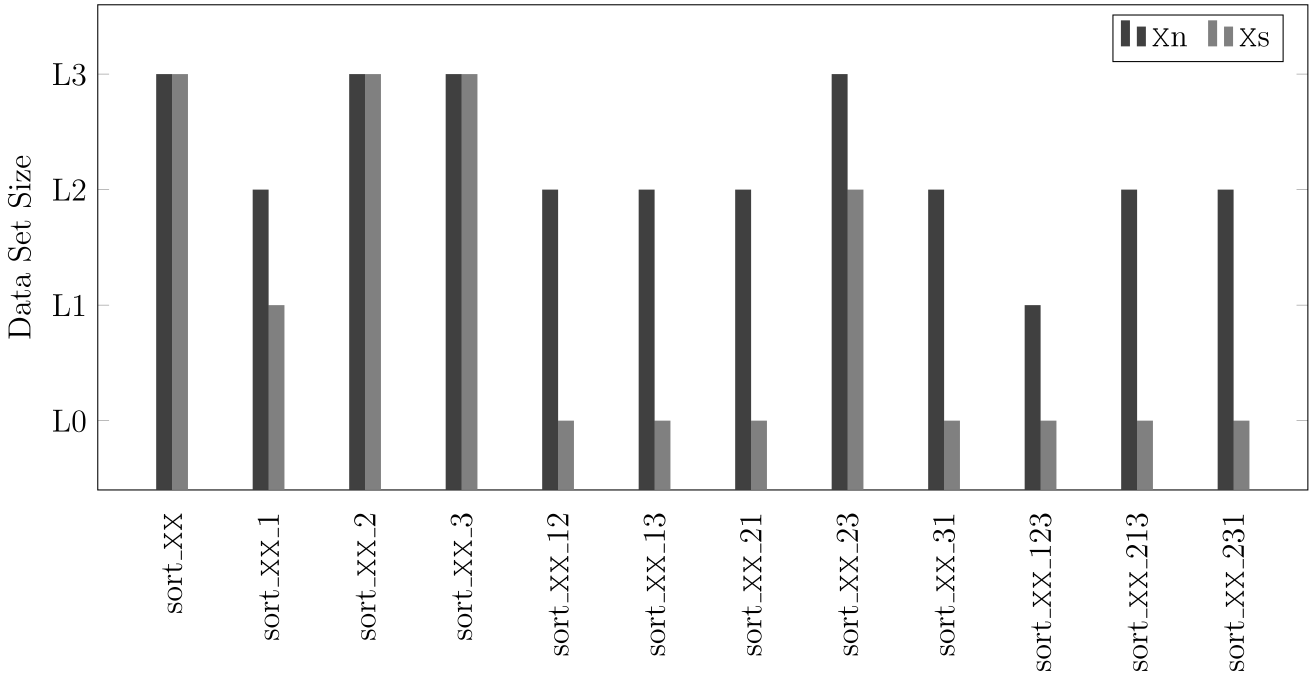

Figure 4.1: Finished ascending data sets within the time limit for each algorithm

validation. With four elements, we introduce yet another comparison. This continues until the last element is added to the sorting space. We have now performed $1+2+3+...+(|X|-1) = (|X|\cdot(|X|-1))/2$ comparisons, which means any algorithm with problem reduction implemented will take $\mathcal{O}(n^2)$ time on validation of the ascending list.

To calculate the best case time complexity of divide and conquer, we will use a shared space where $|X| \in \{2^n \mid n \in \mathbb{N}^+\}$. To simplify the steps, we will work backwards through the algorithm. The final validation will consist of the full shared space, performing $|X| - 1$ comparisons. The second to last step compares two chunks of size $|X|/2$. As there is only a single pair which will not be checked, namely $(|X|/2 - 1, |X|/2)$, we can deduce that this step performs $|X| - 2$ comparisons. We can continue this until we arrive at chunks of size 2. As we constantly divided all chunks into 2 equal parts, we can deduce that this has taken $\log_2(|X|)$ steps. As in each step we have checked $|X| - m$ $(1 \leq m \leq |X|/2)$ pairs, we can deduce that the base algorithm with the divide and conquer transformation applied will take $\mathcal{O}(n \log n)$ time to validate the ascending list.

The last transformation to calculate is interval sorting. As interval sorting and divide and conquer are similar, we will compare the two and deduct the best case time complexity from this. In divide and conquer, we split the shared space into chunks of half the size of the previous chunks. This is done similarly in interval sorting, with the difference that the chunks are now interleaved with each other. This does however not change anything about the amount of comparisons required for validation. Each chunk has a size half of the predecessor, with the final chunks in both algorithms containing 2 elements. This concludes that divide and conquer and interval sorting use the same size of chunks and therefore have the same time complexity, thus we can say that interval sorting takes $\mathcal{O}(n \log n)$ time to validate the ascending list.

4.2.2 Descending Values

A list of descending values is often the basis for the worst case scenario of a sorting algorithm [Sed78, Sha15, MAÇ17]. The algorithm must perform the maximum amount of swaps in order to achieve a sorted list, as values have to move to the exact opposite side of the list in order to be sorted. This issue however is not relevant to the base tUPL algorithm. As the tuple reservoir contains tuples which connect each position in the shared space to any other, it is theoretically possible to sort a descending shared space in $\lfloor |\mathbb{X}|/2 \rfloor$ swaps. In the actual tests however, we make use of different predefined paths, as these paths are focused on sorting any shared spaces, rather than only the descending space.

All results can be found in Table A.2 of Appendix A. As we must perform swaps now, contrary to Section 4.2.1, we will use both the execution time and the number of swaps as the metric. If the algorithm was unable to finish, the percentage of the list that was sorted is also displayed. This percentage is based on the inversion number divided by the total size of the default tuple reservoir. Figure 4.2 denotes which descending input data sets have been finished within the set time limits for each algorithm. As the direction now influences the speed of the algorithm, all algorithms have been plotted.

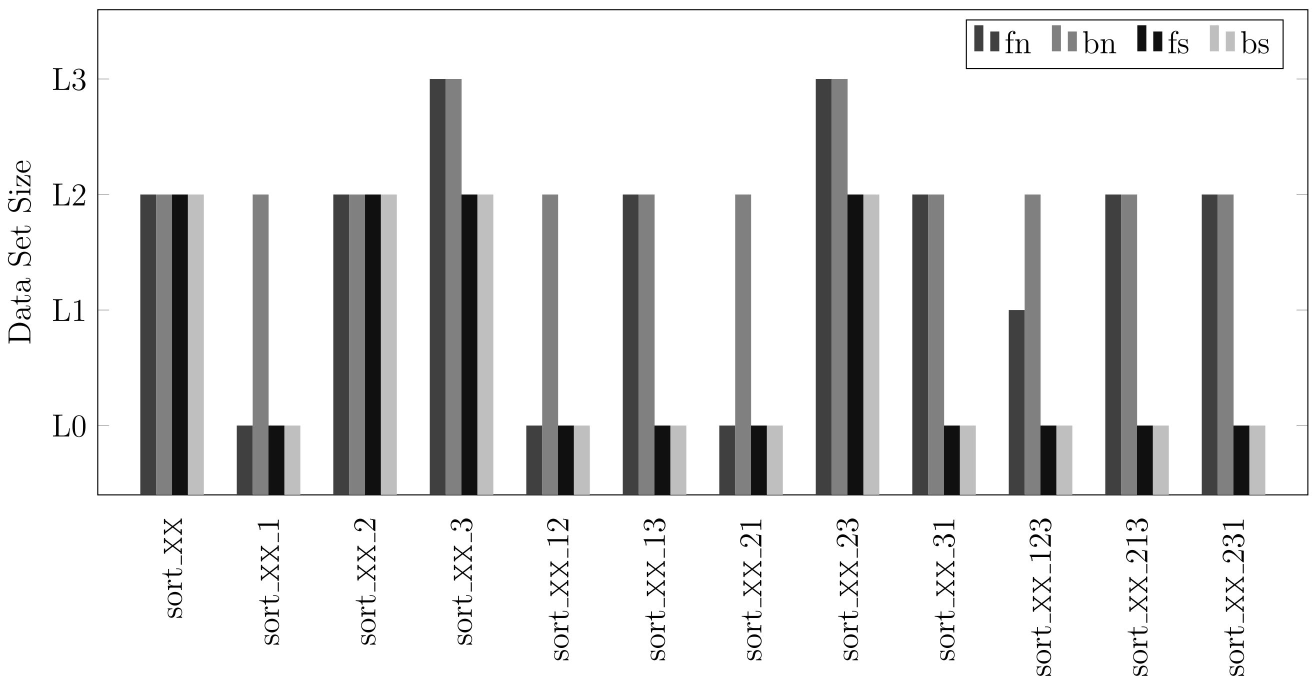

Figure 4.2: Finished descending data sets within the time limit for each algorithm

As we can see from the raw results, the base algorithm using selection swaps performs better than the same algorithm with neighboring swaps. This behaviour seems to not adhere to what was deduced in Section 4.2.1, where selection swap took longer as more validations had to be performed. As we look into the steps that the algorithm takes however, we deduct why this is the case. We will take the forward direction algorithm as example. We look at the first $|\mathbb{X}| - 1$ iterations of the whilelem loop. In this time, sort_fn will have moved the first element, containing the highest value, to the end of the shared space. All other values in the shared space will have shifted one position to the left. In the same amount of iterations, sort_fs will have moved the lowest value from the end of the shared space to the front, shifting all other values one spot to the right. The next step for sort_fn would take $|\texttt{X}| - 1$ steps, as the algorithm will have to visit all tuples, including the final tuple which will be sorted. Sort_fs on the other hand only requires $|\texttt{X}| - 2$, as all tuples in `T.i[1]` will now be validated, which does not include the tuple $< 1, 0 >$. At each step, the amount of tuples that has to be validated decrements. This implies that sort_fs can sort a list in half the amount of comparisons it takes sort_fn to sort the same list. Both algorithms have a time complexity of $\mathcal{O}(n^2)$, as sort_fn will perform $(|\texttt{X}| - 1)^2$ swaps and sort_fs will perform $((|\texttt{X}| - 1)(|\texttt{X}| - 2))/2$ swaps, which is the which is the worst case scenario for both algorithms.

In sort_XX_1, sort_XX_12 and sort_XX_21, we can see a peak in the backwards neighboring algorithms. We deduced in Section 4.2.1 that the problem reduction transformation requires the longest time to validate an ascending list. With the addition of having to sort the shared space, it could be assumed that all algorithms with problem reduction would have the lowest performance overall. If we look at the raw data in Table A.2 however, we can see that sort_bn_1 performs similar to sort_bn and only slightly below sort_bn_2. We can determine why this occurs by looking at an iteration of all four orderings. We will use sort_XX_1 for this example. We assume a shared space X with a chunk of size m, which is sorted for $[0, m - 2]$ and contains the minimal element at position $m - 1$. The algorithm has to move the new element from the last position $m - 1$ to the first position of the sorted part.

We will first look at the selection algorithms, starting with forward selection. The algorithm starts by checking all tuples in `T.i[0]`. Here, the lowest value gets moved to position 0 using the last tuple $< 0, m - 1 >$. The problem now is that the value which was originally at position 0, has been moved to position $m-1$. As all values, except for the new value, have to shift a single position to the right, this value needs to be moved to position 1. The algorithm will now visit all tuples in `T.i`[1], eventually moving the value at position $m - 1$ to position 1. We have now repeated the problem, where the value which should be at position 2 is at position m. We can deduce that, for every value of $0 \leq q < m - 1$, if we place the correct value at that position, the value which is supposed to be at position $q + 1$ will be at position m instead. We can see an example of this in Listing 4.2. As the amount of tuples which get validated for each incremental position q gets decremented, we can say that to append a minimal position to a sorted chunk, it would take $\sum_{k=1}^{m-1} k$ comparisons. As this has to be done for all values $2 \leq m \leq |\texttt{X}|$, we can determine that the whole algorithm will take $\sum_{m=2}^{|\texttt{X}|} \sum_{k=1}^{m-1} k$ comparisons in total, which is equivalent to a time complexity of $\mathcal{O}(n^3)$.

```
{1, 2, 3, 0}
{1, 2, 3, 0}
{1, 2, 3, 0}
{0, 2, 3, 1}
{0, 2, 3, 1}
{0, 1, 3, 2}
{0, 1, 2, 3}
```

Listing 4.1: Steps to sort a decreasing shared space chunk using sort_fs_1

As we have seen, we require $\sum_{k=1}^{m} k$ to sort a single part of size $m+1$ with problem reduction with forwards selection. We will now be looking how backwards selection handles this. In Listing 4.2, we can see the steps which the algorithm performs. We see that this is similar to Listing 4.1 depicting sort_fs_1, with the difference being that instead of moving a value to the back of the sorted part, we now only move the minimal value a single position to the left for each iteration. If we compare the two algorithms, we can deduce that this would also take $\sum_{k=1}^{m} k$ comparisons for each position. From this, we can conclude that this algorithm also has a time complexity of $\mathcal{O}(n^3)$ and takes the same amount of comparisons.

```
{1, 2, 3, 0}
{1, 2, 0, 3}
{1, 2, 0, 3}
{1, 2, 0, 3}
{1, 0, 2, 3}
{1, 0, 2, 3}
{0, 1, 2, 3}
```

Listing 4.2: Steps to sort a decreasing shared space chunk using sort_bs_1

We now know that sorting with selection sort has a time complexity of $\mathcal{O}(n^3)$ for both forward and backward selection. We will now look at neighboring swaps, starting with the forward direction. We move through the sorted part using $m-1$ tuples. From these tuples, the only one which performs a swap is the final one, as this tuple refers to the maximum value of the sorted part and the new minimum value. This places the minimal value at position $m-2$. The algorithm will now go over all tuples again, swapping only the positions of the second to last tuple. This pattern continues until the minimal value is at the second position. With a single swap, the chunk of the shared space will be sorted. Listing 4.3 displays this pattern. As we move the value to the second position in $m-2$ loops, we can deduce that this will take $(m-1)\cdot(m-2)+1$ comparisons. As this has to be performed for all values of m, we get that sorting with sort_fn_1 will take $\sum_{m=2}^{|\mathrm{x}|}((m-1)\cdot(m-2)+1)$ comparisons. This in turn results in a time complexity of $\mathcal{O}(n^3)$.

```
{1, 2, 3, 0}
{1, 2, 3, 0}
{1, 2, 3, 0}
{1, 2, 0, 3}
{1, 2, 0, 3}
{1, 0, 2, 3}
{1, 0, 2, 3}
{0, 1, 2, 3}
```

Listing 4.3: Steps to sort a decreasing shared space chunk using sort_fn_1

We have seen that all other algorithms for problem reduction sort had a time complexity of $\mathcal{O}(n^3)$. We will now explore why backwards neighboring swap sorting stands head and shoulders above the other algorithms. We perform a single pass of the tuple reservoir. The first tuple moves the minimal value a single step to the left and places the maximum value at the correct position. The next tuple performs the same operation, moving the minimal value further to the left and placing the second largest value at the correct position. This continues, until all tuples have been visited. At this point, the minimal value has been moved to the left and all other values have been shifted to the correct positions. Listing 4.4 shows the steps the algorithm takes. This implies that sort_bn_1 can sort a chunk of size m in $m-1$ comparisons. Performing this on all chunks, we get a total of $\sum_{m=2}^{|X|}(m-1)$ comparisons to sort the full shared space. This results in a time complexity of $\mathcal{O}(n^2)$, which is significantly lower than the time complexity of the other algorithms.

```
{1, 2, 3, 0}
{1, 2, 0, 3}
{1, 0, 2, 3}
{0, 1, 2, 3}
```

Listing 4.4: Steps to sort a decreasing shared space chunk using sort_bn_1

From all algorithms, we can see from Figure 4.2 that only sort_Xn_3 and sort_Xn_23 have managed to finish the L3 descending input set. From the raw data in Table A.2, we can find that the fastest algorithms in this set are sort_Xn_3. We can also see that there is a huge margin between both sort_Xn_3 and sort_Xs_3, and sort_Xn_23 and sort_Xs_23. This can both be explained by looking at how this transformation affects the order of the tuples at each iteration of the interval. As we apply interval sorting to each individual chunk, we will focus on sort_fX_3, as this concept can be generalised to each individual chunk of sort_fX_23. We know that the size of the first interval will be of distance $|X|/2$. If we follow the minimal value in the shared space, in this case 0, we can see from Listing 4.5 that this would consist of performing $\log |X|$ swaps for each value. As each swap moves two values, this means we need $|X| * \log |X|/2$ swaps to sort the full share space. If we look at sort_Xs_3 instead, when we reach an interval of $|X|/4$ and for example an offset of 0, we swap tuple $<0, 4>$, but we also validate $<0, 8>$ and $<0, 12>$. Where sort_Xn_3 performs one additional comparison for the second interval, sort_Xs_3 performs 4 additional comparisons. This greatly increases the execution time of sort_Xs_3, which clarifies the difference in execution time between sort_Xn_ and sort_Xs_3.

```
{15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0}
{ 7,  6,  5,  4,  3,  2,  1,  0, 15, 14, 13, 12, 11, 10,  9,  8}
{ 3,  2,  1,  0,  7,  6,  5,  4, 11, 10,  9,  8, 15, 14, 13, 12}
{ 1,  0,  3,  2,  5,  4,  7,  6,  9,  8, 11, 10, 13, 12, 15, 14}
{ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15}
```

Listing 4.5: Steps to sort a decreasing shared space of $|X| = 16$ using sort_Xn_3

4.2.3 Random Values

The most general use case of a sorting algorithm is to sort lists with a random permutation. If we were to know the precise permutation of the input list beforehand, we could use a specific algorithm in order to always sort the list in an optimal time. As we do not know this in advance, we use sorting algorithms in order to handle all possible cases. In tUPL, it is theoretically possible to sort every possible shared space permutation in $|X| - 1$ swaps or less. This would require a perfect algorithm however, which could be considered an impossible task. As every possible permutation is allowed, the average case time and space complexity can be derived from these inputs.

Tables A.3 and A.4 of Appendix A display all raw results. For both input types, two tests have been performed. In both cases, the algorithms reached similar results, where the final data set size was always the same. As these sets contain random permutations, the average result has been displayed instead for both cases. Similar to the other sections, Figure 4.3 denotes which random input data sets have been finished within the set time limits for each algorithm. As both the unique random and non unique random results have the same final data set sizes for each algorithm, a single plot has been used to display both input sets.

If we compare this plot with Figure 4.2, we see that these plots are very similar. The exception to this is sort_bn_123. As we can see in Table A.2, sort_bn_123 took 28,5 minutes to sort L2. As the limit for sorting an L2 set is 30 minutes, this could potentially be an outlier result where the function managed to sort the shared space, while the average execution time for this is higher than the set time limit.

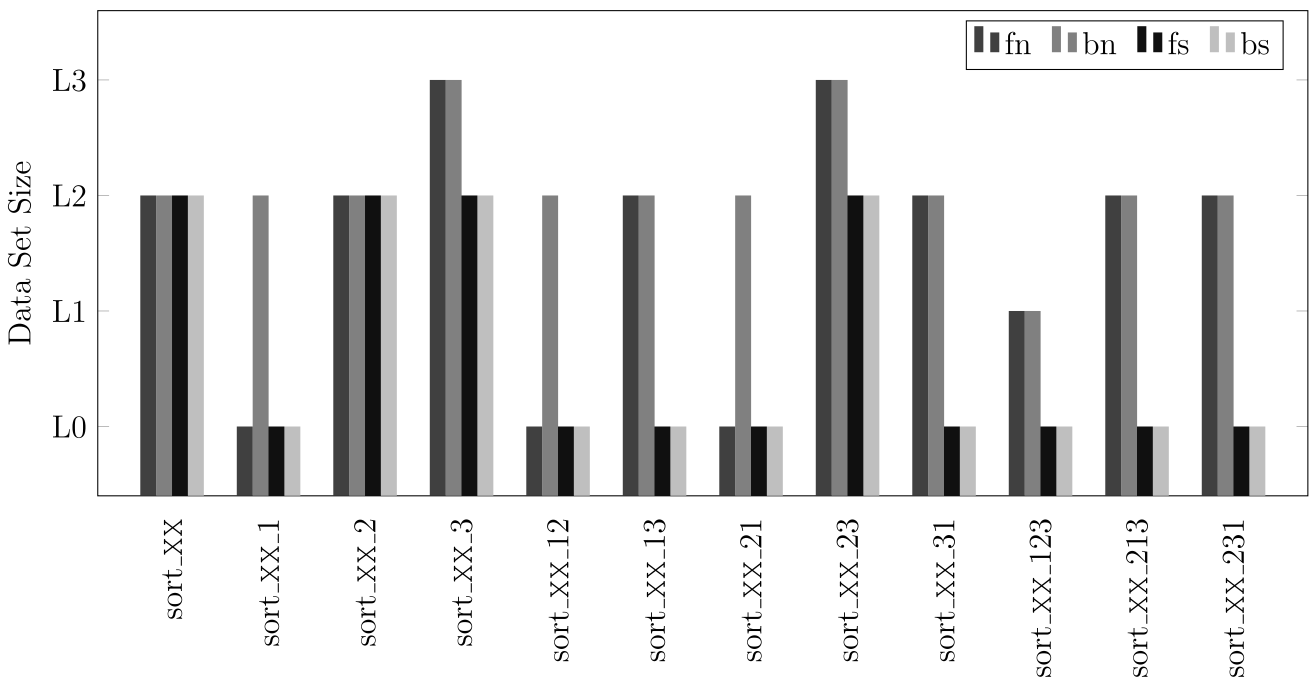

Figure 4.3: Finished random data sets within the time limit for each algorithm

From looking at Figures 4.2 and 4.3, we can see that only sort_Xn_3 and sort_Xn_23 were able to finish data set L3. With what we have discussed in Sections 4.2.1 and 4.2.2, we can deduce why these four sorting algorithms stand out from the rest. In Section 4.2.1, we have seen that the problem reduction transformation greatly increases the amount of comparisons required to validate any permutation of a shared space. The exceptions for this, as discussed in Section 4.2.2, are algorithms with backwards neighboring swaps. We have seen that these algorithms perform better than equivalent algorithms with different tuple ordering. We can see similar results in the random results. Additionally, if we look at the raw data, we can see that the percentage sorted of sort_bn_1, sort_bn_12 and sort_bn_21 is below 10% for the L3 descending data set, while the same algorithms reach over 50% sorted for both L3 random data sets. We know however, from the ascending data sets, that the upper limit for these algorithms is L2, as these were not able to validate the L3 data sets. From this, we can deduce that the only algorithms with potential to finish L3 are sort_XX, sort_XX_2, sort_XX_3 and sort_XX_23. This is also consistent with Table A.1, where these algorithms, excluding sort_Xs_23, were able to validate L3.

We are able to exclude sort_XX and sort_XX_2 from the list by looking at the influence of the algorithms on the inversion number. We can say that for sort_Xn and sort_Xn_2 that the inversion number will only be able to reduce by 1 for each swap. As we have discussed in Section 2.2, the inversion number can only decrease by a value bigger than 1 if any value between the two positions of the swap is an inversion with both positions before the swap. As there are no values between the tuples with neighboring swaps, the algorithm will always require I(X) swaps to sort the given shared space. This argument can also be made for sort_Xs and sort_Xs_2. We will take for example sort_fs with the shared space $\{3, 1, 2, 0\}$. We can see that the shared space could be sorted by a single swap $< 0, 3 >$. However, sort_fs will first perform the swap $< 0, 1 >$, resulting in $\{1, 3, 2, 0\}$. We now know that the values at these positions are in the correct order. If we were to continue, we will find tuple $< 0, 3 >$. We know however that all values between 0 and 3 are are not inversions with 0. For this reason, there can be no additional tuples that will be removed when performing a swap. We can also see that sort_XX_3 and sort_XX_23 are not affected by this, as these both implement interval sorting, which allows gaps between the tuple positions.

When we compare the execution times for sort_Xn_3 and sort_Xn_23, we can see some interesting properties of these algorithms. For the descending data set in Table A.2, we can see that sort_Xn_3 takes roughly 1.37 seconds to sort the shared space, while sort_X_23 takes an average of 4.0 seconds. If we look at the random data sets in Tables A.3 and A.4 however, we can see that sort_X_23 takes approximately 8.3 seconds to sort the shared space, while sort_X_3 takes 131.6 seconds on average. These two comparisons show a strange difference between the algorithms. We can deduce why sort_Xn_3 takes long for the random data sets. If we look at Listing 4.5, we see the path which the minimum value takes. If we were to place this value at another position however, this shortest path is not guaranteed to exist anymore. If a value lands at a position, where it only leaves at a very low interval, this value has to move many positions in order to reach the correct position. This also gives an indication as to why, for the random data sets, sort_bn_3 performs slightly better than sort_fn_3. This is similar to what was discussed in Listings 4.3 and 4.4, where a value moves to the correct position quicker using backward neighboring swaps. It can be speculated that, for the random input sets, values had to be moved further to the left on average than to the right. For this reason, a slight difference in execution time can be measured.

4.3 Comparing with existing algorithms

We have discussed the results of the transformations which we have applied to the base tUPL sorting algorithm. In Section 3.2, we briefly touched upon the concept of a hybrid algorithm, where multiple algorithms are combined to form a single algorithm. We have looked at Timsort and discussed a tUPL algorithm based on Timsort, however this algorithm is only a partial implementation of the full Timsort algorithm. We will be using the a C99 implementation of Timsort to test and compare the algorithm [Per16]. We will also be looking at two other hybrid algorithms, namely qsort and introsort. Qsort is the default glibc sorting algorithm. It is a hybrid sorting algorithm consisting of quicksort and insertion sort. The algorithm first partially sorts the input list using quicksort, after which it uses insertion sort to finish the sorting process. To test this we will be using the base implementation from the stdlib.h library. Introsort is the default sorting algorithm of the C++ STL library. It is also a hybrid algorithm based on quicksort, heapsort and insertion sort. The algorithm begins with quicksort. It switches to heapsort when the recursion depth exceeds a value based on the logarithm of the number of elements in the current list. It switches to insertion sort when the number of elements to be sorted is below a given threshold.

The results for these algorithms can be seen in Table A.5. From looking at the table, we can see that Timsort has the lowest execution time for both the ascending and descending data sets. We can also see that for the random data sets, qsort performs the best. Introsort performs worst in the ascending and descending data sets and places second in the random data sets. We can compare this to the results which we acquired from the previous experiments in Section 4.2. We will compare the fastest algorithm from each data set in L3 to the hybrid algorithms.

For the ascending set, we find the fastest algorithm to be sort_Xn, which was also concluded in Section 4.2.1. If we compare this to the hybrid algorithms, we find that Timsort performs this task in around half the time sort_Xn takes. If we look at the descending set, sort_Xn_3 performs this in 1.39 seconds. All hybrid algorithms on the other hand manage to execute this task in under 0.25 seconds, with Timsort being the fastest with under 0.02 seconds. Looking at the unique random and non-unique random sets, we find that sort_bn_23 is the fastest with 8.2 seconds to sort the random shared spaces. The same random sets were used to test the hybrid algorithms. We fidn that all algorithms manage to perform this sort in under a second, with qsort taking slightly over half a second.

We find that for each case, there exists a hybrid algorithm which can perform the task quicker. We can derive this conclusion from the definition of a hybrid algorithm. A hybrid algorithm is a combination of multiple algorithms which can efficiently sort a list based on the input data. With the transformations, we have only been looking at using a single algorithm composed of multiple transformations. An example for this would be Listing 3.6, which combines divide and conquer and problem reduction based on the size of the chunk size from divide and conquer. It would be possible for a compiler to, with modification of tuple visitation order, simulate a hybrid algorithm.

5 Discussion and Conclusion

We have explored different transformations on the tUPL base sorting algorithm and compared the results between themselves and to other existing algorithms. By defining five transformations, we were able to evaluate 48 different algorithms which can all be reduced to tuple paths which the base algorithm can perform. We found that the sort_Xn_3 and sort_Xn_23 managed to achieve the best performance compared to the other algorithms.

While we did manage to find algorithms which performed better than the other algorithms, this does not define these algorithms as being true best algorithms. As stated in Sections 3.1.1 and 3.1.2, there exist many more paths which could be traversed, which would all provide different results. It would be unfeasible to test all different paths, as this would theoretically take infinite time to perform. It could be a possibility to change this to an optimisation instead of a transformation, where the compiler dynamically generates the order instead of using a predefined order.

We also found that, even though some algorithms manage to sort L3 in a short time, the currently existing hybrid algorithms display better results compared to the tUPL transformed algorithms. As these algorithms are based on finding near-optimal solutions for multiple smaller problems instead of a single big problem, these algorithms can perform better in most general cases. As stated in Section 4.3 and demonstrated by Listing 3.6, it would be feasible for the base tUPL sorting algorithm to be transformed into a hybrid tUPL algorithm by the compiler. This would require further investigation on which combination of algorithms display significant improvement in any way. The defined transformations and the gathered results could serve as a basis for these hybrid algorithms.

Due to the time it took to conduct all tests, some results could have contained outlier values, such as sort_bn_123 described in Section 4.2.3, where these would have been filtered out with average tests instead. Even though the data has a risk of containing slight outliers, we can say that the data fits into the provided explanations, which implies that it is likely that no more outliers are present in the data. Additionally, as the random data is composed of two different input sets, the chances of outliers in these sets are already lowered.

References

[Hom17] Anne Hommelberg. Using the Forelem Framework to Express and Optimize K-means Clustering. Master's thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2017.

[HRW19] Anne Hommelberg, Kristian F. D. Rietveld, and Harry A. G. Wijshoff. Abstracting Parallel Program Specification: a Case Study on k-Means Clustering. In *Proceedings of the 16th conference on Computing Frontiers (CF '19)*, pages 279–282, apr 2019. doi:10.1145/3310273.3322828.

[MAÇ17] Adnan Saher Mohammed, Şahin Emrah Amrahov, and Fatih V Çelebi. Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort. *Future Generation Computer Systems*, 71:102–112, 2017.

[Man85] Heikki Mannila. Measures of Presortedness and Optimal Sorting Algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, 1985. doi:10.1109/TC.1985.5009382.

[Per16] Patrick O. Perry. timsort, 2016. URL: https://github.com/patperry/timsort/.

[Pet02] Tim Peters. Timsort description, 2002. URL: https://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[PF16] Zahida Parveen and Nazish Fatima. Performance Comparison of Most Common High Level Programming Languages. *International Journal of Computing Sciences Research*, 5:246–258, 10 2016.

[Rie14] Kristian F. D. Rietveld. *A versatile tuple-based optimization framework*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2014.

[RW13] Kristian F. D. Rietveld and Harry A. G. Wijshoff. Forelem: A Versatile Optimization Framework For Tuple-Based Computations. In *Proceedings Workshop on Compilers for Parallel Computing*, 2013.

[RW14a] Kristian F. D. Rietveld and Harry A. G. Wijshoff. Re-engineering compiler transformations to outperform database query optimizers. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 300–314. Springer, 2014.

[RW14b] Kristian F. D. Rietveld and Harry A. G. Wijshoff. Towards a New Tuple-Based Programming Paradigm for Expressing and Optimizing Irregular Parallel Computations. *Proceedings of the 11th ACM Conference on Computing Frontiers, CF 2014*, 05 2014. doi:10.1145/2597917.2597923.

[RW15] Kristian F. D. Rietveld and Harry A. G. Wijshoff. Reducing Layered Database Applications to Their Essence through Vertical Integration. *ACM Trans. Database Syst.*, 40(3), oct 2015. doi:10.1145/2818180.

[RW22] Kristian F. D. Rietveld and Harry A. G. Wijshoff. Automatic Compiler-Based Data Structure Generation, 2022. doi:10.48550/ARXIV.2203.07109.

[Sed78] Robert Sedgewick. Implementing Quicksort Programs. *Commun. ACM*, 21(10):847–857, oct 1978. doi:10.1145/359619.359631.

[Sed98] Robert Sedgewick. *Algorithms in c++, parts 1-4: fundamentals, data structure, sorting, searching*. Pearson Education, 1998.

[Sha15] Vipul Sharma. A New Approach to Improve Worst Case Efficiency of Bubble Sort. *International Research Journal of Computer Science (IRJCS) ISSN*, pages 2393–9842, 2015.

[vdZ19] Dennis van der Zwaan. A Data Structure Optimizing Compiler for tUPL. Master's thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2019.

[vSRW17] Bart van Strien, Kristian F. D. Rietveld, and Harry A. G. Wijshoff. Deriving Highly Efficient Implementations of Parallel PageRank. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 95–102, 2017. doi:10.1109/ICPPW.2017.26.

A Result Data

A.1 Ascending

Algorithm	L0 Time (s)	L1 Time (s)	L2 Time (s)	L3 Time (s)
sort_Xn	0.000017	0.000072	0.000635	0.031877
sort_Xs	0.006774	0.196161	5.112715	4.310384
sort_Xn_1	0.016496	0.332075	28.066313	DNF
sort_Xs_1	6.016598	267.841482	DNF	-
sort_Xn_2	0.000192	0.000960	0.010031	0.636601
sort_Xs_2	0.020549	0.531700	26.297225	813.156236
sort_Xn_3	0.000089	0.000553	0.007266	0.601360
sort_Xs_3	0.026245	0.579283	37.944788	3117.971472
sort_Xn_12	0.158135	3.729265	427.548668	DNF
sort_Xs_12	6.697171	DNF	-	-
sort_Xn_13	0.102044	1.567635	251.368989	DNF
sort_Xs_13	18.593539	DNF	-	-
sort_Xn_21	0.032955	0.859541	62.670340	DNF
sort_Xs_21	6.793023	DNF	-	-
sort_Xn_23	0.000815	0.006199	0.075374	3.703634
sort_Xs_23	0.069800	1.449062	100.475182	DNF
sort_Xn_31	0.021531	0.869401	41.904999	DNF
sort_Xs_31	8.984876	DNF	-	-
sort_Xn_123	0.512934	21.922393	DNF	-
sort_Xs_123	29.212412	DNF	-	-
sort_Xn_213	0.186770	6.765592	502.119122	DNF
sort_Xs_213	22.972763	DNF	-	-
sort_Xn_231	0.087759	1.416610	127.899332	DNF
sort_Xs_231	19.839762	DNF	-	-

Table A.1: Average time taken to validate an ascending list

A.2 Descending

Algorithm	L0 Nr of Swaps Time (s) (% Sorted)	L1 Nr of Swaps Time (s) (% Sorted)	L2 Nr of Swaps Time (s) (% Sorted)	L3 Nr of Swaps Time (s) (% Sorted)
sort_fn	2096128 0.045111	42462720 0.892366	3355402240 75.664569	1151307395944 DNF (13.08%)
sort_bn	2096128 0.036351	42462720 0.931791	3355402240 78.167186	1103976220655 DNF (12.55%)
sort_fs	2096128 0.016607	42462720 0.621848	3355402240 22.550353	5732269103817 DNF (65.16%)
sort_bs	2096128 0.031381	42462720 0.345442	3355402240 40.984008	5698810173320 DNF (64.78%)
sort_fn_1	2096128 21.328590	13589308 DNF (32.00%)	-	-
sort_bn_1	2096128 0.046968	42462720 0.757995	3355402240 58.277720	569909126791 DNF (6.47%)
sort_fs_1	2096128 11.948158	37260028 DNF (87.74%)	-	-
sort_bs_1	2096128 12.905672	24085270 DNF (56.72%)	-	-
sort_fn_2	2096128 0.050043	42462720 1.478048	3355402240 101.435200	600937741902 DNF (6.83%)
sort_bn_2	2096128 0.052863	42462720 0.964709	3355402240 71.586795	580670193664 DNF (6.60%)
sort_fs_2	2096128 0.051448	42462720 0.886357	3355402240 57.758699	2303004434961 DNF (26.18%)
sort_bs_2	2096128 0.059783	42462720 1.343066	3355402240 56.225755	2428425717735 DNF (27.60%)
sort_fn_3	11264 0.000343	54272 0.001004	638976 0.011970	46137344 1.369404
sort_bn_3	11264 0.000339	54272 0.001112	638976 0.019076	46137344 1.383927
sort_fs_3	11264 0.040583	54272 0.657869	638976 54.981840	41521578 DNF (99.99%)
sort_bs_3	11264 0.040253	54272 0.953501	638976 38.649166	41490432 DNF (99.99%)
sort_fn_12	2096128 16.235893	15547400 DNF (36.61%)	-	-
sort_bn_12	2096128 0.175443	42462720 4.220805	3355402240 462.239917	67691212996 DNF (0.76%)
sort_fs_12	2096128 9.090255	26423941 DNF (62.22%)	-	-

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_bs_12	2096128 14.901833	25632284 DNF (60.36%)	-	-
sort_fn_13	2096128 0.135053	42462720 2.969532	3355402240 498.669452	186012486910 DNF (2.11%)
sort_bn_13	2096128 0.208544	42462720 3.865258	3355402240 265.505980	223304128014 DNF (2.53%)
sort_fs_13	2096128 29.819064	21836136 DNF (51.42%)	-	-
sort_bs_13	2096128 33.436370	24745034 DNF (58.27%)	-	-
sort_fn_21	2096128 16.621842	19646692 DNF (46.26%)	-	-
sort_bn_21	2096128 0.063759	42462720 1.410025	3355402240 92.986209	379189198848 DNF (4.31%)
sort_fs_21	2096128 7.044232	34074112 DNF (80.24%)	-	-
sort_bs_21	2096128 7.741015	24520101 DNF (57.74%)	-	-
sort_fn_23	11264 0.000957	89088 0.007545	1048576 0.047835	46137344 4.078250
sort_bn_23	11264 0.000485	89088 0.007580	1048576 0.096192	46137344 3.908560
sort_fs_23	11264 0.070363	89088 1.929896	1048576 127.068670	39845888 DNF (12.49%)
sort_bs_23	11264 0.040125	89088 1.951079	1048576 179.269749	39845888 DNF (12.49%)
sort_fn_31	11264 0.045113	54272 0.728213	638976 36.921471	40335315 DNF (99.99%)
sort_bn_31	11264 0.028859	54272 0.610610	671744 57.709613	38707224 DNF (99.99%)
sort_fs_31	11264 17.079443	53246 DNF (99.99%)	-	-
sort_bs_31	11264 12.918621	54272 DNF (100.00%)	-	-
sort_fn_123	2096128 0.727380	42462720 14.202856	2983599840 DNF (88.91%)	-
sort_bn_123	2096128 0.547201	42462720 19.963312	3355402240 1715.459160	36469847951 DNF (0.41%)
sort_fs_123	2096128 38.459365	8146863 DNF (54.18%)	-	-

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_bs_123	2096128 38.515272	8247891 DNF (53.42%)	-	-
sort_fn_213	2096128 0.400090	42462720 9.270643	3355402240 763.578500	149457143169 DNF (1.69%)
sort_bn_213	2096128 0.306731	42462720 5.838961	3355402240 815.772535	188976463872 DNF (2.14%)
sort_fs_213	2096128 48.575900	18141068 DNF (42.72%)	-	-
sort_bs_213	2096128 28.572557	18400893 DNF (43.33%)	-	-
sort_fn_231	11264 0.070660	89088 1.447418	1048576 141.630244	40894464 DNF (18.74%)
sort_bn_231	18318 0.109571	125952 2.074139	1456356 181.735681	65040838 DNF (11.71%)
sort_fs_231	11264 21.648020	58368 DNF (80.24%)	-	-
sort_bs_231	18318 18.438339	125448 DNF (99.99%)	-	-

Table A.2: Number of swaps and time taken to sort a descending list

A.3 Unique Random

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_fn	1054964 0.037579	21338280 0.776394	1675399483 72.149437	1005772453768 DNF (61.42%)
sort_bn	1054964 0.044334	21338280 0.928341	1675399483 71.005161	970425301974 DNF (61.01%)
sort_fs	1054964 0.022789	21338280 0.495471	1675399483 26.522183	796374361848 DNF (59.04%)
sort_bs	1054964 0.027964	21338280 0.504116	1675399483 36.809139	818741542982 DNF (59.29%)
sort_fn_1	1054964 11.072979	10799537 DNF (75.18%)	-	-
sort_bn_1	1054964 0.029491	21338280 0.542997	1675399483 42.794181	379866655361 DNF (54.30%)
sort_fs_1	1054964 11.426693	18789208 DNF (93.99%)	-	-
sort_bs_1	1054964 9.197065	20353360 DNF (97.94%)	-	-
sort_fn_2	1054964 0.043702	21338280 1.307503	1675399483 86.311526	307006107057 DNF (53.47%)
sort_bn_2	1054964 0.044933	21338280 0.834031	1675399483 50.426982	298790137836 DNF (53.38%)
sort_fs_2	1054964 0.045691	21338280 0.839377	1675399483 46.305473	1139403045924 DNF (62.93%)
sort_bs_2	1054964 0.052360	21338280 1.223758	1675399483 48.321028	1220173970330 DNF (63.85%)
sort_fn_3	43106 0.001379	295112 0.008800	9847303 0.358485	4281634011 123.742384
sort_bn_3	43106 0.001025	295112 0.007636	9847303 0.206557	4281634011 121.511550
sort_fs_3	43106 0.058299	295112 1.209596	9847303 56.990061	2330401398 DNF (99.96%)
sort_bs_3	43106 0.052305	295112 0.885851	9847303 59.313737	2261728296 DNF (99.96%)
sort_fn_12	1054964 6.757971	14996573 DNF (85.06%)	-	-
sort_bn_12	1054964 0.161635	21338280 4.063889	1675399483 459.571673	35044207040 DNF (50.38%)
sort_fs_12	1054964 14.757941	8082989 DNF (68.78%)	-	-

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_bs_12	1054964 12.679747	15032841 DNF (85.15%)	-	-
sort_fn_13	1054964 0.237408	21338280 3.508180	1675399483 453.337624	95286791465 DNF (51.06%)
sort_bn_13	1054964 0.170148	21338280 2.982098	1675399483 237.109111	133535239653 DNF (51.50%)
sort_fs_13	1054964 29.847980	13788231 DNF (82.21%)	-	-
sort_bs_13	1054964 23.381217	15641949 DNF (86.58%)	-	-
sort_fn_21	1054964 7.480358	14623634 DNF (84.18%)	-	-
sort_bn_21	1054964 0.048244	21338280 1.061563	1675399483 78.188455	229861373492 DNF (52.59%)
sort_fs_21	1054964 12.486611	17302684 DNF (90.49%)	-	-
sort_bs_21	1054964 9.053450	16546489 DNF (88.71%)	-	-
sort_fn_23	33288 0.000992	229596 0.013247	2962969 0.089137	265046273 8.448423
sort_bn_23	33288 0.001722	229596 0.012340	2962969 0.168040	265046273 8.219742
sort_fs_23	33288 0.112819	229596 2.829468	2962969 151.631321	198042336 DNF (56.23%)
sort_bs_23	33288 0.107454	229596 2.574902	2962969 199.975693	203268762 DNF (57.80%)
sort_fn_31	43106 0.133046	295112 1.735021	9847303 493.636804	341125642 DNF (99.69%)
sort_bn_31	43360 0.029505	296198 0.756526	9857027 60.909006	1854969112 DNF (99.95%)
sort_fs_31	43106 21.146341	289766 DNF (99.98%)	-	-
sort_bs_31	43336 9.520690	293801 DNF (99.99%)	-	-
sort_fn_123	1054964 0.453979	21338280 13.722421	1379132178 DNF (91.17%)	-
sort_bn_123	1054964 0.523004	21338280 16.193078	1296323632 DNF (88.70%)	-
sort_fs_123	1051849 37.164145	10984460 DNF (75.57%)	-	-

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_bs_123	1054964 41.057299	10763606 DNF (75.09%)	-	-
sort_fn_213	1054964 0.374091	21338280 8.671061	1675399483 990.480544	78117242422 DNF (50.87%)
sort_bn_213	1054964 0.272413	21338280 5.828314	1675399483 620.937056	105038048436 DNF (51.18%)
sort_fs_213	1054964 50.820805	15614637 DNF (86.52%)	-	-
sort_bs_213	1054964 29.417080	13599203 DNF (81.77%)	-	-
sort_fn_231	33288 0.087002	229596 1.066666	2962969 86.849212	203572233 DNF (57.80%)
sort_bn_231	33482 0.115225	230564 1.645795	2971633 118.180251	195893733 DNF (55.84%)
sort_fs_231	33288 26.936398	225576 DNF (99.98%)	-	-
sort_bs_231	33712 17.854500	230511 DNF (99.99%)	-	-

Table A.3: Average number of swaps and time taken to sort a unique random list

A.4 Non-Unique Random

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_fn	1034783 0.044668	21234665 0.781311	1671845031 71.169897	1003399099168 DNF (61.40%)
sort_bn	1034783 0.040469	21234665 0.968708	1671845031 71.100322	966970688750 DNF (60.99%)
sort_fs	761032 0.025546	15542043 0.404514	1230404454 32.389005	732584908881 DNF (60.14%)
sort_bs	779491 0.019432	15583570 0.381352	1226676216 31.133989	757412607456 DNF (60.50%)
sort_fn_1	1034783 10.901621	10743611 DNF (75.28%)	-	-
sort_bn_1	1034783 0.030933	21234665 0.523034	1671845031 41.934964	379630159341 DNF (54.31%)
sort_fs_1	761032 12.143632	12701398 DNF (89.58%)	-	-
sort_bs_1	1034783 6.200382	13934263 DNF (82.79%)	-	-
sort_fn_2	1034783 0.043840	21234665 1.399293	1671845031 91.815000	307036630837 DNF (53.49%)
sort_bn_2	1034783 0.044901	21234665 0.827363	1671845031 56.328664	298878452054 DNF (53.39%)
sort_fs_2	883115 0.044816	17562284 1.133026	1373861893 46.671484	1070730395100 DNF (62.70%)
sort_bs_2	896871 0.052104	18783208 0.679015	1487830145 44.396378	1117440290067 DNF (63.29%)
sort_fn_3	52862 0.001563	309654 0.009239	10228177 0.379809	5284825711 142.383325
sort_bn_3	52862 0.001507	309654 0.006805	10228177 0.221572	5284825711 138.704259
sort_fs_3	47608 0.069478	298811 1.299423	9766708 76.703661	2108431882 DNF (99.95%)
sort_bs_3	47559 0.056543	301445 1.224074	9717359 72.933599	2126115670 DNF (99.95%)
sort_fn_12	1034783 6.521379	14978975 DNF (85.25%)	-	-
sort_bn_12	1034783 0.189432	21234665 4.329785	1671845031 436.157660	34715779258 DNF (50.39%)
sort_fs_12	762887 9.117285	11336221 DNF (84.74%)	-	-

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_bs_12	1034783 10.035396	14894531 DNF (85.05%)	-	-
sort_fn_13	885909 0.236295	18071554 3.498455	1419140475 308.131197	94868242600 DNF (51.10%)
sort_bn_13	885909 0.169525	18071554 2.326735	1419140475 254.917601	130471026607 DNF (51.53%)
sort_fs_13	846136 26.973508	8198103 DNF (72.08%)	-	-
sort_bs_13	885909 19.776540	11730802 DNF (81.40%)	-	-
sort_fn_21	1034783 7.269802	14602584 DNF (84.37%)	-	-
sort_bn_21	1034783 0.041798	21234665 1.157687	1671845031 85.235844	229736889971 DNF (52.61%)
sort_fs_21	883115 7.014793	13416896 DNF (86.07%)	-	-
sort_bs_21	1034783 6.065331	15322435 DNF (86.06%)	-	-
sort_fn_23	32222 0.000988	222185 0.013170	2905577 0.174299	262608852 8.484805
sort_bn_23	32222 0.000935	222185 0.012682	2905577 0.166913	262608852 8.104582
sort_fs_23	32222 0.108550	222184 2.443923	2905203 184.865100	197984502 DNF (56.25%)
sort_bs_23	32222 0.137392	222185 2.945188	2905255 219.765985	202856202 DNF (57.81%)
sort_fn_31	52862 0.109595	309654 1.289607	10228177 612.185646	340746972 DNF (99.69%)
sort_bn_31	53099 0.029736	310679 0.885239	10237671 59.558023	1714213012 DNF (99.94%)
sort_fs_31	47608 21.039848	292947 DNF (99.96%)	-	-
sort_bs_31	53075 9.488318	308618 DNF (99.98%)	-	-
sort_fn_123	882195 0.509126	18031946 24.185745	1006268286 DNF (84.46%)	-
sort_bn_123	882195 0.558230	18031946 14.217641	1490674116 DNF (94.41%)	-
sort_fs_123	849918 36.991328	6361137 DNF (66.64%)	-	-

Algorithm	**L0** Nr of Swaps Time (s) (% Sorted)	**L1** Nr of Swaps Time (s) (% Sorted)	**L2** Nr of Swaps Time (s) (% Sorted)	**L3** Nr of Swaps Time (s) (% Sorted)
sort_bs_123	882195 37.233967	8286825 DNF (71.90%)	-	-
sort_fn_213	948078 0.370813	19148552 7.343043	1500045636 726.164973	77796270137 DNF (50.88%)
sort_bn_213	948078 0.270903	19148552 5.282602	1500045636 442.249563	104516615356 DNF (51.19%)
sort_fs_213	937027 29.954984	9362481 DNF (73.23%)	-	-
sort_bs_213	948078 38.379727	15750372 DNF (90.02%)	-	-
sort_fn_231	32222 0.076760	222185 1.627745	2905577 83.576241	203171552 DNF (57.81%)
sort_bn_231	32424 0.116410	223113 1.632834	2914540 116.507698	195909913 DNF (55.86%)
sort_fs_231	32222 27.931792	220689 DNF (99.98%)	-	-
sort_bs_231	32652 14.488620	223661 DNF (99.98%)	-	-

Table A.4: Average number of swaps and time taken to sort a non-unique random list

A.5 Hybrid Algorithms

Data Set	Algorithm	**L0** Time (s)	**L1** Time (s)	**L2** Time (s)	**L3** Time (s)
Ascending	introsort	0.000378	0.001821	0.006975	0.273307
	qsort	0.000095	0.000719	0.007446	0.119889
	timsort	0.000013	0.000141	0.001297	0.015562
Descending	introsort	0.000348	0.001864	0.003841	0.224986
	qsort	0.000148	0.000773	0.006188	0.121322
	timsort	0.000042	0.000177	0.001530	0.018047
Unique Random	introsort	0.001010	0.005330	0.012233	0.771161
	qsort	0.000391	0.002636	0.010059	0.540909
	timsort	0.000869	0.004840	0.014157	0.834823
Non-Unique Random	introsort	0.001005	0.005260	0.012691	0.774225
	qsort	0.000575	0.002983	0.010398	0.538410
	timsort	0.000883	0.005046	0.012729	0.833785

Table A.5: Time taken to sort each input set using hybrid sorting algorithms