Lexicographic Order

Julio Candanedo
Department of Physics, Arizona State University, Tempe, AZ 85287, USA

1 introduction

Here we provide python functions to study numerical *lexicographic-order* over arrays. These may be installed from a github-repositiory, `https://github.com/jcandane/LexicographicSort` (with or without numba), via `$git clone https://github.com/jcandane/LexicographicSort`. Or by copy-n-pasting the following code in this pdf.

1.1 list-of-tuples

Numerical *Lexicographic-order* is defined on a 2-dimensional array, a matrix (over some data-type), and we would like to sort along a given axis, say the columns, while preserving the rows. Alternatively, we may call this data-structure as a *list-of-tuples*, in order to distinguish that the *tuples'* order is preserved (rows), and we permute the *list*, i.e. columns. We may denote this 2d array, list-of-tuples as: $A[n|I]$, for a *list-index* (denoted by I) that acts *row-wise* refers to the Ith tuple, while acting *column-wise* with a *tuple-index* (denoted by n).

1.2 tuple-tuple comparison

Now we would like to define a notion of a tuple being smaller/larger than another, we denote this comparison by $\prec$ xor $\preceq$. Suppose we have two tuples (of the same finite length, filled in with $n+1$ entries) $a, b \in \mathbb{N}^{n+1}$, then $a < b$ if we preform an element-wise subtraction $b - s = c$, and the first nonzero entry in c (from left-to-right) is positive. Equivalently, following [Munkres, 2000]:

$$(a[0], a[1], \cdots, a[n]) \prec (b[0], b[1], \cdots, b[n])$$
$$\text{starting left-to-right, if } a[i] < b[i] \text{ , else } a_i = b_i \quad \& \quad a[i+1] < b[i+1] \text{ (for } 0 < i \leq n) \quad .$$

1.2.1 tuple-tuple equality

Suppose we have two tuples (of the same size, filled in with $n+1$ entries) $a, b \in \mathbb{N}^{n+1}$, then $a = b$ iff $a[i] = b[i]$ for all i (the ith element of a matches the ith element of b).

1.2.2 tuples-to-numbers

Next we provide a comparison is useful for mapping tuples to a number-system, while maintaining order (given two tuples, if one is greater than the other, the corresponding numbers will also obey the relationship). Suppose for a number-system we have radix/base r.

Then we may map a tuple to number in this system (1-1) by the inner-product (superscripts here are the exponential operation, i.e. power):

$$\text{num}_r(a) = a[n] * r^n + a[n-1] * r^{n-1} + \cdots + a[1] * r^1 + a[0] * r^0 \qquad .$$

E.g. the binary number-system ($r = 2$), decimal ($r = 10$), duodecimal system ($r = 12$), hexadecimal ($r = 16$). It can be shown if $a < b$, then their numbers base r also comply with $\text{num}_r(a) < \text{num}_r(b)$.

1.3 Definition of Lexicographic/Dictionary/Sawtooth/ssort order

A list-of-tuples (the index-array) is said to be *lexicographically*-ordered if all pairs-of-tuples comply with the tuple-tuple comparison $\prec$. Note this does not include equality, and hence all entries must be unique, we denote this order as *sssort* or well-ordered in a mathematical sense. If we relax to equality in the comparison, we allow for duplicates and use the $\preceq$ comparison instead, this is denoted as *ssort* order.

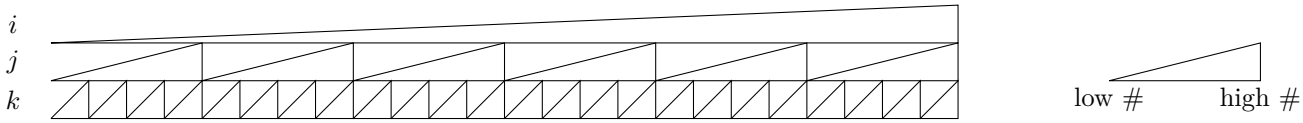

Figure 1: A cartoon depicting the ordering of a list-of-tuples array for a 3-index $A[i, j, k|I]$, along i numerical values are sorted in ascending-order. Afterward j is sorted in ascending-order within duplicate-values in i (the previous column). Lastly, k is sorted in ascending-order within duplicate-values in i and j (both previous arrays.

1.3.1 Lexicographic Sort Algorithm

Suppose we have a generic unsorted, potentially non-unique entry, list-of-tuples array (with column labels **0, 1, 2, 3**, $\cdots, \mathbf{N}$):

$$A\,[\ \mathbf{0,\ 1,\ 2,\ 3,}\ \cdots, \mathbf{N}\ |\ I\ ] \qquad .$$

Next we introduce an auxiliary-array called the domain, u, which constrains the ranges of successive (column/index) sorts. For instance the sort is executed within successive elements in the domains array (range of list indices are $[u[n], u[n+1]]$), a constrained-sort:

$$A\,[\ \mathbf{0,\ 1,\ 2,\ 3,}\ \cdots, \mathbf{N}\ |\ u[n]\ :\ u[n+1]\ ] \qquad .$$

Upon sorting each index (**0, 1, 2, 3**, $\cdots, \mathbf{N}$), the constrains on the sort are changed, and must increase. Initially there are no constraints. Then differences between the elements in the sorted index, form domains which constrains the sort of successive sorts. Importantly, the domains u^n must unionize with earlier domains u^m ($m < n$) to respect earlier boundaries:

$$u = \bigcup_{\eta} u^{(\eta)} \qquad .$$

It would be convenient to have an `argsort` over small-domains, that can be parallelized over these small-domains. This is quite natural for `mergesort` which partitions the data-set, before merging. Except we would like to specify this partition, which is potentially of different sizes. Then implement a sorting algorithm depending on the size of the partition: either *Insertion-sort* xor *Merge-sort*.

1.3.2 computational time-complexity

Let $N = M^n$ (M is the length of a side of the dense-tensor), then we have to do successively constrained sort given by:

$$\begin{aligned}\mathcal{O} &\sim M^n \left(\log\left(\frac{M^n}{M^0}\right) + \log\left(\frac{M^n}{M^1}\right) + \cdots + \log\left(\frac{M^n}{M^n}\right)\right) = \frac{n+1}{2} M^n \log\left(M^n\right) \\ &\sim \frac{n+1}{2} N \log N \qquad .\end{aligned}$$

Compare this to: $\mathcal{O} \sim nN \log N$, the independent sort of each column.

1.3.3 theorems

The following may be easily shown:

- Adding an arbitrary (random) column to the right sssort-order order respects the original sssort-order.
- Removing an arbitrary-row (a tuple) maintains sssort-order.
- Removing an arbitrary-column in sssort-order breaks it to ssort-order (creating the possibility of duplicates).

1.4 using `numba`

The Python-*for-loop* is notoriously known for its slowness. Therefore, *just-in-time* compilers such as [Numba *et al.*, 2015] may aid in it's computational speed. It may be easily installed by: `$ pip install numba`, and run by:

```
1 from numba import njit
```

for the optional python-definition/function decorator `njit`, as shown in the sample codes.

2 binary-search

Given a list with N entries the naïve-search requires a time-complexity of at worse $\mathcal{O} \sim N$ comparisons, in order to return the searched-element index-location. If the list is first sorted instead (which takes roughly $\mathcal{O} \sim N \log N$ operations), then the binary-search requires $\log N$ operations. This is beneficial if more than $\log N$ searches are desired on the same list. For python, [NumPy, 2022] has the `numpy.searchsorted` subroutine, but is not constrained by lower nor upper limits, that are required for tuple-binary-searches. Therefore we introduce 4 new subroutines, accelerated by [Numba *et al.*, 2015]: Left, Right, interval, and tuple-interval binary-searches.

2.1 Left/Right/Interval binary-search

The binary-search looks to reduce the list size by half with each iteration. We start in the middle-index of the array. From here we have 2 options if this value is smaller (larger)

than the chosen value, the algorithm is applied to the left (right) part of this partition. We assume the searched list may have duplicates, therefore we desire an interval, of the first (Left-binary-search) and last (Right-binary-search) occurrence of the search. If the element is not found, the Left and Right binary searches yield the same value, the *insertion point*. This is the point whereby if the searched element is inserted would keep the list ordered. Hence the interval-binary-search yields a 2-tuple, indicating the `start` and `end` (non-inclusive index) location in the searched array; if no match is found `start = end`, and this indicates the insertion spot. Example "pseudo-codes" are simple and hence are in-fact functional python-code, for each each type of binary-search.

2.1.1 Left

```python
#@njit ## for numba (comment out if not needed)
def L_binarysearch(A, T, L=0, R=None):
    """
    np.searchsorted has no contraints, i.e. L nor R
    GIVEN: A (1d sorted numpy.array)
           T (searched for entry)
           *L (lowest index to search for)
           *R (highest index to search for)
    GET:   L (left-(lowest) most index of entry)
    """
    if R is None:
        R = len(A)
    while L < R:
        m = (L + R) // 2
        if A[m] < T:
            L = m + 1
        else:
            R = m
    return L
```

2.1.2 Right

```python
#@njit ## for numba (comment out if not needed)
def R_binarysearch(A, T, L=0, R=None):
    """
    np.searchsorted has no contraints, i.e. L nor R
    GIVEN: A (1d sorted numpy.array)
           T (searched for entry)
           *L (lowest index to search for)
           *R (highest index to search for)
    GET:   R (right-(highest) most index of entry)
    """
    if R is None:
        R = len(A)
    while L < R:
        m = (L + R) // 2
        if A[m] > T:
            R = m
        else:
            L = m + 1
```

```
    return R
```

2.1.3 Interval

```
#@njit ## for numba (comment out if not needed)
def interval_binarysearch(A, value, L=0, R=None):
    return L_binarysearch(A, value, L=L, R=R), R_binarysearch(A, value, L=L, R=R)
```

2.2 tuple-Binary-search

Suppose we have a list-of-tuples, lexicographically-sorted, the binary-search may be straightforwardly implemented. The tuple is searched one entry at a time, successively in order of the tuple, each search reduces the search leftmost and rightmost tuple, until all elements of the tuple have been searched. The final answer like before is a 2-tuple, the first (L) and last (R) occurrence.

2.2.1 Interval

```
#@njit ## for numba (comment out if not needed)
def tuplebsearch_interval(B, value, L=0, R=None):
    for columns in range(B.shape[0]): ### over entries/columns of the tuple
        L, R = interval_binarysearch(B[columns,:], value[columns], L=L, R=R)
    return L, R
```

2.2.2 time-complexity

As mentioned the time-complexity of a single binary-search on list of numbers (of length N, given an already sorted list), is $\log N$. Let $N = M^n$ (M is the length of typical numerical domains), then we have to do successively constrained binary-searches given by:

$$\mathcal{O} \sim \log\left(\frac{M^n}{M^0}\right) + \log\left(\frac{M^n}{M^1}\right) + \cdots + \log\left(\frac{M^n}{M^n}\right) = n\,\frac{n+1}{2}\log M = \frac{n+1}{2}\log\left(M^n\right)$$
$$\sim \frac{n+1}{2}\log N \qquad .$$

This may be compared to $n \log N$, or n binary-searches on an array of length N.

3 Domains Finder

Suppose we have a sorted-array `A`, we can determine the maximum amount of domains (a range of identical entries) by: $\frac{\texttt{len(A)}}{\texttt{A[-1]-A[0]}}$. There are two algorithms to find the range-of-duplicates: linear and right-binary-search/logarithmic, with time-complexity as $\mathcal{O} \sim N$ and $\mathcal{O} \sim n \log N$ respectively, where n is the actual number-of-domains and N is the total-number of entries.

Therefore in a wholly-unique sorted-array, $n = N$, and the right-binary-search method will always be slower. The condition in which the logarithmic method is faster is given by:

$$\begin{aligned} n \log_2 N &< N \qquad (\text{if } \texttt{true},\ \text{right-binary-search is faster}) \\ \frac{N}{\max} = n &< \frac{N}{\log N} \\ \texttt{(A[-1]-A[0])} &< \log_2 N \approx \texttt{(len(A).bit\_length())} - 1 \qquad . \end{aligned}$$

Above N is an integer and thus we need a quick method to compute the base-2-logarithm over an integer. This may be achieved approximately (only perfect for exact powers of 2) by counting the number of bits representing the integer (of the length of `A`).

3.1 linear algorithm

This algorithm computes the difference of all successive elements of a given 1d array.

```
def linear_domain(A):
    """
    GIVEN: A (sorted 1d numpy array)
    GET:   domains (sorted 1d numpy array, ranges of unique elements)
    """
    domain = np.where( np.diff( A , prepend=-1, append=len(A)) != 0)[0]
    return domain
```

3.2 log algorithm

This algorithm computes the: successively-constrained successive right-side binary-search. That is we successively search for the right-most (highest-index entry) of a given element, until we get to the end of the array.

```
def log_domain(A):
    """
    GIVEN: A (sorted 1d numpy array)
    GET:   domains (sorted 1d numpy array, ranges of unique elements)
    """
    domain = [0]
    while domain[-1]!=len(A):
        domain.append( R_binarysearch(A, A[domain[-1]], L=domain[-1], R=None) )
    return np.asarray(domain)
```

3.3 combined

Given the constraint of §3, we combined both previous domains searching algorithms of §3.1 and §3.2.

```
def domain_finder(A):
    """
    GIVEN: A (sorted 1d numpy array)
    GET:   domains (sorted 1d numpy array, ranges of unique elements)
    """
```

```
    if (A[-1]-A[0]) < len(A).bit_length() - 1:
        return log_domain(A) # log_domain_finder(A)
    else:
        return linear_domain(A) # linear_domain_finder(A)
```

3.4 find uniques

The leading entry of the domains array yields the unique entries.

4 ssort

Next, we would like to implement the lexicographic-sort in python/numpy. This can be done sub-optimally, using methods from [NumPy, 2022], however still achieving pseudo-linear time-complexity, see fig. 2. The essential idea is to: implement typical 1-dimensional sort along a column, only to find domains, §3. This can be done for every column, once done this is lexicographically-sorted. Crucially, it is important to use stable sorting algorithms: insertion and merge sort. The implemented code is:

```
def ssort(A, return_unique=False):

    ARG = np.argsort(A[0] , kind="mergesort")
    A   = A[:,ARG]

    ikk = np.array([], dtype=np.int32)
    for k in range(1,len(A)):
        ik = np.where( np.diff(A[k-1], prepend=A[k-1,0]-1, append=A[k-1,-1]+1) != 0)[0]
        ik = np.union1d(ikk, ik)
        for l in range(len(ik)-1): ## For
            argssort = np.argsort(A[k,ik[l]:ik[l+1]] , kind="mergesort")
            A[:,ik[l]:ik[l+1]] = (A[:,ik[l]:ik[l+1]])[:,argssort]
            ARG[ik[l]:ik[l+1]] = (ARG[ik[l]:ik[l+1]])[argssort]
        ikk = 1*ik

    k = len(A)-1 ## 8/30
    if return_unique:
        return A, ARG, ik
    return A
```

An improvement can be made for a native tuple sorting algorithm or a sorting algorithm within certain limits and within certain columns currently unavailable in [NumPy, 2022].

References

[Munkres, 2000] Munkres, J. R. (2000). *Topology*, volume 2. Prentice Hall.

[Numba *et al.*, 2015] Numba, Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA. Association for Computing Machinery.

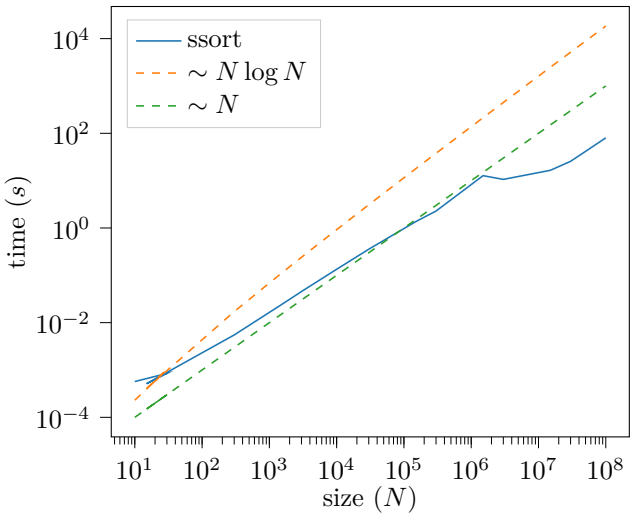

Figure 2: comparison the implemented lexicographic-sort algorithm with pseudo-linear time scaling.

[NumPy, 2022] NumPy (2022). Numerical python. `https://numpy.org/doc/stable/reference/`. [Online; accessed 7-January-2022].