

A class of orders with linear? time sorting algorithm

Laurent Lyaudet

¹ <https://lyaudet.eu/laurent/>

received 2018/12/30, revised , accepted .

In this article, we give a precise mathematical meaning to “linear? time” that matches experimental behaviour of the algorithm. The sorting algorithm is not our own, it is a variant of radix sort with counting sort as a subroutine. The true result of this article is an efficient universality result for lexicographic order, or more generally for some linear extensions of the partial order “Next”: “if current items are equal, compare next items”. We define new classes of orders: (Finite width) Tree Structured Orders. We show that an instance of a finite width tree structured order can be converted in linear time and space to an instance of lexicographic order. The constants implied by the “nextification” algorithm are small (around 3 for real world orders). The class of finite width tree structured orders contains finite orders ($\{0, 1\}$, int32, int64, ..., float, double, ...), and orders constructed from them on a tree structure. In particular, unbounded integers, strings with arbitrary collation, and all orders used for sorting SQL queries are finite width tree structured orders.

Keywords: order, sort, lexicographic, string, integer

Current version : 2018/12/30

1 Introduction

The classical complexity result that is taught in a computer science curriculum is that sorting can be done in $\Theta(n \times \log(n))$. Of course, the word “comparisons” is purposely missing at the end of the previous sentence (and also the fact that n is the number of elements to sort), but it unfortunately reflects the frequently found informal content of the teachings and beliefs on sorting. It is a problem both for the $\Omega(n \times \log(n))$ and the $O(n \times \log(n))$ asymptotic bounds, as we shall see.

First, it is a problem for $\Omega(n \times \log(n))$. Indeed, if we consider a finite order of size k , the counting sort (see Seward (1954)) solves any instance of n elements of this order in time $O(n \times \log(k)) = O(n)$.

Second, it is also a problem for $O(n \times \log(n))$ since this is “comparison complexity” instead of time complexity. It makes the implicit assumption that comparison should have $O(1)$ time complexity for practical purposes. It does not take into account the size of the input ; but it is relatively to input size that we are supposed to measure time complexity. Let us denote m the size of an instance of sorting problem. There is no polynomial upper bound of m given n . This is a situation that is different from what occurs in algorithmic graph theory, for example, where the complexity of algorithms is given relatively to the number of vertices and edges of the graph. (The size of a reasonable encoding of a graph is upper bounded

by $O(n^2 \times \log(n))$, or $O((n+m) \times \log(n))$, if n is the number of vertices, and m is the number of edges.) However, comparison of long strings with a long common prefix would have $\Theta(\frac{m}{n})$ time complexity, assuming that all strings to be sorted have roughly the same size. If $M(p)$ denotes the time complexity of multiplication of integers of p bits, comparison of two unbounded rational numbers would have $\Theta(M(\frac{m}{n}))$ time complexity (assuming again that all rationals to be sorted have roughly the same size). (Currently the best asymptotic upper bound known for $M(p)$ is $p \times \log(p) \times 4^{\log^*(p)}$ (see Harvey and van der Hoeven (2018)).) These two practical cases would yield $\Theta(m \times \log(n))$, and $\Theta(m \times \log(\frac{m}{n}) \times 4^{\log^*(\frac{m}{n})} \times \log(n))$ time complexity respectively.

Worse yet, we could consider a pathological sort based on an undecidable problem like the halting problem. An element of the order would be an encoding of a Turing machine with an input for it, with the knowledge that this Turing machine does halt on this input, and the order would be based on the number of steps of computation needed for the Turing machine to halt. (We cannot construct this order but we can define it and thus we can define the associated sorting problem.) Sorting only two elements would have no computable upper bound on its time complexity, whilst comparison complexity is 1. In particular, the general sorting problem is not polynomial time solvable, and not even decidable. (It would be funny to ask students, computer engineer, and computer scientists to see how many of them would answer that sorting is polynomial time solvable if asked whether this is true.)

As we can see clearly with these last examples, $\Theta(n \times \log(n))$ comparison complexity result is here for a good reason: **We cannot say anything more precise and as general than that.** But we never explain too much why it is so.

This article tries its best to provide results that are much more precise but at the cost of being slightly less general. For achieving this goal, we make a paradigm change. Instead of considering the comparison “black box” model, we open the box and consider “common” orders as they are defined and constructed. It appears that all these common orders are defined using an expression, i.e. a labeled tree, that these common orders exclude pathological sorts, and that for them we have $\Theta^?(m)$ time and space complexity. We note that many researchers already partially opened the box to consider integers sorting and finite orders sorting, and as an aside lexicographic sorting (see for example Andersson et al. (1998) where they obtained $\Theta(n \times \log(\log(n)))$ bounds for finite orders, and $\Theta(m \times \log(\log(n)))$ bounds for string sorting). We think the results in our article give an order theory and logical addition to the literature on sorting that complements well these results.

The question mark on asymptotic notation is just a short-hand for some “conditional complexity”. In fact every complexity result is conditional ; when we write $\Theta(n \times \log(n))$ time complexity, we may want to say:

- assuming Turing machine model, $\Theta(n \times \log(n) \mid \textit{time Turing machine model})$,
- assuming RAM model, $\Theta(n \times \log(n) \mid \textit{time RAM model})$.

In this article, we use the question mark to say “assuming RAM model where pointers and size integers have constant size and arithmetic on pointers/sizes takes constant time”. We could have written something like $\Theta(m \mid \textit{time RAM, pointer/size(size : } O(1), + : O(1), - : O(1), \times : O(1)))$ instead. The question mark on asymptotic notations will have this meaning within this article, however we hope it may take other meanings in other articles where it can serve as a handy short-hand.

The justification for this conditional complexity is linked to current 64-bit processors ; with 8-bit processors we could not have considered that sorting 256 octets of data is all that we need. However, we do

consider that sorting 16 exbibytes is more than enough for current real world problems. With 128-bits processors we could roughly sort all the atoms of the earth...

A great success of the black box model is that the algorithms are generic, and you just need to provide a callback for the comparison function. Our results will enable similar reuse of any algorithm suited for doing lexicographic or variants of lexicographic sorting, and we aim at providing a library (see <https://github.com/LLyaudet/TSODLJLS>) for generating the code for or dynamically converting the keys of the elements to sort into a simple string. In order to achieve this goal, we will try to provide a language for describing orders.

Section 2 contains most of the definitions and notations used in this article. In Section 3, we build our first results using the definitions, so that the reader gets accustomed to them. Section 4 recalls a result of Cantor and proves an efficient version of it for a wide class of orders. In Section 5, we prove that finite width tree structured orders are countable orders. Section 6 looks at the frontiers of our results and open problems. Finally, Section 7 presents an attempt at a tree structured order definition language.

2 Definitions and notations

Throughout this article, we use the following definitions and notations. O denotes an order (it is a total/linear order), in particular $O^{0,1}$ denotes the binary order where $0 < 1$. We denote $\text{Domain}(O)$, the domain of the order O (for example, $\text{Domain}(O^{0,1}) = \{0, 1\}$). \mathcal{O}_i denotes a sequence of orders indexed by the ordinal i , $i = L(\mathcal{O}_i)$ is the length of \mathcal{O}_i , in particular $\mathcal{O}_\omega^{0,1} = (O^{0,1})_\omega$ denotes the sequence of binary orders repeated a countable number of times, ω is its length. It is not necessary to know what an ordinal is to understand 99% of this article. Later we define Tree Structured Orders using arbitrary ordinals, but the algorithmic results are for finite width tree structured orders. For these later orders we use only the finite ordinals (first, second, third, fourth, etc.), and the first limit ordinal ω as an upper bound (in order to express finite but not with a finite upper bound, it can be as big as we want so long it stays finite). Given two ordinals $i < j$, and a sequence of orders \mathcal{O}_j , we denote $\mathcal{O}_j[i] = O_i$, the item of rank i in the sequence (the ranks start at 0). The reader might know Von Neumann's construction of the ordinals (an ordinal can be seen as a set that contains exactly all ordinals that are strictly before it, the 0th ordinal is the empty set), in which case we can consider that $i \in j \Leftrightarrow i < j$. We also use this notation, for example in $\mathcal{O}_i = (O_j)_{j \in i}$. While ordinals are frequently denoted by greek letters, we will try to keep using i, j, k, l for this purpose, so that it recalls finite indices to the reader.

We denote $\text{Inv}(O)$, the inverse order of O ; for example, $\text{Inv}(O^{0,1}) = O^{1,0}$ is the order on 0 and 1 where $1 < 0$. We also denote $\text{Inv}(\mathcal{O}_i) = (\text{Inv}(O_j))_{j \in i}$, the sequence of inverse orders of \mathcal{O}_i ; for example, if $\mathcal{O}_3 = (O^{0,1}, O^{0,1}, O^{0,1,2})$, then $\text{Inv}(\mathcal{O}_3) = (O^{1,0}, O^{1,0}, O^{2,1,0})$ (each order in the sequence is inverted but the ranks of the items are preserved). Note that Inv is an involution: $\text{Inv}(\text{Inv}(O)) = O$, and $\text{Inv}(\text{Inv}(\mathcal{O}_i)) = \mathcal{O}_i$.

In the definitions below, we indicate “(new term)” when we used a new word for denoting a mathematical object that probably exists somewhere else in mathematical literature ; we indicate “(new)” when we defined a new mathematical object for which we found no anteriority in mathematical literature. If some reader might provide references for early use of these concepts, we would be happy to add them here.

Definition 2.1 (Prelude sequence (new term)). *Given two ordinals $i < j$, and two sequences of orders $\mathcal{O}_i, \mathcal{O}_j$, we say that \mathcal{O}_i is a prelude (sequence) of \mathcal{O}_j , if and only if $\mathcal{O}_i[k] = \mathcal{O}_j[k], \forall k \in i$ (we have $k < i < j$). We note $\text{Prelude}(\mathcal{O}_j)$ the set of all prelude sequences of the order sequence \mathcal{O}_j .*

For example, $(O^{0,1}, O^{0,1})$ is a prelude of $(O^{0,1}, O^{0,1}, O^{0,1,2})$; it is also a prelude of $\mathcal{O}_\omega^{0,1}$.

We say that X is an element of \mathcal{O}_i (denoted $X \in \mathcal{O}_i$), when $X = (x)_i$ is a sequence indexed by i with $x_k \in \text{Domain}(O_k) = \text{Domain}(\mathcal{O}_i[k]), \forall k \in i$. By convention, there is a unique sequence of orders of length 0 \mathcal{O}_0 ; it has only one element denoted ϵ (the empty word). \mathcal{O}_0 is a prelude sequence of any other order sequence. We try to avoid confusion by distinguishing *element* X of \mathcal{O}_i from *item* $\mathcal{O}_j[i]$ of \mathcal{O}_i ; for example, the “word” 002 is an element of $\mathcal{O}_3 = (O^{0,1}, O^{0,1}, O^{0,1,2})$, whilst $O^{0,1}$ is the first and second item of \mathcal{O}_3 .

Definition 2.2 (Compatible (sequences) (new)). *Given two sequences of orders $\mathcal{O}_i, \mathcal{O}_j$, we say that they are compatible if they are equal, or one is a prelude sequence of the other. Let $X \in \mathcal{O}_i, Y \in \mathcal{O}_j$, we say that X, Y are compatible if $\mathcal{O}_i, \mathcal{O}_j$ are compatible.*

As we shall see, *compatible* elements are easily converted into *comparable* elements. Indeed, both “element-items” at the same rank in the two elements may be compared, since they belong to the domain of the same “order-item”. The missing step for obtaining a total order is the object of several definitions below.

Definition 2.3 (Question (new term)). *Given two compatible elements X, Y of sequences of orders $\mathcal{O}_i, \mathcal{O}_j$, we say that (k, x_k, y_k) is the question of X, Y , if k is the smallest ordinal such that $x_k \neq y_k$. If $X \neq Y$, and neither X is a prefix of Y , nor Y is a prefix of X , such a k exists because ordinals are well-ordered. By contrapositive, if no such a k exists, then $X = Y$, or either X is a prefix of Y , or Y is a prefix of X . Thus if no such a k exists, then $X = Y$, or $L(X) \neq L(Y)$.*

Definition 2.4 (Order-sequence-operator (new)). *An order-sequence-operator is an operator that maps a set of compatible order sequences to an order.*

Lemma 2.5. *Let S be a set of compatible order sequences, then there exists an order sequence \mathcal{O}_j of which all order sequences of S are preludes.*

Proof :

By definition of compatible sequences, it is clear that for any ordinal i , there is at most one sequence of length i in S . Since S is a set, there exists an ordinal j such that all sequences of S have length less than j (because the class of all ordinals is not a set). Let j be the smallest such ordinal (because ordinals are well-ordered).

- If j is a successor ordinal, $\forall i \in j - 1, \forall \mathcal{O}_k \in S, \mathcal{O}'_{k'} \in S$, with $i < k, k' < j$, we have $\mathcal{O}_k[i] = \mathcal{O}'_{k'}[i] = O_i$. Moreover, by minimality of $j, \forall i \in j - 1, \mathcal{O}_k \in S, \mathcal{O}_k[i]$, and O_i exist. Let us denote \mathcal{O}_j , the sequence of all O_i padded with $O_{j-1}^{0,1}$. It is clear that \mathcal{O}_j satisfies that all order sequences of S are preludes of it.
- If j is a limit ordinal, $\forall i \in j, \forall \mathcal{O}_k \in S, \mathcal{O}'_{k'} \in S$, with $i < k, k' < j$, we have $\mathcal{O}_k[i] = \mathcal{O}'_{k'}[i] = O_i$. Moreover, by minimality of $j, \forall i \in j, \mathcal{O}_k \in S, \mathcal{O}_k[i]$, and O_i exist. Let us denote \mathcal{O}_j , the sequence of all O_i . It is clear again that \mathcal{O}_j satisfies that all order sequences of S are preludes of it.

■

Using these first lemma and definitions, we shall now define five order-sequence-operators. The notations may be surprising because instead of considering arbitrary sets of compatible order sequences, we

will consider all preludes of some order sequence for four of them. (The orders constructed with arbitrary sets would be sub-orders of the orders that we will define now.) The only step toward arbitrary sets of compatible order sequences will be a possible restriction on the minimum length of the sequences. This step will be justified by optimizations and other results later in this article.

We start by a “partial-order-sequence-operator”; its result is a partial order instead of a total order.

Definition 2.6 (Next partial order). *Given two ordinals $i < j$, and a sequence of orders $\mathcal{O}_j = (O_k)_{k \in j}$, the next partial order denoted $\text{Next}(i, j, \mathcal{O}_j)$ is a partial order defined on the set of all elements of the preludes of \mathcal{O}_j , such that these preludes have length at least i . This is the partial order satisfying $\forall X, Y \in \text{Domain}(\text{Next}(i, j, \mathcal{O}_j))$ (the lengths $L(X), L(Y)$ of X and Y are such that $i \leq L(X), L(Y) < j$),*

- if X, Y have a question (k, x_k, y_k) , then:
 - if $x_k < y_k$ in O_k , then $X < Y$,
 - if $x_k > y_k$ in O_k , then $X > Y$,
- if they don't have a question, then elements are not ordered.

The partial order $\text{Next}(i, j, \mathcal{O}_j)$ corresponds to the simple idea for comparing sequences “if current items are equal, compare next items”.

Definition 2.7 (Lexicographic order). *Given two ordinals $i < j$, and a sequence of orders $\mathcal{O}_j = (O_k)_{k \in j}$, the lexicographic order denoted $\text{Lex}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the preludes of \mathcal{O}_j , such that these preludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{Lex}(i, j, \mathcal{O}_j))$,*

- if X, Y have a question (k, x_k, y_k) , then:
 - if $x_k < y_k$ in O_k , then $X < Y$,
 - if $x_k > y_k$ in O_k , then $X > Y$,
- if they don't have a question, then either $X = Y$, or $L(X) \neq L(Y)$:
 - if $L(X) < L(Y)$, then $X < Y$,
 - if $L(X) > L(Y)$, then $X > Y$.

Example. $\text{Lex}(0, \omega, \mathcal{O}_\omega^{0,1})$ is the lexicographic order on all finite binary strings, including the empty word.

Lexicographic order may be found under many names (see wikipedia):

- lexicographical order,
- lexical order,
- dictionary order,
- alphabet order (– the term “alphabet order” should be reserved for the order on the letters, even if the lexicographic order is based on it recursively),

- lexicographic(al) product.

Definition 2.8 (Contre-Lexicographic order (new term)). *Given two ordinals $i < j$, and a sequence of orders $\mathcal{O}_j = (O_k)_{k \in j}$, the contre-lexicographic order denoted $\text{ContreLex}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the preludes of \mathcal{O}_j , such that these preludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{ContreLex}(i, j, \mathcal{O}_j))$,*

- if X, Y have a question (k, x_k, y_k) , then:
 - if $x_k < y_k$ in O_k , then $X < Y$,
 - if $x_k > y_k$ in O_k , then $X > Y$,
- if they don't have a question, then either $X = Y$, or $L(X) \neq L(Y)$:
 - if $L(X) < L(Y)$, then $X > Y$,
 - if $L(X) > L(Y)$, then $X < Y$.

Lexicographic and contre-lexicographic orders are the two simplest linear extensions of the partial order $\text{Next}(i, j, \mathcal{O}_j)$. With lexicographic order, if no difference is found in the items of the sequence, then the shortest sequence comes first. With contre-lexicographic order, if no difference is found in the items of the sequence, then the shortest sequence comes last.

Definition 2.9 (Hierarchic order). *Given two ordinals $i < j$, and a sequence of orders $\mathcal{O}_j = (O_k)_{k \in j}$, the hierarchic order denoted $\text{Hierar}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the preludes of \mathcal{O}_j , such that these preludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{Hierar}(i, j, \mathcal{O}_j))$,*

- if $L(X) < L(Y)$, then $X < Y$,
- if $L(X) > L(Y)$, then $X > Y$,
- otherwise $X = Y$, or X, Y have a question (k, x_k, y_k) , then:
 - if $x_k < y_k$ in O_k , then $X < Y$,
 - if $x_k > y_k$ in O_k , then $X > Y$.

Example. $\text{Hierar}(0, \omega, \mathcal{O}_\omega^{0,1})$ ($\text{Hierar}(0, \omega, \mathcal{O}_\omega^{0,1,2,3,4,5,6,7,8,9})$) is the hierarchic order on all finite binary strings (finite decimal strings respectively). If you keep only the string 0 and all strings with a leading non-zero digit, then it is the order on base 2 integers writing (base 10 integers writing respectively) that matches the order on the integers. Thus it is the order we use without naming it when we compare two written integers. The funny fact is that we need to compare the lengths of these strings first. These lengths are also integers but much smaller (logarithmic) and for “real life” integers our brain just reads the lengths as they are written in unary. We do not think that our brain applies recursive thinking to compare the length, however we can do so consciously for really long integers by counting the number of digits.

Hierarchic(al) order has also many names (some of them are listed on wikipedia shortlex). One can found it under the names :

- radix order,

- military order (it is the same idea that hierarchical order ; it could be also political order ; guess who deserves an empty word ? ;) ,
- pseudo-lexicographic order,
- short(-)lex(icographic) order,
- length-lexicographic order,
- genealogical order,
- lexicographic (– We found a book where the author used only one order throughout and he kept the only name familiar to him).

Words in hierarchic order may be found in dictionaries suited for crosswords and other word games.

Definition 2.10 (Contre-Hierarchic order (new term)). *Given two ordinals $i < j$, and a sequence of orders $\mathcal{O}_j = (O_k)_{k \in j}$, the contre-hierarchic order denoted $\text{ContreHierar}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the preludes of \mathcal{O}_j , such that these preludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{ContreHierar}(i, j, \mathcal{O}_j))$,*

- if $L(X) < L(Y)$, then $X > Y$,
- if $L(X) > L(Y)$, then $X < Y$,
- otherwise $X = Y$, or X, Y have a question (k, x_k, y_k) , then:
 - if $x_k < y_k$ in O_k , then $X < Y$,
 - if $x_k > y_k$ in O_k , then $X > Y$.

As with ContreLex, for ContreHierar, shortest elements comes last.

Definition 2.11 (Generalized sum of orders (new term)). *Given a “master order” O_m , and an injective mapping f that associates an order $f(x)$ to each element x of O_m , the generalized sum denoted $\sum_{x \in O_m} f(x)$ is an order defined on the disjoint union of all elements of the $f(x)$ orders, or with redundant information on a subset of the cartesian product between O_m and the disjoint union of all elements of the $f(x)$ orders. Thus an element of this order may be ordered as a couple $X = (x, y \in f(x))$. For two elements $X = (x, y \in f(x))$ and $X' = (x', y' \in f(x'))$, we have $X < X'$ if and only if $x < x'$, or $x = x'$ and $y < y'$ in the order $f(x)$.*

We call this operator a generalized sum because when $O_m = O^{0,1}$, $\sum_{x \in O_m} f(x)$ is the order sum $f(0) + f(1)$. The order sum has also many names: linear sum, ordinal sum, series composition, star product. If $f(x) = \text{copy}(O)$, $\forall x \in O_m$, we obtain the order product (linear product, ordinal product) $O \times O_m$ (this product is O repeated in O_m , each element of O_m is replaced by a copy of O).

The generalized sum does not look like the other *order-sequence-operators*. However, applying Zorn’s axiom to the domain of O_m , we can well-order the elements x of O_m , and we can replace the mapping f by a sequence \mathcal{O}_j indexed by an ordinal, and thus we can denote $\sum_{x \in O_m} f(x) = \sum(O_m, \mathcal{O}_j)$ where it is implied that O_m and \mathcal{O}_j are correlated by an implicit well-ordering of O_m . Since there is no efficient way to apply Zorn’s axiom, we shall put severe restrictions on O_m for efficient results later on.

Note that, whilst hierarchic order enables to order unsigned integers of arbitrary precision, it is easy with generalized sum to obtain signed integers of arbitrary precision.

Indeed it is a suborder of $\sum(O^{0,1}, (\text{Inv}(\text{Hierar}(1, \omega, \mathcal{O}^{0,1})), \text{Hierar}(1, \omega, \mathcal{O}^{0,1})))$.

3 First results on orders

Let us compare the orders we can construct so far with the building blocks of the previous section.

$\text{Lex}(0, 3, \mathcal{O}_3^{0,1})$ = $\text{Inv}(\text{ContreLex}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1})))$	$\text{ContreLex}(0, 3, \mathcal{O}_3^{0,1})$ = $\text{Inv}(\text{Lex}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1})))$	$\text{Lex}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1}))$ = $\text{Inv}(\text{ContreLex}(0, 3, \mathcal{O}_3^{0,1}))$	$\text{ContreLex}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1}))$ = $\text{Inv}(\text{Lex}(0, 3, \mathcal{O}_3^{0,1}))$
ϵ	00	ϵ	11
0	01	1	10
00	0	11	1
01	10	10	01
1	11	0	00
10	1	01	0
11	ϵ	00	ϵ

$\text{Hierar}(0, 3, \mathcal{O}_3^{0,1})$ = $\text{Inv}(\text{ContreHierar}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1})))$	$\text{ContreHierar}(0, 3, \mathcal{O}_3^{0,1})$ = $\text{Inv}(\text{Hierar}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1})))$	$\text{Hierar}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1}))$ = $\text{Inv}(\text{ContreHierar}(0, 3, \mathcal{O}_3^{0,1}))$	$\text{ContreHierar}(0, 3, \text{Inv}(\mathcal{O}_3^{0,1}))$ = $\text{Inv}(\text{Hierar}(0, 3, \mathcal{O}_3^{0,1}))$
ϵ	00	ϵ	11
0	01	1	10
1	10	0	01
00	11	11	00
01	0	10	1
10	1	01	0
11	ϵ	00	ϵ

The equalities in the header of the two tables are not dependent of the ordinals or the binary order, and our first results will be to prove them in their full generality.

Remark 3.1. Given two ordinals $i < j$, a sequence of orders \mathcal{O}_j , and two order-sequence-operators μ, ν in $\{\text{Lex}, \text{ContreLex}, \text{Hierar}, \text{ContreHierar}\}$, $\text{Domain}(\mu(i, j, \mathcal{O}_j)) = \text{Domain}(\mu(i, j, \text{Inv}(\mathcal{O}_j))) = \text{Domain}(\nu(i, j, \mathcal{O}_j)) = \text{Domain}(\nu(i, j, \text{Inv}(\mathcal{O}_j)))$.

Proposition 3.2. Given two ordinals $i < j$, and a sequence of orders \mathcal{O}_j , $\text{ContreLex}(i, j, \mathcal{O}_j) = \text{Inv}(\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j)))$.

Proof :

Let $X, Y \in \text{Domain}(\text{ContreLex}(i, j, \mathcal{O}_j)) = \text{Domain}(\text{Inv}(\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j))))$ No order-sequence-operator changes equality. Hence we need only to prove that $X < Y$ for $\text{ContreLex}(i, j, \mathcal{O}_j)$ if and only if $X < Y$ for $\text{Inv}(\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j)))$.

Suppose $X < Y$ for $\text{ContreLex}(i, j, \mathcal{O}_j)$.

- Either X, Y have a question (k, x_k, y_k) , in which case $x_k < y_k$ in \mathcal{O}_k . Hence $x_k > y_k$ in $\text{Inv}(\mathcal{O}_k)$, and $X > Y$ in $\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j))$. Thus $X < Y$ in $\text{Inv}(\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j)))$.
- Or Y is a prefix of X . Hence $X > Y$ in $\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j))$. Thus $X < Y$ in $\text{Inv}(\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j)))$.

Suppose $X < Y$ for $\text{Inv}(\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j)))$.

- Either X, Y have a question (k, x_k, y_k) , in which case $x_k < y_k$ in $\text{Inv}(\text{Inv}(O_k)) = O_k$. Thus $X < Y$ in $\text{ContreLex}(i, j, \mathcal{O}_j)$.
- Or Y is a prefix of X . Thus $X < Y$ in $\text{ContreLex}(i, j, \mathcal{O}_j)$.
- Or X is a prefix of Y . Hence $X < Y$ in $\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j))$. Thus $X > Y$ in $\text{Inv}(\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j)))$. A contradiction so this case: X is a prefix of Y , was not possible.

■

Corollary 3.3. *Given two ordinals $i < j$, and a sequence of orders \mathcal{O}_j , $\text{Lex}(i, j, \mathcal{O}_j) = \text{Inv}(\text{ContreLex}(i, j, \text{Inv}(\mathcal{O}_j)))$, $\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j)) = \text{Inv}(\text{ContreLex}(i, j, \mathcal{O}_j))$, and $\text{Inv}(\text{Lex}(i, j, \mathcal{O}_j)) = \text{ContreLex}(i, j, \text{Inv}(\mathcal{O}_j))$.*

Proposition 3.4. *Given two ordinals $i < j$, and a sequence of orders \mathcal{O}_j , $\text{ContreHierar}(i, j, \mathcal{O}_j) = \text{Inv}(\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j)))$.*

Proof :

Let $X, Y \in \text{Domain}(\text{ContreHierar}(i, j, \mathcal{O}_j)) = \text{Domain}(\text{Inv}(\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j))))$ No order-sequence-operator changes equality. Hence we need only to prove that $X < Y$ for $\text{ContreHierar}(i, j, \mathcal{O}_j)$ if and only if $X < Y$ for $\text{Inv}(\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j)))$.

Suppose $X < Y$ for $\text{ContreHierar}(i, j, \mathcal{O}_j)$.

- Either $L(Y) < L(X)$. Hence $X > Y$ in $\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j))$. Thus $X < Y$ in $\text{Inv}(\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j)))$.
- Or X, Y have a question (k, x_k, y_k) , in which case $x_k < y_k$ in O_k . Hence $x_k > y_k$ in $\text{Inv}(O_k)$, and $X > Y$ in $\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j))$. Thus $X < Y$ in $\text{Inv}(\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j)))$.

Suppose $X < Y$ for $\text{Inv}(\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j)))$.

- Either $L(Y) < L(X)$. Thus $X < Y$ in $\text{ContreHierar}(i, j, \mathcal{O}_j)$.
- Or $L(X) < L(Y)$. Hence $X < Y$ in $\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j))$. Thus $X > Y$ in $\text{Inv}(\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j)))$. A contradiction so this case: $L(X) < L(Y)$, was not possible.
- Or X, Y have a question (k, x_k, y_k) , in which case $x_k < y_k$ in $\text{Inv}(\text{Inv}(O_k)) = O_k$. Thus $X < Y$ in $\text{ContreHierar}(i, j, \mathcal{O}_j)$.

■

Corollary 3.5. *Given two ordinals $i < j$, and a sequence of orders \mathcal{O}_j , $\text{Hierar}(i, j, \mathcal{O}_j) = \text{Inv}(\text{ContreHierar}(i, j, \text{Inv}(\mathcal{O}_j)))$, $\text{Hierar}(i, j, \text{Inv}(\mathcal{O}_j)) = \text{Inv}(\text{ContreHierar}(i, j, \mathcal{O}_j))$, and $\text{Inv}(\text{Hierar}(i, j, \mathcal{O}_j)) = \text{ContreHierar}(i, j, \text{Inv}(\mathcal{O}_j))$.*

In light of these results, it appears that we could define another involution: Contre that operates on order-sequence-operators, $\text{Contre}(\text{operator}(\cdot)) = \text{Inv}(\text{operator}(\text{Inv}(\cdot)))$. Clearly $\text{Contre}(\text{Contre}(\text{operator}(\cdot))) = \text{operator}(\cdot)$. Thus we have $\text{Lex} = \text{Contre}(\text{ContreLex})$, $\text{ContreLex} = \text{Contre}(\text{Lex})$, $\text{Hierar} = \text{Contre}(\text{ContreHierar})$, $\text{ContreHierar} = \text{Contre}(\text{Hierar})$.

In later results, both operators Lex and ContreLex will be needed for labelling the internal nodes of the trees defining orders, if we want to exclude Inv . We could have also given names to $\text{Lex}(i, j, \text{Inv}(\mathcal{O}_j))$, and $\text{Inv}(\text{Lex}(i, j, \mathcal{O}_j))$ and use these two operators instead.

We shall now prove that $\sum = \text{Contre}(\sum)$, the generalized sum of orders is its own contre -order.

Proposition 3.6. *Given a “master order” O_m , and a corresponding sequence of orders \mathcal{O}_j ,*

$$\text{Inv}(\sum(O_m, \mathcal{O}_j)) = \sum(\text{Inv}(O_m), \text{Inv}(\mathcal{O}_j)).$$

Proof :

Clearly $\text{Domain}(\text{Inv}(\sum(O_m, \mathcal{O}_j))) = \text{Domain}(\sum(\text{Inv}(O_m), \text{Inv}(\mathcal{O}_j)))$ since it is the disjoint union of the orders \mathcal{O}_j , or $\text{Inv}(\mathcal{O}_j)$ equivalently. Let $X = (x, y)$, $X' = (x', y')$ be two elements of this domain.

- If $x < x'$ in O_m , then:
 - $X < X'$ in $\sum(O_m, \mathcal{O}_j)$, and $X' < X$ in $\text{Inv}(\sum(O_m, \mathcal{O}_j))$,
 - $x' < x$ in $\text{Inv}(O_m)$, and again $X' < X$ but in $\sum(\text{Inv}(O_m), \text{Inv}(\mathcal{O}_j))$.
- If $x' < x$ in O_m , then, by symmetry with the first case, $X < X'$ both in $\text{Inv}(\sum(O_m, \mathcal{O}_j))$ and in $\sum(\text{Inv}(O_m), \text{Inv}(\mathcal{O}_j))$.
- If $x = x'$ in O_m , then there exists some $O_i \in \mathcal{O}_j$ of which y and y' are elements.
 - If $y < y'$ in O_i , $X < X'$ in $\sum(O_m, \mathcal{O}_j)$, and $X' < X$ in $\text{Inv}(\sum(O_m, \mathcal{O}_j))$. And we have also that $y' < y$ in $\text{Inv}(O_i)$, thus $X' < X$ but in $\sum(\text{Inv}(O_m), \text{Inv}(\mathcal{O}_j))$.
 - If $y' < y$, by symmetry with the preceding subcase $X < X'$ both in $\text{Inv}(\sum(O_m, \mathcal{O}_j))$ and in $\sum(\text{Inv}(O_m), \text{Inv}(\mathcal{O}_j))$.

■

4 Universality of lexicographic order

4.1 Theoretical universality

Definition 4.1 (Universal order). *We say that an order O is universal for a class of orders \mathcal{A} , if for any order $O' \in \mathcal{A}$, there exists an embedding of O' in O . An order embedding is an injective mapping such that order is preserved.*

This is a classical result by Cantor (1895) that the order on the rationals \mathbb{Q} (between 0 and 1) is universal for countable orders. It is easy to see that the same applies for the lexicographic order on binary words. We adapt the proof to lexicographic order and give it here for the sake of completeness.

Theorem 4.2 (Cantor 1895). *$\text{Lex}(1, \omega, \mathcal{O}_\omega^{0,1})$ is universal for countable orders.*

Proof :

Let O be a countable order and $(x_i)_{i \in \mathbb{N}}$ be an enumeration of its elements. We associate to each element inductively a rational number between 0 and 1 for which the denominator is a power of two. Since these numbers are strictly between 0 and 1, there is a bijection between these numbers and their decimal part, which can be written as a word on the digits 0,1. Since the denominator is a power of two, the length is finite. Since the numbers are strictly positive, the length is at least one. The last digit is a 1.

The first element is associated to $\frac{1}{2}$, i.e. the word “1”. For each element after the first, assume by induction, that all previous elements were embedded in $\text{Lex}(1, \omega, \mathcal{O}_\omega^{0,1})$.

If the current element is the smallest of all elements considered so far, then assign to it the number $\frac{n}{2}$, where n is the rational number associated to the smallest of all previous elements. Since n has a denominator that is a power of two, the same is true for $\frac{n}{2}$.

If the current element is the biggest of all elements considered so far, then assign to it the number $\frac{1+n}{2}$, where n is the rational number associated to the biggest of all previous elements. Since n has a denominator that is a power of two, the same is true for $\frac{1+n}{2}$.

If the current element is between two elements considered so far, then assign to it the number $\frac{n+m}{2}$, where n, m are the rational numbers associated to these two elements. Since n, m have a denominator that is a power of two, the same is true for $\frac{n+m}{2}$. ■

Even the pathological example of the introduction (sorting Turing machines by halting number) fits in this theorem. However, it is not efficient. Nobody will provide us with an enumeration of some arbitrarily chosen countable order and an oracle to know where to insert the new element. Our goal in this article is to prove that for a wide class of orders, the lexicographic order (or any one of its three variants) is efficiently universal.

Before we do so, we remark that hierarchic order of a sequence of finite orders cannot be universal. Indeed, there is a finite number of elements of length at most k . Thus, there is no infinite descending chain in this order. Since it is a total order (or linear order), there is no infinite anti-chain (all chains have size 1), hierarchic order is well-ordered ; so it cannot embeds arbitrary countable orders. The same reasoning applies to contre-hierarchic order since it doesn't contain infinite ascending chain.

The generalized sum of order cannot produce infinite orders from finite orders so we will not try to use it for universality results.

4.2 Efficient universality

Let us now define the class of orders for which we will have efficient embeddings in lexicographic order over binary strings.

Definition 4.3 (Tree structured orders). *Tree structured orders are constructed recursively as follow:*

- *Finite orders are tree structured orders.*
- *Given a tree structured order O , $\text{Inv}(O)$ is a tree structured order.*
- *Given two ordinals $i < j$, and a sequence of tree structured orders \mathcal{O}_j , $\text{Lex}(i, j, \mathcal{O}_j)$ is a tree structured order.*
- *Given two ordinals $i < j$, and a sequence of tree structured orders \mathcal{O}_j , $\text{Hierar}(i, j, \mathcal{O}_j)$ is a tree structured order.*
- *Given a master tree structured order O_m , and a corresponding sequence of tree structured orders \mathcal{O}_j , $\sum(O_m, \mathcal{O}_j)$ is a tree structured order.*

We also have that (by propositions 3.2, and 3.4):

- *Given two ordinals $i < j$, and a sequence of tree structured orders \mathcal{O}_j , $\text{ContreLex}(i, j, \mathcal{O}_j)$ is a tree structured order.*

- Given two ordinals $i < j$, and a sequence of tree structured orders \mathcal{O}_j , $\text{ContreHierar}(i, j, \mathcal{O}_j)$ is a tree structured order.

Thus any tree structured order is defined by a labelled tree with possibly infinite depth and infinite width. Note that any path from the root of the tree to a leaf is finite, but because we have infinite sequences of orders, the depth of the tree may be infinite. The leaves of this labelled tree are finite orders. The internal nodes are labelled with two ordinals and an order-sequence-operator. In order to have efficient universality, we have to restrict ourself to more finitely described orders. We cannot consider arbitrary sequences of orders. Instead we will consider ultimately periodic sequences ; the period will be written between [], for example $\mathcal{O}_\omega = (O^{0,1}, [O^{0,1}, O^{1,0}])_\omega$ is the sequence where the first item and all items of even rank are the binary order, whilst all items of odd rank greater than one are the inverse of the binary order (we considered that rank started at 1 for this explanation).

When the sequence is periodic with a period of length 1, like $\mathcal{O}_\omega^{0,1} = ([O^{0,1}])_\omega$, we say that the sequence is uniform.

Definition 4.4 (Finite width tree structured orders). Finite width tree structured orders are constructed recursively as follow:

- Finite orders are finite width tree structured orders.
- Given a finite width tree structured order O , $\text{Inv}(O)$ is a finite width tree structured order.
- Given two ordinals $i < j \leq \omega$, and an ultimately periodic sequence of finite width tree structured orders \mathcal{O}_j , $\text{Lex}(i, j, \mathcal{O}_j)$ is a finite width tree structured order.
- Given two ordinals $i < j \leq \omega$, and an ultimately periodic sequence of finite width tree structured orders \mathcal{O}_j , $\text{Hierar}(i, j, \mathcal{O}_j)$ is a finite width tree structured order.
- Given a master finite order O_m , and a corresponding sequence of finite width tree structured orders \mathcal{O}_j , $\sum(O_m, \mathcal{O}_j)$ is a finite width tree structured order.

We also have that (by propositions 3.2, and 3.4):

- Given two ordinals $i < j \leq \omega$, and an ultimately periodic sequence of finite width tree structured orders \mathcal{O}_j , $\text{ContreLex}(i, j, \mathcal{O}_j)$ is a finite width tree structured order.
- Given two ordinals $i < j \leq \omega$, and an ultimately periodic sequence of finite width tree structured orders \mathcal{O}_j , $\text{ContreHierar}(i, j, \mathcal{O}_j)$ is a finite width tree structured order.

With these restrictions and using [] notation on sub-expressions or sub-trees, finite width tree structured orders can be represented by a finite expression and/or a finite labelled tree.

We used Inv , Lex , Hierar , \sum to define (finite width) tree structured order, however, by propositions and corollaries 3.2, 3.3, 3.4, 3.5, 3.6 we could have chosen the following sets of operators instead:

- Inv , ContreLex , Hierar , \sum ,
- Inv , Lex , ContreHierar , \sum ,
- Inv , ContreLex , ContreHierar , \sum ,

- Lex, ContreLex, Hierar, ContreHierar, Σ .

It is trivial to see except maybe for the last one. For Lex, ContreLex, Hierar, ContreHierar, Σ we have to remark that we can “push” all Inv nodes from the root of the tree to its leaves. Indeed we can exchange an internal node labelled with an order-sequence-operator together with its parent Inv node, for an internal node with the “contre” order-sequence-operator and children Inv nodes between the original children of the order-sequence-operator node. Since the depth of the tree is finite, we end up with Inv nodes only above leaves, where they can disappear since the inverse of a finite order is a finite order.

We continue with a remark where almost every order-sequence-operator considered so far is “universal”.

Remark 4.5 (Fundamental remark on orders). *For fixed length sequences of orders, there is no difference at all between (contre-)lexicographic, and (contre-)hierarchic order. For any ordinal i , and any order sequence \mathcal{O}_{i+1} , $\text{Lex}(i, i + 1, \mathcal{O}_{i+1}) = \text{ContreLex}(i, i + 1, \mathcal{O}_{i+1}) = \text{Hierar}(i, i + 1, \mathcal{O}_{i+1}) = \text{ContreHierar}(i, i + 1, \mathcal{O}_{i+1}) = \text{Next}(i, i + 1, \mathcal{O}_{i+1})$.*

Whilst we never saw this remark explicitly stated, it is used very frequently, and is probably the source of the confusion between lexicographic and hierarchic order. It renders useless to name the operator, for example in SQL queries: **SELECT User.* FROM User ORDER BY User.login, User.birthdate, User.firstname, User.lastname** ; a few commas are sufficient. It explains that radix sort that was created for fixed length integers should in reality be considered a string sorting algorithm, for strings of arbitrary size.

What we obtain with fixed length, we will try to achieve for arbitrary length. It will require that the length itself, or more precisely the end of the string, is explicitly encoded in the string so that everything is sorted *before* we actually reach the end of the string. It means that we will not only prove that lexicographic order is efficiently universal, but also that contre-lexicographic order and “next” partial order are efficiently universal. A “mirror work” could be done with $\text{Inv}(\text{Next}), \text{Inv}(\text{Lex}), \text{Inv}(\text{ContreLex})$.

From the fundamental remark on orders, we can deduce our encoding of finite orders. We enumerate the elements in the finite order from the smallest to the greatest and we use fixed length integer to do so. Thus finite orders are embedded in $\text{Next}(i, i + 1, \mathcal{O}_{i+1}^{0,1})$, for some integer i . (We must remark that we assume that the used finite orders are not “obfuscated”. Indeed, for example, the order $\mathcal{O}^{0,1}$ could be obfuscated behind the halting problem as in the pathological example of the introduction. The elements to be sorted would be Turing machine encodings with an input for them and the key of the sort is 0 if the Turing machine halts on its input, 1 otherwise. We assume that the keys for sorting finite orders are given explicitly or can be computed in constant time from the element to be sorted. Any efficient result has to make this assumption.)

What would happen if we had an Inv node? If the order below has been “binarily nextified”, then all we need to do is switch 0s and 1s so that the new strings are ordered correctly by “(contre-)lex/next”.

Let us look at other internal nodes.

If we have a Lex node, and everything below has been binarily nextified, then we almost only have to concatenate the strings. Almost because we want to have an indicator that the obtained string is finished. Maybe we could decrement some counter placed at the end of string. But for that this counter must be initialized to some value that is greater than the number of Lex node from a leaf to the tree.

The ContreLex node will also require a counter that we could increment.

Definition 4.6 (TSO-padding (new)). *Given a finite width tree structured order with tree/expression T , let l be an integer such that 2^l is greater than the maximum number of Lex nodes on a path from a leaf of T*

to its root; let c be an integer such that 2^c is greater than the maximum number of ContreLex nodes on a path from a leaf of T to its root. A l -lex-padding is a set of l contiguous bits that will serve as a counter for Lex nodes. A c -contrelex-padding is a set of c contiguous bits that will serve as a counter for ContreLex nodes. A (l, c) -TSO-padding is a l -lex-padding and a c -contrelex-padding that are contiguous.

For the rest of this subsection, it will be clearer to the reader if she assumes that the tree T has depth less than 15, which is more than enough for “common” orders. So that we can have TSO-paddings that are one octet where the first four bits are the lex-padding and the last four bits are the contrelex-padding. Let us assume in the following that a byte = an octet is the smallest allocatable space. The default value for TSO-paddings is $(2^l - 1, 0)$ (assume it is $(15, 0)$).

We can encode a finite order by adding after each byte of the original encoding a TSO-padding byte. However, how do we encode an empty string at the level of a Lex node? If we encode an empty string with a single TSO-padding byte, we have a shift between this TSO-padding byte at rank 0 that should be compared with TSO-padding bytes at odd rank. There are two simple possibilities:

- We can padd along the repeated pattern (padding0, data1, padding2). Thus the empty string in a Lex node (or ContreLex node) would be matched to a pattern with padding0 = (0, 0), data1 = 0, padding2 = (15,0). It would be slightly unefficient since padding0 bytes would serve only to deal with empty string, but still it would be linear space with a constant of 3.
- We can consider that there is no empty string and that all strings should be “null started”. Hence we use the repeated pattern (data0, padding1). All non-empty strings start with a pattern data0 = 0, padding1 = (15,0) The empty string is coded by the pattern data0 = 0, padding1 = (0,0). The problem is: What happens if an empty string is the last item to be concatenated in an above Lex node (or ContreLex node)? If we decrement the final lex-padding, since it is 0, we will have a problem. Another attempt where the empty string would be coded by data0 = anything and padding1 = (d,0), where d is the depth of the node, would fail to distinguish an empty string at depth d from an empty string at depth $d+1$, for which the lex-counter was decremented because it is the only item at depth d . We need to encode the depth of empty strings but we cannot decrement/increment this information.

These two simple possibilities show something important: making the padding right may be technical and open to traps, so in what follows we will use the repeated pattern (padding0, data1, padding2) for its robustness and simplicity. However for some finite width tree structured order, more efficient paddings can be possible. (When there is no path between the root and a leaf with more than one Lex or ContreLex node, or when all such nodes have strictly positive minimum length of preludes, we can use the pattern (data0, padding1). When we have only a finite order, we don’t need padding. When all the leaves correspond to finite orders encoded using exactly k bytes, then we can use the repeated pattern (padding0, data1, padding2) where data1 is k bytes, instead of only one byte. Etc.) It is one of our goal (out of scope for this article) to code a library that does the right choices for padding according to each particular tree/expression given to describe an order.

Definition 4.7 (TSO-encoding (new)). *We say that an element of an order is TSO-encoded if it is represented by a binary string that contains TSO-padding along the repeated pattern (padding0, data1, padding2), so that binary Next partial order applied to it and to another TSO-encoded string for the same order yields the same result than the original order. We may say that this binary string is the TSO-encoding of the element.*

Given a finite order encoded by integers with k bytes, we obtain in linear time and space a TSO-encoded string with k times the pattern (padding0 = (15, 0), data1, padding2 = (15,0)), and we decrement the final padding2 to (14,0).

Assume we obtained a TSO-encoding for all children suborders. Assume also for the length of this paragraph that we did remove the Inv nodes as we explained after defining finite width tree structured orders.

- If we have a Lex node above, we can concatenate in linear time and space the TSO-encodings of the suborders and decrement the lex-value of the final padding2. If the length of the prelude for this Lex node is zero, we can create in constant time and space a pattern padding0 = (d, 0), data1 = 0, padding2 = (15, 0), where d is the depth of the Lex node in the tree (it should be 0 if it is the root node). Let X, Y be elements of the order obtained at this node, and $E(X), E(Y)$ be their obtained encoding. If X, Y have a question (k, x_k, y_k) , then all subelements before the question are equal and thus $E(X), E(Y)$ starts with a common prefix (that may be empty) before we compare the patterns encoding x_k, y_k . But we assumed these patterns were TSO-encodings, hence comparing these patterns yields the same result as comparing x_k, y_k . If they don't have a question and assuming they are different, we can suppose, without loss of generality, that X is a prefix of Y ; if X is the empty string, then the starting padding0 of $E(X)$ has lex-value lower than the corresponding padding0 of $E(Y)$ (even if $E(Y)$ started with the encoding of an empty string from a subnode); if X is not the empty string, then the final padding2 of $E(X)$ has lex-value lower than the corresponding padding2 of $E(Y)$ and bytes before it are equal in $E(X)$ and $E(Y)$; hence comparing these patterns yields $X < Y$.
- If we have a ContreLex node above, we can concatenate in linear time and space the TSO-encodings of the suborders and increment the contrelex-value of the final padding2. If the length of the prelude for this ContreLex node is zero, we can create in constant time and space a pattern padding0 = (15, 15 - d), data1 = 0, padding2 = (15, 0), where d is the depth of the Lex node in the tree (it should be 0 if it is the root node). Let X, Y be elements of the order obtained at this node, and $E(X), E(Y)$ be their obtained encoding. If X, Y have a question (k, x_k, y_k) , then all subelements before the question are equal and thus $E(X), E(Y)$ starts with a common prefix (that may be empty) before we compare the patterns encoding x_k, y_k . But we assumed these patterns were TSO-encodings, hence comparing these patterns yields the same result as comparing x_k, y_k . If they don't have a question and assuming they are different, we can suppose, without loss of generality, that X is a prefix of Y ; if X is the empty string, then the starting padding0 of $E(X)$ has lex-value and contrelex-value greater than the corresponding padding0 of $E(Y)$; if X is not the empty string, then the final padding2 of $E(X)$ has contrelex-value greater than the corresponding padding2 of $E(Y)$ and bytes before it are equal in $E(X)$ and $E(Y)$; hence comparing these patterns yields $X < Y$.
- If we have a Hierar node above, we can encode the number of subelements as follow: we encode in unary the number of bytes needed to express the byte length of the number of subelements, then we concatenate the bytes encoding the byte length of the number of subelements, and finally we concatenate the binary encoding of the number of subelements. (Let us give an example, if the number of subelements is 2^{400} , the byte length of the number of subelements is $\lceil 401/8 \rceil = 51$, the number of bytes needed to encode this length is 1. So we encode 10000000 in unary for the

first byte, then 00110011 = 51 for the following byte, then the 51 bytes representing 2^{400} . (It is important that the unary encoding is terminated by a 0. Thus, if the number of bytes needed to encode the byte length of the number of subelements was 8, then the unary encoding would be 11111111 00000000. It would require that the number of subelements is greater than $2^{8 \times (2^{56} - 1)} - 1$, which is way more than the number of atoms in the universe.) The needed number of bytes is in $O(\log(n) + \log(\log(n)))$ where n is the number of subelements.

All these number of subelements bytes are padded along the common pattern (padding0, data1, padding2) as we would for finite orders. Then we concatenate the TSO-encodings of the subelements. It is trivial to see that if the numbers of subelements are different then the corresponding patterns will be ordered accordingly, otherwise the concatenations of TSO-encodings of the subelements will be ordered correctly. Note that the empty string with 0 subelement is already handled by the number of elements encoding (10000000 10000000 00000000), but we could simplify with a single null byte. Note also that we do not have to increment/decrement anything.

- If we have a ContreHierar node above, similarly to Hierar node, we encode the number of subelements but we then switch 0s and 1s in the obtained encoding, before padding.
- If we have a \sum node above, we can treat this node as a Lex, Next node with a sequence of orders of length 2. Thus we just need to concatenate the TSO-encoding of the element of O_m , with the TSO-encoding of the element of $f(x)$. It is not a problem that we have TSO-encodings for several distinct orders $f(x), \forall x \in O_m$. Indeed such encodings will ever be compared if they correspond to the same order, since the encoding of the element of O_m is compared first.

Inv node had the simplest way to convert “binarily nextified” encodings that were not yet TSO-encodings. However switching 0s and 1s will be a problem with padding2 TSO-paddings since we may decrement the lex-part and/or increment the contre-lex part later on. What can we do about it? Lex-part and contre-lex part can be in this order or in the other order without altering the result. Let’s just do that. If the depth of the finite width tree structured order is less than 15, then, after switching 0s and 1s, we exchange the four bits of lex-padding and the four bits of contre-lex padding of the last padding2. Its simple arithmetic to see that, after that, we can increment/decrement until we reach the root node. Note that for performance, we would probably remove the Inv nodes, since switching 0s and 1s on all the string constructed so far would be costly. A little rewriting on the tree/expression would be more efficient if the number of elements to sort is enough.

Theorem 4.8. *Lex($1, \omega, \mathcal{O}_\omega^{0,1}$), ContreLex($1, \omega, \mathcal{O}_\omega^{0,1}$), and Next($1, \omega, \mathcal{O}_\omega^{0,1}$) are efficiently universal for finite width tree structured orders (and for orders that can be efficiently embedded in finite width tree structured orders). Given a finite width tree structured order O and its tree/expression T , there is a linear-time and space algorithm that takes as input T and an array A of elements of O and converts A into an array of cells where each cell contains a pointer to the original element of O and another pointer to a binary string suited for “(contre-)lex/next” sorting.*

Proof :

The content of this subsection so far was the proof of this theorem. ■

Corollary 4.9. *The order on the rationals \mathbb{Q} between 0 and 1 is efficiently universal for finite width tree structured orders. Given a finite width tree structured order O and its tree/expression T , there is a linear? time and space algorithm that takes as input T and an array A of elements of O and converts A into an array of cells where each cell contains a pointer to the original element of O and a binary string coding the decimal part of a rational number between 0 and 1.*

4.3 Other nodes for defining orders

We finish this section with syntactic sugar, i.e. new nodes for defining orders. These nodes doesn't add new orders to the class of tree structured orders.

Definition 4.10 (Postlude sequence (new term)). *Given two finite ordinals $i < j$, and two sequences of orders $\mathcal{O}_i, \mathcal{O}_j$, we say that \mathcal{O}_i is a postlude (sequence) of \mathcal{O}_j , if and only if $\mathcal{O}_i[k] = \mathcal{O}_j[k + (j - i)], \forall k \in i$ (we have $k < i < j$). We note $\text{Postlude}(\mathcal{O}_j)$ the set of all postlude sequences of the order sequence \mathcal{O}_j .*

Definition 4.11 (Anti-lexicographic order). *Given two finite ordinals $i < j$, and an uniform sequence of orders $\mathcal{O}_j = ([O])_j$ (O repeated j times), the anti-lexicographic order denoted $\text{AntiLex}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the postludes of \mathcal{O}_j , such that these postludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{AntiLex}(i, j, \mathcal{O}_j))$ (the lengths $L(X), L(Y)$ of X and Y are such that $i \leq L(X), L(Y) < j$),*

- if there exists a finite ordinal $0 < k \leq \min(L(X), L(Y))$ such that $x_{L(X)-k} \neq y_{L(Y)-k}$, then:
 - if $x_{L(X)-k} < y_{L(Y)-k}$ in O , then $X < Y$,
 - if $x_{L(X)-k} > y_{L(Y)-k}$ in O , then $X > Y$,
- otherwise no such k exists, then either $X = Y$, or $L(X) \neq L(Y)$:
 - if $L(X) < L(Y)$, then $X < Y$,
 - if $L(X) > L(Y)$, then $X > Y$.

Anti-lexicographic order is just lexicographic order made backward. Thus it cannot work for arbitrary ordinals ; it works only for finite ordinals. Assuming that the software objects we are applying anti-lexicographic order (or some other order-sequence-operator) are themselves sorted in order to yield the expected result, it may be useful to have AntiLex nodes in order to avoid reversing an array of objects. One can treat these nodes with a for loop concatenating the suborders TSO-encoded strings backward (each substring is written forward but the set of substrings is treated backward, then the final lex-padding is decremented).

Similarly, one can define anti-contre-lexicographic order, anti-hierarchic order and anti-contre-hierarchic order for finite sequences of orders. Here are the definitions for the sake of completeness.

Definition 4.12 (Anti-contre-lexicographic order (new-term)). *Given two finite ordinals $i < j$, and an uniform sequence of orders $\mathcal{O}_j = ([O])_j$ (O repeated j times), the anti-contre-lexicographic order denoted $\text{AntiContreLex}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the postludes of \mathcal{O}_j , such that these postludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{AntiContreLex}(i, j, \mathcal{O}_j))$,*

- if there exists a finite ordinal $0 < k \leq \min(L(X), L(Y))$ such that $x_{L(X)-k} \neq y_{L(Y)-k}$, then:
 - if $x_{L(X)-k} < y_{L(Y)-k}$ in O , then $X < Y$,

- if $x_{L(X)-k} > y_{L(Y)-k}$ in O , then $X > Y$,
- otherwise no such k exists, then either $X = Y$, or $L(X) \neq L(Y)$:
 - if $L(X) < L(Y)$, then $X > Y$,
 - if $L(X) > L(Y)$, then $X < Y$.

Definition 4.13 (Anti-hierarchical order (new term)). *Given two finite ordinals $i < j$, and an uniform sequence of orders $\mathcal{O}_j = ([O])_j$ (O repeated j times), the anti-hierarchical order denoted $\text{AntiHierar}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the postludes of \mathcal{O}_j , such that these postludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{AntiHierar}(i, j, \mathcal{O}_j))$,*

- if $L(X) < L(Y)$, then $X < Y$,
- if $L(X) > L(Y)$, then $X > Y$.
- otherwise $X = Y$, or there exists a finite ordinal $0 < k \leq L(X) = L(Y)$ such that $x_{L(X)-k} \neq y_{L(Y)-k}$, then:
 - if $x_{L(X)-k} < y_{L(Y)-k}$ in O , then $X < Y$,
 - if $x_{L(X)-k} > y_{L(Y)-k}$ in O , then $X > Y$.

Definition 4.14 (Anti-contre-hierarchical order (new term)). *Given two finite ordinals $i < j$, and an uniform sequence of orders $\mathcal{O}_j = ([O])_j$ (O repeated j times), the anti-contre-hierarchical order denoted $\text{AntiContreHierar}(i, j, \mathcal{O}_j)$ is an order defined on the set of all elements of the postludes of \mathcal{O}_j , such that these postludes have length at least i . This is the order satisfying $\forall X, Y \in \text{Domain}(\text{AntiContreHierar}(i, j, \mathcal{O}_j))$,*

- if $L(X) < L(Y)$, then $X > Y$,
- if $L(X) > L(Y)$, then $X < Y$.
- otherwise $X = Y$, or there exists a finite ordinal $0 < k \leq L(X) = L(Y)$ such that $x_{L(X)-k} \neq y_{L(Y)-k}$, then:
 - if $x_{L(X)-k} < y_{L(Y)-k}$ in O , then $X < Y$,
 - if $x_{L(X)-k} > y_{L(Y)-k}$ in O , then $X > Y$.

5 Other results on finite width tree structured orders

Theorem 5.1. *For any finite width tree structured order, its domain is countable.*

Proof :

We can deduce this theorem from the fact that any element of this order can be associated (efficiently) to a finite binary string. However, we give here a proof that does not need this previous result. The proof follows the definition and uses the fact that a countable union of countable sets is countable.

- Finite orders are countable.

- Given a countable order O , $\text{Inv}(O)$ is a countable order.
- Given two ordinals $i < j \leq \omega$, and an ultimately periodic sequence of countable orders \mathcal{O}_j , $\text{Prelude}(\mathcal{O}_j)$ is countable and each prelude has finite length, hence the set of elements of the prelude are a finite union of countable sets, so are countable. Thus the union of elements of the sequences in $\text{Prelude}(\mathcal{O}_j)$ are a countable union of countable sets, hence this union is countable. $\text{Lex}(i, j, \mathcal{O}_j)$, and $\text{Hierar}(i, j, \mathcal{O}_j)$ are countable orders.
- Given a master finite order O_m , and a corresponding sequence of countable orders \mathcal{O}_j , $\text{Domain}(\sum(O_m, \mathcal{O}_j))$ is the finite union of the countable domains of the O_i , hence is countable.

■

6 Orders that almost fit in our framework and open problems

We start this section with the result of this article we like the most. What about \mathbb{Q} ? How can rationals be efficiently TSO-encoded? No solution seems to appear when we are only given the numerator and denominator, since, when you compare two rationals, you multiply the numerator of one with the denominator of the other, and vice-versa, and you compare the two results of these multiplications. But one can consider the continued fraction representing a rational: $\frac{p}{q} = n_0 + \frac{1}{n_1 + \frac{1}{n_2 + \frac{1}{n_3 + \frac{1}{n_4 + \dots}}}}$. A number is rational, if and

only if it can be represented by a finite continued fraction. Assume we have non-negative rationals, since it is easy to add the sign with generalized sum. Then for these rationals we have a sequence of positive integers. When the integer of rank 0 is greater then the rational is greater, when the integer of rank 1 is greater then the rational is lower, when the integer of rank 2 is greater then the rational is greater, when the integer of rank 3 is greater then the rational is lower, etc. We have a problem for when this sequence finishes however since we have to apply lexicographic way of dealing with end of sequence alternatively with contre-lexicographic way of doing so. We note here however that only n_0 may have value equal to zero (it is the case only when the rational is less than 1), since continued fractions integers are derived from gcd-algorithm, that the continued fraction may be written so that the last number is at least 2 if it is not n_0 . It is thus easy to see by induction that any continued fraction without integer part is strictly between 0 and 1, and then:

- If we finished with a number of even rank, then the rational is lower than any other rational with the same beginning of continued fraction. We need to encode one more “extended integer” with positive infinite value because it will be compared with inverse integer order with the next integer.
- If we finished with a number of odd rank, then the rational is greater than any other rational with the same beginning of continued fraction. Here again we need to encode one more “extended integer” with positive infinite value because it will be compared with integer order with the next integer.

Since positive integers are a suborder of the order $O^{uint} = \text{Hierar}(1, \omega, \mathcal{O}^{0,1})$, the order on extended integers is a suborder of $O^{uint.ext} = \text{Next}(2, 3, (O^{0,1}, O^{uint}))$ (a leading 1 denotes positive infinity, a leading 0 denotes an integer). We have that non-negative rationals order is a suborder of $O^{\mathbb{Q}^+} = \text{Next}(1, \omega, ([O^{uint.ext}, \text{Inv}(O^{uint.ext})])$.

We let the reader write the expression for all rationals using generalized sum. Note that it is our first use of an ultimately periodic sequence of period greater than 1. Unfortunately it is not linear? time to compute the continued fraction of a rational number. We may end-up with $\Theta(M(\frac{m}{2n}) \times \log(\frac{m}{2n}) \times n)$ complexity, if all rationals are roughly of size $\frac{m}{n}$, with a denominator and a numerator of size approximately $\frac{m}{2n}$, and $M(p)$ denotes the time complexity of multiplication/division of integers of p bits (gcd-algorithm to compute the continued fraction can be done with $\log(p) = \log(\frac{m}{2n})$ such divisions). Nevertheless, we have this theorem.

Theorem 6.1. \mathbb{Q} can be almost efficiently embedded in a finite width tree structured order.

Open problem 6.2. Prove or disprove that \mathbb{Q} can be efficiently embedded in a finite width tree structured orders. (A linear time algorithm for computing the continued fraction of a rational would be a solution.)

We now turn our attention on the many orders that do not directly fit in our framework. We saw that Next had two simple linear extensions, Lex, ContreLex. But in fact, sequence termination could be ordered in between the elements of the next order in the sequence. Let us call this kind of linear extensions middle-lexicographic, MidLex; for $\mathcal{O}^{0,1}$, the choices of middle-lexicographic linear extensions are restricted to one choice: sequence termination is between 0 and 1. With $\mathcal{O}^{0,1,2,\dots,9}$, we have many more choices, but still we could do something. Indeed for a sequence of finite orders, we can embed each order in the order of cardinality one more; this additional element would be the image of the sequence termination. It would be very cumbersome with non-uniform sequence (sequence of period lengthier than one), but it would work.

With infinite orders, it may become very difficult. For example, what can we do if we have a middle-lexicographic order on an uniform sequence of the rationals order (or more precisely of the order $\mathcal{O}^{\mathbb{Q}}$) where we want to have sequence termination mapped to π , or $\sqrt{2}$?

Open problem 6.3. Study in depth the frontiers of (finite width) tree structured orders.

Open problem 6.4. We have all these callback comparison functions in the wild. Is it possible to code an algorithm that will take the assembly code, C code, or “insert your favorite programming language here” code of such a function and extracts the corresponding finite width tree structured order definition/expression? The problem must be undecidable but for “common” comparison functions it should be possible.

Last but not least, we look at open problems around universality results. Please read again the following definition, but this time thinking that the orders may be partial or total.

Definition 6.5 (Universal order). We say that an order O is universal for a class of orders \mathcal{A} , if for any order $O' \in \mathcal{A}$, there exists an embedding of O' in O . An order embedding is an injective mapping such that order is preserved.

“Order is preserved” means that no order relation is removed or added (adding was not possible with total orders). What we proved efficiently, and that Cantor also proved with some minor addition (double each binary digit and put 0 on the last digit, 1 otherwise), is that the partial order Next on a sequence of ω binary orders is universal for the countable total orders. Using the partial order Next limits comparison to solving the unique question if it exists. We can say that all countable total orders have a representation that is “questionable”. The fact that this representation is “questionable” implies that this representation is well-ordered. This is counter-intuitive if you think that rationals can be represented in a well-ordered way. A “questionable representation” has two parameters: the maximum cardinality of the order-items it

uses that we call its width, and the ordinal length of the sequence of order-items. (It reminds us of path-width and path-decompositions, but suited for orders.) Countable total orders have a binary questionable representation of length ω , but they do not all have a unary questionable representation of length ω .

Open problem 6.6. *Is it true that any total order O has a finite (binary?) width questionable representation? (The length of the representation may adapt to the cardinality of O .) If not, what is the minimal cardinal of an order without finite (binary?) width questionable representation?*

Note that the same problem for partial orders has an immediate answer. Indeed, the partial order POC with four elements a, b, c, d such that a is less than c and b is less than d does not admit a questionable representation. (Two elements are not ordered if and only if they don't have a question if and only if one is the prefix of the other. Thus, without loss of generality, a is a prefix of b . But then, since c is more than a , it has a question with a , and since a is a prefix of b , c has a question with b , and c is more than b , a contradiction.) It seems we must consider partial questionable representations, where the order-items in the representation may be partial. If the question yields two incomparable element-items, then the elements are incomparable. Hence, we have the two following open problems.

Open problem 6.7. *Is it true that any partial order PO has a total questionable representation if and only if it does not contain some partial order POC taken in a finite set of forbidden finite partial orders?*

Open problem 6.8. *Is it true that any (countable) partial order PO has a (finite) partial questionable representation?*

7 Tree Structured Orders Definition Language

We would like to create a language to create orders and sort software objects accordingly. We hope the software and scientific community can contribute to this goal. Here is an attempt at what it would look like. It resembles SQL, but with a different scope:

- it could be used for order by clauses of SQL queries or other query languages,
- but it could also be used to generate code in your favorite programming language if you intend to sort objects without using a database,
- or it could be used as an input of a library that provides functions to prepare the array to sort (TSO-encoding) and sort accordingly.

Both generated code or dynamic code in library could provide a comparison function and a “nextification” function, computed from the tree structured order definition, and both code could switch between comparison model and lexicographic model, whichever is faster, according to the number of elements to sort and other parameters deduced from the tree structured order definition.

First each node of a tree structured order is centered around an object (or a structure in C language, etc.). The root node is centered on the main objects, the ones that correspond to the wanted level of granularity.

Let us explain our ideas using an example from our current work in business software for freight forwarders. Each night/morning, a freight forwarder receives trucks with freight from other freight forwarders. For each truck and each day, we have an “Arrival note” in the database. To this arrival note are linked shipments, each shipment has a certain number of handling units. We have these three levels of granularity: arrival note (coarse grained), shipments, handling units (fine grained). Each of these three

classes may have between a dozen and a few hundreds fields. Assume we want to sort shipments according to the name of the freight forwarder that brought his truck (field found in the arrival note), the weight of the freight in the truck by decreasing order (arrival note), the weight of the shipment by increasing order, the barcodes of the handling units.

A Tree Structured Order Definition for this sort would look like:

```
Next (
  current.arrival_note.forwarder_name STRING collation=UTF8_test,
  current.arrival_note.total_weight DOUBLE DESC,
  current.weight DOUBLE,
  Lex (
    1,
    0, //0 codes omega
    current.number_of_handling_units, //the actual prelude length
    current.array_of_handling_units, //the array of pointers to redefine current
    ([
      //current has been redefined for the suborders.
      current.barcode STRING collation=ASCII
    ])
  )
)
```

Note that we assumed denormalization or caching for the total weight of an arrival note. With TSO-encoding, that value, if not directly available, would be computed once and stocked into the TSO-encoding. With black box model and callbacks, you have to cache this value somehow, otherwise it will be computed on each comparison.

8 Conclusion

We did all this research because of the following reasons:

- linear time sorting was an old problem we were thinking about,
- we encountered real life problems with MySQL optimizer where it would use an index instead of another because of the ORDER BY component of the query, and this choice was dramatic since the chosen index was not discriminating enough. Because of the number of rows, it would end up with an execution plan taking more than several minutes instead of an execution plan taking less than 3 seconds; When you have to optimize over and over dynamic queries with over 100 lines of SQL code with an ORDER BY that the user can choose among more than a dozen choices, and when you have almost no control on what does the optimiser, you have to add dynamically USE INDEX or FORCE INDEX to ensure that the execution plan is correct. It gets very frustrating that your DBMS does not have an option “filtering is hard, sorting is easy”. We hoped to make sorting out of scope for index choice altogether. With SQL we noticed how Lex, Next (partial) order was important.

We were thinking about optimizing DBMS all along, and more recently we thought it may also help search engines. For search engines, we lack knowledge but probably the score of a webpage is coded into a

float/double. Since this score can be converted efficiently to unsigned int (see our library on GitHub), and the number of webpages is enormous, it makes a good candidate for efficient use of our ideas. Moreover all our results are straightforwardly parallelizable.

One of the interests of (efficient) universality results is that they open a new possibility: Use optimized hardware for “nextifying” and sorting. (We have specialized hardware for things as stupid as mining bitcoins; it would be very interesting to have specialized hardware for sorting; with comparison based sorting, because of the comparison function, you need a CPU, so there is less room for a dedicated hardware.)

We benchmarked our ideas (see <https://github.com/LLyaudet/TSODLULS> and try it yourself) on our laptop with finite orders ((signed and unsigned) integers of 8, 16, 32, and 64 bits respectively, and floats and doubles), and also with strings. Although our ideas are memory expensive compared to comparison based sorting, we succeeded to beat glibc comparison based qsort implementation in most cases with finite orders. We tested power of 2 number of elements between 1 and 2^{25} . We were comparing our algorithm to glibc qsort in three settings:

- “direct” where the input is an array of keys (an element of a finite order),
- “in short cell” where the element of the array is a cell containing a key that is an unsigned integer on 64 bits, and a pointer to the object to be sorted according to the key (128 bits for the short cell).
- “in long cell” where the element of the array is a cell containing a pointer to the key, a pointer to the object to be sorted according to the key, the size taken by the key, and the size allocated for the key (it is 256 bits for a cell without the key).

Our sorting algorithm always works “in cell” and we added the cost of creating the “in cell” array and filling it in our benchmarks. However, we implemented our algorithm in two variants: one for short cells and one for long cells. The variant on short cells, when it can be used, is faster than the variant on long cells. For finite orders, our algorithm on short cells is faster than glibc qsort, as soon as there are at least 64 elements to sort. It can then be between 1.3 (random poor performance) and 14 times faster (on int8) but there is a lot of variations. (Please launch the custom benchmark wrapper PHP script in TSODLULS with choice 1: qsort direct and choice 2: TSODLULS_sort_radix8_count_insertion_short with macraffs. Your mileage may vary.) It is fair to say that in our experiments, we were most of the time at least 2 times faster than glibc qsort; and it was not uncommon to be 3 or 4 times faster. We truly hope it may be #ClimateChangeBrake.

Results for strings are not that good. It seems that our algorithm on long cells is not cache efficient when dealing with long strings. Hence, for a small number of elements we can be as bad as 40 times slower than qsort. When the number of elements grows, it gets better, and we ended being 2 times faster with all optimizations activated. (Please launch the custom benchmark for strings wrapper PHP script in TSODLULS with choice 1: qsort direct and choice 2: TSODLULS_sort_radix8_count_insertion with macraffs. The benchmark sorts strings of printable ascii characters with two settings: raw order of the ascii characters, or custom collation to have the order AaBbCc... Results are slower for custom collation, but they are “less slower” for our algorithm than for glibc qsort, because we only apply the custom collation once during the “nextification”, whilst glibc qsort applies it for each comparison. We tested with strings of length between 0 and 128 and no common prefix, and with strings of length between 100 and 128 and a common prefix of length 90. With a huge number of elements, we were still 2 or 5 times slower than

qsort. We finally succeeded to beat glibc for sorting long strings when the number of strings is enough by optimizing the padding. First padding was byte-level padding, hence we were multiplying the length of the strings by two, because of the lexicographic ordering emulated by next partial order. We saw that too much memory waste was the main problem and we implemented bit-level padding because it was sufficient to increase the length of the strings by $\frac{1}{8}$. It was a huge improvement. Your mileage may vary.) If you look at the source code in detail, you will probably agree that all the optimizations outside of the sorting algorithms are too complicated for a simple task such as sorting. It makes more important to design an order description language that will deal with this kind of code generation.

Michaël Rao suggested us to look at Boost C++ library. It is clear there that the algorithmic ideas for sorting, and nextifying finite orders are not new. We will try to improve our C library using ideas from Boost.

Acknowledgements

We thank God: Father, Son, and Holy Spirit. We thank Maria. They help us through our difficulties in life.

We thank Michaël Rao for his interest in our results and his comments. We thank Pascal Koiran and Gilles Villard for useful corrections on the complexity of results about rationals.

References

- A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74–93, 1998.
- G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Math. Ann.*, 46:481–512, 1895.
- D. Harvey and J. van der Hoeven. Faster integer multiplication using short lattice vectors. *CoRR*, abs/1802.07932, 2018. URL <http://arxiv.org/abs/1802.07932>.
- H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954. Report R-232, "2.4.6 Internal Sorting by Floating Digital Sort", pp. 25–28.