Matching under inequality via sorting algorithms

Dr. Markos Mitsos*
Deutsche Krankenversicherung AG DKV - ERGO
Actuarial Department

January 8, 2017

Abstract

The solution of many problems encountered while processing data with APL depends on matching algorithms. This is taken to mean that data points are searched in a reference list; the found matches are noted and used subsequently. In the case of vectors basic APL provides the necessary primitive to conduct the search via Index Of as in

```
IND ← REF ⍳ DAT.
```

The more general case where (rows of) matrices are to be matched can be tackled through enclosing rows.[1] This solution though leads to very slow processing. Instead a sorting algorithm can be used. (In fact *some* algorithm underlies ⍳ itself as well...)

This paper discusses the generalization of the problem to different kinds of matching under inequality. In one case a possible solution with the help of "special sorting" algorithm is proposed. For the latter the APL primitive Grade Up ⍋ cannot be used.

A big part of the article discusses simple examples or provides the motivation to examine matching problems in a more general setting. This parts could be used in a presentation and may of course be skipped by the expert. We try to use APL code which is basically dialect independent but uses control structures for legibility.

Contents

*markos.mitsos@ergo.de

[1] In newer version of Dyalog APL however the primitive ⍳ has been enhanced to encompass the case of matrices.

List of Algorithms

1 Matching under equality via sorting algorithms in APL

This (first) section is an introduction. The presented problems and algorithms are only intended as examples and a motivation for more complex cases.

1.1 The problem of matching rows of matrices

We want to match data, denoted by `DAT`, with a reference list, denoted by `REF`, in APL. We are interested in the case of matrices with the same number of columns (possibly one) and will always assume that `DAT` and `REF` are such.

We want to find the first occurrence of each row of `DAT` in `REF` (meaning the first identical row). We are only interested in an effective and fast method to solve the problem in case at least one of the matrices is big (at the very least some tens of thousands of rows) and the other at least medium sized (at the very least some hundreds of rows).

That fore we ignore the general case of arbitrary depth and assume the matrices are simple. We can then further assume that they are numeric (for example by replacing alphanumerical data with their Unicode code).

In fact in most cases we will only be interested in integers. Of course the discussed algorithms can be generalized to other data types.

1.2 Solutions via APL primitives

There are of course many possible solutions to the problem using APL primitives. For example the "monster comparison"

```
IND ← (⍳ 1 ↑ ⍴REF) ×[2] DAT ∘.= ⍉REF
```
(1)

returns all matches and can be used the find the first one. It is of course not of much practical use.

Just the same enclosing the matrices along the second axis will deliver the desired result via

```
IND ← (⊂[2] REF) ⍳ (⊂[2] DAT) .
```
(2)

However enclosing big matrices is not very effective and/or fast (although the process has been constantly improved in subsequent versions of many APL dialects). Furthermore the used primitive Index Of may not (depending on the used APL dialect) be optimized for this special kind of data.

Newer versions of Dyalog APL are a special case. Here the APL primitive ⍳ has been enhanced to encompass the case of matrices. As far as publicly known this is done by hashing each row and by this means reducing the problem to ordinary numerical vectors.

1.3 Solution with sorting algorithm

The problem with expression (1) is that it compares all provided pairs of rows (from the data and the reference respectively). Effective ways to compare only "necessary" pairs are encoded in the well known sort algorithms. The APL primitive ⍋ (of course!) uses those.

A possible solution to the problem can be created along the following lines.

1. Catenate the two matrices while keeping track of the origin of each row.
2. Sort the result.
3. Split the result into (disjoint) groups of identical rows.
4. Choose the first element of `REF` from each group as a match for all elements of `DAT` in the same group.

The main part of the work is the sort. The principle of sorting and then grouping can also used to solve many other problems. The algorithm should not depend on the sort being stable.[2] To achieve this some additional information (like the row numbers of the two matrices) may be used.

[2] In most APL dialects the Grade Up primitive ⍋ seems to implement a stable sort. There does not seem however to exist a direct warranty for this in the language references! On the other hand the APL Standard demands stable sorting, so any APL dialect adhering to the Standard must provide it.

1.4 Match on equality as a example for sorting algorithms

We write down one possible implementation of the algorithm outlined. A similar function that also takes care of cases a little more general works very well with matrices with significantly more than one million rows.

The algorithm selects the first row of each group (of rows). It must therefore ensure that this is "the right one", namely the *first* row of REF fulfilling the necessary criteria. (In absence of such a match a dummy must be used.) This is achieved through the selected sort.

The version we give is not necessarily always the fastest but is easy to read. The construction of the sort vector takes up up to 75% of the total runtime (for random data) and is build in a way that does not assume that ⍋ is stable.

Algorithm "match on equality"

```
IND ← (1 ↑ ⍴DAT) ⍴ (NO_MATCH ← 1 + 1 ↑ ⍴REF)
    ⍝ result is 4 byte integer

MAT ← REF ,[1] DAT ⍝ catenate
ROW ← ⍳ 1 ↑ ⍴MAT   ⍝ member numbers

V ← ⍋ MAT , ROW ⍝ sorting vector
MAT ← MAT[V;]   ⍝ sort matrix
ROW ← ROW[V ]   ⍝ sort member numbers

BEG_I ← 1 , ∨ / (1 ↓[1] MAT) ≠ (¯1 ↓[1] MAT) ⍝ inequality?

DAT_I ← NO_MATCH ≤ ROW ⍝ mark data

V ← BEG_I / ⍳ 1 ↑ ⍴MAT                  ⍝ begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ⍴MAT))  -  V ⍝ members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
    ← DAT_I / MAT_NR / NO_MATCH ⌊ BEG_I / ROW
        ⍝ member number of first match in reference
```

2 Matching with one, "final" inequality via sorting algorithms in APL

This section presents a problem that cannot be solved through enclosing (as in expression (2) of section 1). The sorting algorithm "match on equality" of subsection 1.4 however can be modified and used for this purpose. The necessary sort is done by the Grade Up APL primitive ⍋.

2.1 Cases where a match is not an equality

In many cases the desired match cannot be expected to be an exact one. The most common example in German health insurance is "begin business time". Let us consider premiums or annuity values for a certain plan.

The plan has been recalculated on certain dates $r_i \in \mathcal{R}$, $i \in \{1,\dots,l\}$. Each person j, $j \in \{1,\dots,m\}$ in this plan has contracted at a certain date d_j. At the time of contract the valid annuities where those with the highest date before or at contract.

The equality is possible but not probable. For each person j the subset

$$\mathcal{R}(d_j) = \{r = r_{max}(d_j)\} \subset \mathcal{R}\,, \quad r_{max}(d_j) = \max\{r; r \le d_j\},$$

defines the desired matches.

In this section we will consider problems with a series of matches on equality followed by one last (and therefore "final") match on inequality, more specific on the condition "less or equal". For example those could be one or more criteria defining a plan (like plan name and sex) followed be business time.

On the APL side the first columns of our matrices are to be matched on equality (=) whereas the last one is to be matched on "less or equal" (≤).

2.2 A sorting algorithm as solution

We can still use a (lexicographic) sort to find the desired match. Sorting the leading columns and grouping them is obviously necessary. Sorting additionally the last column brings the data in the right succession.

The grouping however from subsection subsection 1.4 has to be modified. Different dates belong to separate groups exactly when they come from the reference list. All subsequent data point dates with the same leading columns and up to the next reference date belong to the same group regardless of their value.

In our algorithm "match on equality" we just replace

```
BEG_I  ← 1 , ∨ / (1 ↓[1] MAT) ≠ (¯1 ↓[1] MAT) ⍝ inequality?
```

and obtain the following.

Algorithm "match on final inequality ≤"

```
...
V ← ⍋ MAT , ROW ⍝ sorting vector
...
BEG_I ← 1 , ∨ / (1 ↓[1] V) ≠ (¯1 ↓[1] (V ← ¯1 ↓[2] MAT))
    ⍝ inequality in leading columns?
I ← (1 ↓ V) ≠ (¯1 ↓ (V ← , ¯1 ↑[2] MAT))
    ⍝ inequality in final column?
REF_I ← NO_MATCH > ROW
    ⍝ mark references
BEG_I ← BEG_I  ∨  (1  ,  I ⍲ 1 ↓ REF_I)
    ⍝ final inequalities count at reference only
...
```

The sort is unchanged and only given for better comparison with the following subsections.

2.3 Strict inequality

In some cases a strict inequality may be needed. To obtain a match where the last property (column) is defined by a subset like

$$\mathcal{R}_j = \{r_i = r_{max}(d_j)\} \subset \mathcal{R}\,, \quad r_{max}(d_j) = \max\{r_i; r_i < d_j\},$$

the same technique may be used.

The only necessary change to the algorithm "match on equality" additionally to the one used to obtain the algorithm "match on final inequality ≤" are

- to replace

  ```
  V ← ⍋ MAT , ROW    ⍝ sorting vector
  ```

  with a sort that puts elements of the reference *after* those of the data if all columns are identical and

- to recognize a reference preceded by an identical data point as a group begin.

One possible (though not the most effective) solution is the following.

Algorithm "match on final inequality <"

```
...
V ← ⍋ MAT , (NO_MATCH > ROW) ,[1.5] ROW ⍝ sorting vector
...
BEG_I ← 1 , ∨ / (1 ↓[1] V) ≠ (¯1 ↓[1] (V ← ¯1 ↓[2] MAT))
    ⍝ inequality in leading columns?
I ← (1 ↓ V) ≠ (¯1 ↓ (V ← , ¯1 ↑[2] MAT))
    ⍝ inequality in final column?
REF_I ← NO_MATCH > ROW ⍝ mark references
BEG_I ← BEG_I  ∨  (1  ,  (I ^ 1 ↓ REF_I)
    ∨  ((1 ↓ REF_I)  ^  (¯1 ↓ ~ REF_I)))
        ⍝ final inequalities count at reference only,
          final equalities if after data and at reference
...
```

We emphasize that the algorithm works in that form only for the special case of one, final inequality discussed here.

2.4 Other relations

To complete the discussion on matching with one, final inequality we work out the conditions "greater or equal" and "greater". Those problems are complements to the ones discussed in subsections 2.2 and 2.3.

They can be tracked either by reversing the sort (using sort down through ⍒) or by using the *last* element of each group. We use the first option to get algorithms that are as easily comparable as possible.

We obtain the following by replacing the sort in the algorithm "match on final inequality ≤".

Algorithm “match on final inequality ≥”

```
...
V ← ⍒ MAT , (-ROW) ⍝ sorting vector
...
BEG_I ← 1 , ∨ / (1 ↓[1] V) ≠ (¯1 ↓[1] (V ← ¯1 ↓[2] MAT))
    ⍝ inequality in leading columns?
I ← (1 ↓ V) ≠ (¯1 ↓ (V ← , ¯1 ↑[2] MAT))
    ⍝ inequality in final column?
REF_I ← NO_MATCH > ROW
    ⍝ mark references
BEG_I ← BEG_I  ∨  (1  ,  I ⍲ 1 ↓ REF_I)
    ⍝ final inequalities count at reference only
...
```

Similarly we obtain the following by replacing the sort in the algorithm ”match on final inequality <”.

Algorithm “match on final inequality >”

```
...
V ← ⍒ MAT , (NO_MATCH ≤ ROW) ,[1.5] (-ROW) ⍝ sorting vector
...
BEG_I ← 1 , ∨ / (1 ↓[1] V) ≠ (¯1 ↓[1] (V ← ¯1 ↓[2] MAT))
    ⍝ inequality in leading columns?
I ← (1 ↓ V) ≠ (¯1 ↓ (V ← , ¯1 ↑[2] MAT))
    ⍝ inequality in final column?
REF_I ← NO_MATCH > ROW
    ⍝ mark references
BEG_I ← BEG_I  ∨  (1  ,  (I ⍲ 1 ↓ REF_I)
    ∨  ((1 ↓ REF_I)  ⍲  (¯1 ↓ ~ REF_I)))
        ⍝ final inequalities count at reference only,
          final equalities if after data and at reference
...
```

We have created four very similar algorithms to find matches on four inequalities. The whole burden of the solutions rests on the strong shoulders of the brave Grade Up and Grade Down sort primitives ⍋ and ⍒!

3 Matching with “initial” inequality

This section modifies the problem of section 2 and discusses different kinds of matching in a simple setting. The sorting algorithm “match on equality” of subsection 1.4 can mostly still be used as the basis of a solution. In one case however the limits of the normal sort are reached.

3.1 Cases with initial inequality

In some cases the first property of the desired match cannot be expected to be an exact one. The most common example in German health insurance is the

business time of a "premium capping scheme". It is not important what the latter exactly is. The point of interest is, that on certain dates $r_i^{(1)}$ a capping scheme is devised as a global property for all plans.

Each capping scheme contains instructions for some, but not all, plans. To answer the question "are the premiums of person j to be capped at time $d_j^{(1)}$", the (last) valid capping scheme has to be found and analysed. Should it contain instructions for the plans of the person they are to be used. Otherwise no capping is done.

Mathematically speaking our reference list is a set $\mathcal{R}$ of (distinct) pairs $r = (r^{(1)}, r^{(2)})$ of business times and plans. For a person j with plan $d_j^{(2)}$ at business time $d_j^{(1)}$ we are interested in the subset $\mathcal{R}(d_j)$ defined as

$$r_{max}^{(1)}(d_j) = \max\{r^{(1)}; r = (r^{(1)}, r^{(2)}) \in \mathcal{R} \text{ and } r^{(1)} \le d_j^{(1)}\},$$
$$\mathcal{R}(d_j) = \{r; r^{(1)} = r_{max}^{(1)}(d_j) \text{ and } r^{(2)} = d_j^{(2)}\} \subset \mathcal{R}.$$

In this section we will consider problems with one first (and therefore "initial") match on inequality, more specific on the condition "less or equal", followed by one match on some relation.

On the APL side the first column of our matrices is to be matched on "less or equal" ($\le$) whereas the second (and last) one is to be matched on some relation (= or $\le$).

3.2 The first type of match on initial inequality

We start with a match of the type we just described. We want to use a sorting algorithm. There does not seem to exist a single sort that would allow us to subsequently group the rows as necessary.

To see this consider `REF ← 2 2 ⍴ 1 1  2 0` and `DAT ← 3 2 ⍴ 3 1  4 0 5 1`. We expect *no* match for the first and third row of the data, the result should be `IND ← 3 2 3`. Neither a sort after the first column, the second one or a combination of both delivers the right groups.

Without a formal proof of correctness we give a solution based on the algorithm "match on final inequality ≤" but equipped with a loop through the columns. It is relatively clear that it produces the desired result.

Algorithm "example of match on initial inequality"

```
IND ← (1 ↑ ⍴DAT) ⍴ (NO_MATCH ← 1 + 1 ↑ ⍴REF)
    ⍝ result is 4 byte integer

MAT ← REF ,[1] DAT ⍝ catenate
ROW ← ⍳ 1 ↑ ⍴MAT   ⍝ member numbers

V ← ⍋ MAT , ROW ⍝ first sorting vector
MAT ← MAT[V;]   ⍝ sort matrix
ROW ← ROW[V ]   ⍝ sort member numbers

I ← (1 ↓ V) ≠ (¯1 ↓ (V ← MAT[;1])) ⍝ inequality in first column?
REF_I ← NO_MATCH > ROW              ⍝ mark references
```

```
BEG_I ← 1  ,  I ⍲ 1 ↓ REF_I
    ⍝ inequalities count at reference only


V ← ⍋ (+ \ BEG_I) , MAT[;,2] , ROW ⍝ second sorting vector
MAT ← MAT[V;]                      ⍝ sort matrix
ROW ← ROW[V ]                      ⍝ sort member numbers
                                   ⍝ BEG_I needs no sorting

I ← (1 ↓ V) ≠ (¯1 ↓ (V ← MAT[;2]))
    ⍝ inequality in second column?
BEG_I ← BEG_I  ∨  1 , I
    ⍝ all inequalities count

DAT_I ← NO_MATCH ≤ ROW ⍝ mark data

V ← BEG_I / ⍳ 1 ↑ ⍴MAT                  ⍝ begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ⍴MAT))  -  V ⍝ members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
    ← DAT_I / MAT_NR / NO_MATCH ⌊ BEG_I / ROW
        ⍝ member number of first match in reference
```

3.3 A different type of match on initial inequality

At the beginning of the section we have described some actuarial data (capping schemes) and its interpretation. However a second interpretation is possible. For a person j with plan $d_j^{(2)}$ at business time $d_j^{(1)}$ we could instead first build the subset

$$\mathcal{R}_0 = \{r; r^{(1)} \le d_j^{(1)} \text{ and } r^{(2)} = d_j^{(2)}\} \subset \mathcal{R} \tag{3}$$

and based on that select

$$r^{(1)}_{max}(d_j) = \max\{r^{(1)}; r = (r^{(1)}, r^{(2)}) \in \mathcal{R}_0 \text{ and } r^{(1)} \le d_j^{(1)}\},$$
$$\mathcal{R}(d_j) = \{r; r^{(1)} = r^{(1)}_{max}(d_j) \text{ and } r^{(2)} = d_j^{(2)}\} \subset \mathcal{R}_0.$$

The resulting matches are *not* the ones obtained before. In the example $\mathcal{R} = \{(1,1),(2,0)\}$ and $D = \{(3,1),(4,0),(5,1)\}$ there is now a match for *all* data points, namely $(1,1)$ for the first and third. The procedure does not expect or respect "holes" in data for subsequent business dates.

There is one important technical reason to use this interpretation of "match". For *one* data point it allows a very effective implementation as (static) SQL[3], something like

```
SELECT BUS_TIME, PLAN
FROM CAPPING_SCHEMES
WHERE BUS_TIME <= #B_T AND PLAN = #P
```

[3]More precisely the statement delivers an unambiguous result if there is only one match in the reference. Normally the data model will guarantee this. If not, the assumption that `ORDER BY` is stable must be added to achieve uniqueness.

```
ORDER BY BUS_TIME DESC, PLAN ASC
FETCH FIRST 1 ROWS ONLY
```

In this (very) special case we can switch the columns and use the algorithm "match on final inequality ≤" on the APL side to obtain the same result. This simple example also shows that data modelling must take such questions as the "right" order of the columns and the desired kind of match into account — but that should be self-evident.

3.4 A practically not very important match type

We introduce another match type. It has no obvious practical applications for actuarial simulations in German health insurance, but it puts the two types we already encountered in perspective. It imposes a stronger condition on matches — and will surely have some application!

We consider a very common setting. There are some properties (like insurance number, plan et cetera) that define each policy of a person. Each of them "lives" in system as well as in business time (in the language of DB2 bi-temporal tables).

We ignore the first properties and try to match a (single) bi-temporal data point $d = (d^{(1)}, d^{(2)})$, $\mathcal{D} = \{d\}$, with a (single, whole) policy as reference list $\mathcal{R}$. As the subset of matches we define

$$r^{(k)}_{max}(d) = \max\{r^{(k)}; r = (r^{(1)}, r^{(2)}) \in \mathcal{R} \text{ and } r^{(k)} \le d^{(k)}\}, \quad k = 1, 2,$$
$$\mathcal{R}(d) = \{r; r^{(1)} = r^{(1)}_{max}(d) \text{ and } r^{(2)} = r^{(2)}_{max}(d)\} \subset \mathcal{R}.$$

This may not seem "right" (and is probably not the common interpretation of system and business time!), but mathematically it is as sound as the other types of match on inequality we discussed...

On the APL side we can find (each) $r^{(k)}_{max}(d)$ through a search for a match on one column. Because the additional requirement to find the first match defines a unique r for each d we can then search and match $(r^{(k)}_{max}(d))_k$ on equality. We can use an algorithm like the following.

Algorithm "example of match on two ≤-inequalities"

```
IND ← (1 ↑ ⍴DAT) ⍴ (NO_MATCH ← 1 + 1 ↑ ⍴REF)
    ⍝ result is 4 byte integer

MAT ← REF ,[1] DAT ⍝ catenate
ROW ← ⍳ 1 ↑ ⍴MAT   ⍝ member numbers


MATCH_I ← (1 ↑ ⍴DAT) ⍴ 1 ⍝ all data points may have matches

:FOR COL :IN ⍳ 1 ↓ ⍴MAT
    MATP ← MAT[;COL] ⍝ relevant column

    V ← ⍋ MATP ,[1.5] ROW ⍝ sorting vector
    MATP ← MATP[V]        ⍝ sort column
```

```
    ROWP ← ROW [V]          ⍝ sort member numbers

    I ← (1 ↓ MATP) ≠ (¯1 ↓ MATP) ⍝ inequality?
    REF_I ← NO_MATCH > ROWP       ⍝ mark references
    BEG_I ← 1  ,  I ⌈ 1 ↓ REF_I
        ⍝ inequalities count at reference only

    DAT_I ← NO_MATCH ≤ ROWP ⍝ mark data

    V ← BEG_I / ⍳ ⍴MATP                    ⍝ begin of each group
    MAT_NR ← (1 ↓ V , (1 + ⍴MATP))  -  V ⍝ members per group

    MAT[DAT_I / ROWP ; COL] ← DAT_I / MAT_NR / BEG_I / MATP
        ⍝ replace data with extremal values
    V ← (1 - NO_MATCH) + DAT_I / ROWP
        ⍝ member numbers of data points
    I ← NO_MATCH > DAT_I / MAT_NR / NO_MATCH ⌊ BEG_I / ROWP
        ⍝ extremal value found?
    MATCH_I[V] ← MATCH_I[V] ⌈ I
        ⍝ may still find match?
:ENDFOR

V ← ⍋ MAT , ROW ⍝ final sorting vector
MAT ← MAT[V;]   ⍝ sort matrix
ROW ← ROW[V ]   ⍝ sort member numbers

BEG_I ← 1 , ∨ / (1 ↓[1] MAT) ≠ (¯1 ↓[1] MAT)
    ⍝ all inequalities count

DAT_I ← NO_MATCH ≤ ROW ⍝ mark data

V ← BEG_I / ⍳ 1 ↑ ⍴MAT                  ⍝ begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ⍴MAT))  -  V ⍝ members per group

IND[(1 - NO_MATCH) + MATCH_I / DAT_I / ROW]
    ← MATCH_I / DAT_I / MAT_NR / NO_MATCH ⌊ BEG_I / ROW
        ⍝ member number of first match in reference
```

3.5 A match problematic on the APL side

Now we consider the bi-temporal setting of subsection 3.4 but with a match of the type discussed in subsection 3.3. This is probably a common way to handle such tables.[4]

We consider a trivial reference list with two elements, $\mathcal{R} = \{(1,1),(2,3)\}$ and the single data point $\mathcal{D} = \{(3,2)\}$. The desired match is the first element of the reference list.

If we catenate the two sets APL side the result is (lexicographically) sorted. There is no obvious sort that would make our grouping algorithm work. An APL

[4]Except of course if they are bi-temporal DB2 tables in the specific technical meaning of IBM, in which case they are completely system handled!

algorithm could of course first try to find out which rows fulfil the necessary inequalities (those that define the analog of subset $\mathcal{R}_0$ in definition (3)).

The answer however depends on the data point. It is not obvious how an effective and fast algorithm could be build around this idea in APL.

3.6 Some remarks on matches and their different types

The standard matches (only on equality) are much easier than the ones containing inequalities. The predicative conditions on the columns may be examined one after another in an arbitrary order (commutativity). The may also be examined simultaneously.

When inequalities are required not only is the sequence in which they are examined important but also the type of the match used. We have encountered three match types. In each case there must be an explicit agreement on what data model is used — the data itself does not provide the necessary information.

On the APL side we have a similar situation. Each inequality brings more problems and makes the necessary algorithms (much) more complicated and slow. In subsection 3.5 we encountered a setting with no obvious APL solution.

Finding matches can be a goal itself. It can however also form the kernel of many other algorithms. In particular matches on inequality may be very useful even if the reference has keys defined in bounded intervals. It can be very useful to match the lower bound on inequality $\leq$ and then compare the data point with the upper bound — always assuming of course that the reference is overlap free!

4 Matching on set equality and relation

This section generalizes and formalizes the matching problem. It defines and discusses different kinds of matching on set equality and relation. It is shown that each matching problem corresponds to a disjoint dissection of the direct sum of reference and data list.

4.1 Setting

Let $\mathcal{S}_0$ be a set (taken to mean "with distinct elements") equipped with a relation $<$ under which it is completely ordered. One may think of this base set $\mathcal{S}_0$ being $\mathbb{Z}$ or $\mathbb{R}$. In fact we are mainly interested in their special subsets that are presentable on a digital computer.

We are interested in finite groups $\mathcal{S}$ of elements of $\mathcal{S}_0^n$, $n \geq 1$. We equip $\mathcal{S}$ with the set equality $=$ (in particular it may contain identical elements) and use $<, \leq, >$ and $\geq$ in the obvious way for each coordinate $s^{(k)}$ of each $s \in \mathcal{S}$.

In particular the quotient $\mathcal{S}/=$ is a set. We (try to) use "group", "subgroup" and "(group) member" when we refer to $\mathcal{S}$ and its members (!). In contrast we use "set", "subset" and "(set) element" when we refer to equivalence classes of members or elements of $\mathcal{S}_0^n$.

We fix a "dimension" n, select two corresponding groups and call the one "reference", denoted by $\mathcal{R}$, and the other "data", denoted by $\mathcal{D}$. We number the members of $\mathcal{R} = \{r_1, \ldots, r_i, \ldots, r_l\}$, $l = |\mathcal{R}|$, in an arbitrary but fixed way. This is done in order to achieve unique matches (to give the phrase "the *first* match" meaning).

4.2 Matches in general

We fix a vector $\leftrightsquigarrow = (\leftrightsquigarrow^{(k)})_k \in \{=, <, \leq, >, \geq\}^n$, of relations on $\mathcal{R}$, $\mathcal{D}$ and $\mathcal{M} = \mathcal{R} \oplus \mathcal{D}$. We use

$$m_i \leftrightsquigarrow m_{i'} \quad :\Leftrightarrow \quad m_i^{(k)} \leftrightsquigarrow^{(k)} m_{i'}^{(k)} \text{ for all } k = 1, \ldots, n,$$

for members of $\mathcal{M}$.

Associating pro forma the relation $=$ with the order of $<$ (for this relation "sort" is taken to mean "usual ascending sort") and extending naturally we define a corresponding lexicographic relation and order on ($\mathcal{R}$, $\mathcal{D}$ and) $\mathcal{M}$. We use

$$m_i \leftrightsquigarrow_{lex} m_{i'} \quad :\Leftrightarrow \quad m_i^{(k)} \leftrightsquigarrow^{(k)} m_{i'}^{(k)} \text{ for } k = \min\{k'; m_i^{(k')} \neq m_{i'}^{(k')}\},$$

for elements of $\mathcal{M}$.

Of course the defined relations cannot distinguish identical members

$$m_i = m_{i'} \quad \Rightarrow \quad m_i \leftrightsquigarrow m_{i'},\ m_i \leftrightsquigarrow_{lex} m_{i'},\ m_{i'} \leftrightsquigarrow m_i \text{ and } m_{i'} \leftrightsquigarrow_{lex} m_i$$

For each member $d \in \mathcal{D}$ we seek "the first match" $r \in \mathcal{R}$. More specifically we consider three types of match. In all cases we define filtered groups of members with the last one, $\mathcal{R}^T(d)$, containing exactly the matches for d. Those of course obey all relations with respect to d

$$\mathcal{R} \supseteq \mathcal{R}_0^T(d) \supseteq \mathcal{R}_1^T(d) \supseteq \cdots \supseteq \mathcal{R}_n^T(d) = \mathcal{R}^T(d)$$
$$\mathcal{R}^T(d) \subseteq \mathcal{R}_{\leftrightsquigarrow}(d) = \{r \in \mathcal{R}; r \leftrightsquigarrow d\}$$

More specifically $\mathcal{R}^T(d)$ contains at most one element of $\mathcal{S}_0^n$ (all contained members are identical). In all cases we select $r_i \in \mathcal{R}^T(d)$ with $i = \min\{i'; r_{i'} \in \mathcal{R}^T(d)\}$ as the desired (first) match and denote it by its number i. In case $\mathcal{R}^T(d) = \varnothing$ is empty we define $i = 1 + |\mathcal{R}|$.

4.3 Definition of some interesting types of match

The types of match that interest us are the following. With $\mathrm{extr}\{*\}$ we mean the obvious "extremal" value (=, min, max) under the relevant relation. A match is "weak" when it is sought only under references fulfilling a starting condition, "strong" otherwise.

In some cases the extremal values are "global" and may that fore be defined simultaneously whereas in others they are "local" and may not. Note that in all cases the filtered subgroups are defined successively, although in the case of global extrema the final subgroup can be defined directly. The selection of "the first match" is a possible add on in all cases.

weak local We start with $\mathcal{R}_0^{wk,loc}(d) = \mathcal{R}_{\leftrightsquigarrow}(d)$ and define successively for $k = 1, \ldots, n$

$$r_{extr}^{(k)}(d) = \mathrm{extr}\{r^{(k)}; r \in \mathcal{R}_{k-1}^{wk,loc} \text{ and } r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}\},$$
$$\mathcal{R}_k^{wk,loc}(d) = \{r \in \mathcal{R}_{k-1}^{wk,loc}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{wk,loc}.$$

strong local We start with $\mathcal{R}_0^{str,loc}(d) = \mathcal{R}$ and define successively for $k = 1, \ldots, n$

$$r_{extr}^{(k)}(d) = \mathrm{extr}\{r^{(k)}; r \in \mathcal{R}_{k-1}^{str,loc} \text{ and } r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}\},$$
$$\mathcal{R}_k^{str,loc}(d) = \{r \in \mathcal{R}_{k-1}^{str,loc}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{str,loc}.$$

weak global We start with $\mathcal{R}_0^{wk,gl}(d) = \mathcal{R}_{\leftrightsquigarrow}(d)$ and define successively for $k = 1, \ldots, n$

$$r_{extr}^{(k)}(d) = \mathrm{extr}\{r^{(k)}; r \in \mathcal{R}_{\leftrightsquigarrow}(d) \text{ and } r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}\} = \mathrm{extr}\{r^{(k)}; r \in \mathcal{R}_{\leftrightsquigarrow}(d)\},$$
$$\mathcal{R}_k^{wk,gl}(d) = \{r \in \mathcal{R}_{k-1}^{wk,gl}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{wk,gl}.$$

strong global We start with $\mathcal{R}_0^{str,gl}(d) = \mathcal{R}$ and define successively for $k = 1, \ldots, n$

$$r_{extr}^{(k)}(d) = \mathrm{extr}\{r^{(k)}; r \in \mathcal{R} \text{ and } r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}\},$$
$$\mathcal{R}_k^{str,gl}(d) = \{r \in \mathcal{R}_{k-1}^{str,gl}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{str,gl}.$$

The weak local match is also "lexicographic" because, apart from the obvious condition imposed by the relation $\leftrightsquigarrow$, it involves only a lexicographic sort. For each (but only *one*!) d the match[5] may be found with a SQL statement of the form

```
SELECT FIELD1, FIELD2, ..., FIELDN
FROM TABLE
WHERE FIELD1 <= #D1
AND   FIELD2  = #D2
...
AND   FIELDn >= #Dn
ORDER BY FIELD1 DESC, FIELD2 DESC, ..., FIELDn ASC
FETCH FIRST 1 ROWS ONLY
```

As a filler we have used $\leftrightsquigarrow = (\leq, =, \ldots, \geq)$ in the statement. We could of course have used "`FIELD2 ASC`" in the `ORDER BY` clause as well.

The strong local match is better suited to a non-technical setting. It assumes that there are some properties ordered after descending importance. Each one defines the subgroup (in a meaningful real world setting probably a subset...) of values relevant for the rest.

If all properties are "hole-less", meaning that each key used in the k-th position of one combination of keys appears at the k-th position of some combination of keys for *all* combinations of keys without their k-th position, then the difference with weak local matches disappears.

[5] More precisely the statement delivers a member of the subgroup of matches. Normally the data model will guarantee that there is exactly one such member. If not, the assumption that `ORDER BY` is stable must be added if we want to get the *same* member each time.

4.4 Analysis of the different types of match

4.4.1 Comparison and true inclusions

As already mentioned, groups of matches are either one element sets or empty because extremal values are unique. Let now $d \in \mathcal{D}$ be arbitrary but fixed.

If there exists $r \in \mathcal{R}^{str,gl}(d)$ all its coordinates are global extremal values. That fore they are also extremal in every subgroup. Furthermore $r \in \mathcal{R}_{\leftrightsquigarrow}(d)$ holds. It follows that $r \in \mathcal{R}_0^{wk,loc}(d) \cap \mathcal{R}_0^{str,loc}(d)$ also holds.

Inductively $r \in \mathcal{R}_k^{wk,loc}(d) \cap \mathcal{R}_k^{str,loc}(d)$ holds for each k. It follows that r is also an member of $\mathcal{R}^{wk,loc}(d)$ and $\mathcal{R}^{str,loc}(d)$. That fore $\mathcal{R}^{str,gl}(d) \subset \mathcal{R}^{wk,loc}(d) \cap \mathcal{R}^{str,loc}(d)$ holds.

Now let $r \in \mathcal{R}^{str,loc}(d)$ exist. In particular $r \in \mathcal{R}_{\leftrightsquigarrow}(d)$ holds. Because $r^{(1)}$ is globally extremal it is also with respect to $\mathcal{R}_0^{wk,loc}(d) = \mathcal{R}_{\leftrightsquigarrow}(d)$ and so $r \in \mathcal{R}_1^{wk,loc}(d)$ also holds.

Inductively the same argument can be used and $r \in \mathcal{R}_k^{wk,loc}(d)$ holds for each k. It follows that r is also an member of $\mathcal{R}^{wk,loc}(d)$. That fore $\mathcal{R}^{str,loc}(d) \subset \mathcal{R}^{wk,loc}(d)$ holds.

Now let $r \in \mathcal{R}^{str,gl}(d)$ (again) exist. In particular $r \in \mathcal{R}_{\leftrightsquigarrow}(d)$ holds. Because each $r^{(k)}$ is globally extremal it is also with respect to $\mathcal{R}_0^{wk,gl}(d) = \mathcal{R}_{\leftrightsquigarrow}(d)$ and so $r \in \mathcal{R}^{wk,gl}(d)$ also holds.

Finally let $r \in \mathcal{R}^{wk,gl}(d)$ exist. As each $r^{(k)}$ is globally extremal with respect to $\mathcal{R}_0^{wk,loc}(d) = \mathcal{R}_{\leftrightsquigarrow}(d)$ it is also with respect to $\mathcal{R}_k^{wk,loc}(d)$ and so $r \in \mathcal{R}^{wk,loc}(d)$ also holds.

For each $d \in \mathcal{D}$ have established

$$\mathcal{R}^{wk,loc}(d) \supseteq \mathcal{R}^{str,loc}(d) \supseteq \mathcal{R}^{str,gl}(d)$$
$$\mathcal{R}^{wk,loc}(d) \supseteq \mathcal{R}^{wk,gl}(d) \supseteq \mathcal{R}^{str,gl}(d).$$

Now let T and T' be two different types of match, $d \in \mathcal{D}$ arbitrary but fix with a match of *both* types, $\mathcal{R}^T(d) \neq \varnothing$ and $\mathcal{R}^{T'}(d) \neq \varnothing$. We recall the fact that $\mathcal{R}^T(d) \cap \mathcal{R}^{T'}(d) \subseteq \mathcal{R}_{\leftrightsquigarrow}(d)$ and use it repeatedly.

If $\mathcal{R}_0^T(d) = \mathcal{R}_0^{T'}(d)$ then of course the extremal values $r_{extr}^{T,(1)}(d) = r_{extr}^{T',(1)}(d)$ are equal. Else let $r_{extr}^{T,(1)}(d) \nleftrightsquigarrow r_{extr}^{T',(1)}(d)$ hold. But then there exists $r \in \mathcal{R}_{\leftrightsquigarrow}(d)$ with $r^{(1)} = r_{extr}^{T',(1)}(d)$ (by defintion of $\mathcal{R}^{T'}(d)$) *and* $r^{(1)} \nleftrightsquigarrow r_{extr}^{T,(1)}(d)$ (by definition of $\mathcal{R}_{\leftrightsquigarrow}(d)$), so $r_{extr}^{T',(1)}(d) \nleftrightsquigarrow r_{extr}^{T,(1)}(d)$ and that fore $r_{extr}^{T,(1)}(d) = r_{extr}^{T',(1)}(d)$ follows.

Now let $r_{extr}^{T,(k')}(d) = r_{extr}^{T',(k')}(d)$ hold for all $k' = 1, \ldots, k-1$. Similarly to the case $k = 1$ we can find an $r \in \mathcal{R}_{\leftrightsquigarrow}(d)$ with

$$\begin{aligned} r^{(k')} &= r_{extr}^{T,(k')}(d) = r_{extr}^{T',(k')}(d) \ , \qquad \text{for all } k' < k \\ r^{(k)} &= r_{extr}^{T',(k)}(d) \\ r^{(k)} &\nleftrightsquigarrow r_{extr}^{T,(k)}(d) \end{aligned}$$

so $r_{extr}^{T',(k)}(d) \nleftrightsquigarrow r_{extr}^{T,(k)}(d)$ and then $r_{extr}^{T,(k)}(d) = r_{extr}^{T',(k)}(d)$ follow.

For each $d \in \mathcal{D}$ and each pair T and T' of match types we have established

$$\mathcal{R}^T(d) \neq \varnothing \quad \text{and} \quad \mathcal{R}^{T'}(d) \neq \varnothing \qquad \Rightarrow \qquad \mathcal{R}^T(d) = \mathcal{R}^{T'}(d).$$

4.4.2 False inclusions

Evidently *neither* the inverse *nor* additional inclusions are true. Some absolutely trivial examples are the following:

- $\mathcal{R} = \{(1,1),(2,3)\}$, $d = (3,2)$ and $\leftrightsquigarrow = (\leq,\leq)$. Here $\mathcal{R}^{wk,loc}(d) = \{(1,1)\}$ whereas $\mathcal{R}^{str,loc}(d) = \varnothing$.
- $\mathcal{R} = \{(3,0),(0,3)\}$, $d = (4,4)$ and $\leftrightsquigarrow = (\leq,\leq)$. Here $\mathcal{R}^{str,loc}(d) = \{(3,0)\}$ whereas $\mathcal{R}^{str,gl}(d) = \varnothing$.
- $\mathcal{R} = \{(1,2),(2,1)\}$, $d = (3,2)$ and $\leftrightsquigarrow = (\leq,\leq)$. Here $\mathcal{R}^{wk,loc}(d) = \{(2,1)\}$ whereas $\mathcal{R}^{wk,gl}(d) = \varnothing$.
- $\mathcal{R} = \{(3,0),(5,3)\}$, $d = (4,4)$ and $\leftrightsquigarrow = (\leq,\leq)$. Here $\mathcal{R}^{wk,gl}(d) = \{(3,0)\}$ whereas $\mathcal{R}^{str,gl}(d) = \varnothing$.
- $\mathcal{R} = \{(1,1),(2,3)\}$, $d = (3,2)$ and $\leftrightsquigarrow = (\leq,\leq)$. Here $\mathcal{R}^{wk,gl}(d) = \{(1,1)\}$ whereas $\mathcal{R}^{str,loc}(d) = \varnothing$.
- $\mathcal{R} = \{(1,2),(2,1)\}$, $d = (3,2)$ and $\leftrightsquigarrow = (\leq,\leq)$. Here $\mathcal{R}^{str,loc}(d) = \{(2,1)\}$ whereas $\mathcal{R}^{wk,gl}(d) = \varnothing$.

4.4.3 "Commutativity" of relations

We now analyse what happens if we work through $k \in \{1,\ldots,n\}$ in an order different than the natural one. Do the resulting matches $\mathcal{R}^T(d)$ stay the same? In some cases the order of the coordinates is important and in others not.

weak local The definition of $\mathcal{R}_0^{wk,loc}(d) = \mathcal{R}_{\leftrightsquigarrow}(d)$ does not use the order of the coordinates. Moreover (if not empty) all its members have the same $r^{(k)}$ for each k with $\leftrightsquigarrow^{(k)}$ an equality. Therefore the corresponding extremal values are fixed and equalities may be processed in an arbitrary order *before* all inequalities. This freedom includes the process of building $\mathcal{R}_0^{wk,loc}(d)$ itself.

On the other hand the trivial example $\mathcal{R} = \{(3,2),(4,4)\}$, $d = (3,4)$ and $\leftrightsquigarrow = (=,\leq)$ shows that it is not possible to completely process the inequalities (including during the building of $\mathcal{R}_0^{wk,loc}(d)$) *first*, because this would give $\mathcal{R}^{wk,loc}(d) = \varnothing$, whereas $\mathcal{R}^{wk,loc}(d) = \{(3,2)\}$ is the correct answer. Of course once $\mathcal{R}_0^{wk,loc}(d)$ has been build equalities may also be processed after inequalities.

Furthermore the example $\mathcal{R} = \{(0,3),(3,0)\}$, $d = (4,4)$ and $\leftrightsquigarrow = (\leq,\leq)$ shows that it is not possible to process inequalities in an arbitrary order, because a switch would give $\mathcal{R}^{wk,loc}(d) = \{(0,3)\}$, whereas $\mathcal{R}^{wk,loc}(d) = \{(3,0)\}$ is the correct answer.

strong local The examples $\mathcal{R} = \{(3,2),(4,4)\}$, $d = (3,4)$ and $\leftrightsquigarrow = (=,\leq)$ as well as $\mathcal{R} = \{(0,3),(3,0)\}$, $d = (4,4)$ and $\leftrightsquigarrow = (\leq,\leq)$ we just used for weak local matches also show that it is neither possible to switch equalities and inequalities nor to process inequalities in an arbitrary order when searching strong local matches.

weak global The definition of $\mathcal{R}_0^{wk,gl}(d) = \mathcal{R}_{\leftrightarrow}(d)$ does not use the order of the coordinates. Furthermore the extremal values are defined as global ones on $\mathcal{R}_0^{wk,gl}(d)$. They are thus of course independent of each other. Therefore $\mathcal{R}^{wk,gl}(d)$ is completely independent of the order in which the coordinates are processed.

strong global The extremal values are defined as global ones. They are thus of course independent of each other. Therefore $\mathcal{R}^{str,gl}(d)$ is completely independent of the order in which the coordinates are processed.

4.5 Disjoint dissections

Now we consider two members $d, d' \in \mathcal{D}$ and their associated filtered groups $\mathcal{R}_k^T(d)$ and $\mathcal{R}_k^T(d')$. For $k = 1$, and then inductively for all k, d and d' are associated either with the same extremal value or not. As a result either $\mathcal{R}_k^T(d) = \mathcal{R}_k^T(d')$ or $\mathcal{R}_k^T(d) \cap \mathcal{R}_k^T(d') = \varnothing$ holds.

If $d = d'$ then of course the first is the case. We get disjoint dissections of $\mathcal{D} = \biguplus \mathcal{D}_i^T$ and $\mathcal{R} = \biguplus \mathcal{R}_i^T$ of data points and associated matches. Of course there are subgroups $\mathcal{D}_i^T$ associated with $\varnothing$ as well as subgroups $\mathcal{R}_i^T$ without "interested" data points. The dissections are also disjoint dissections of the underlying sets.

Let as apply the result to the direct sum $\mathcal{M} = \mathcal{D} \oplus \mathcal{R}$. We have created a disjoint dissection of $\mathcal{M} = \biguplus \mathcal{M}_i^T$ such that the matches of each $d \in \mathcal{M}_i^T \cap \mathcal{D}$ are to be found in $\mathcal{M}_i^T \cap \mathcal{R}$ (as long as there are any).

The created dissection is (as already remarked in subsection 3.6) useful even if the reference values are not given as begin or end points but rather as intervals. The dissection finds (of course assuming that the correct data model is used...) the (only) potentially right interval using only one end and avoiding Cartesian products. The other end of the interval can then be easily used to check if the match is the desired (or else there is no match).

5 Algorithms for two kinds of match

In this section sorting algorithms to find strong (local and global) matches are given. Additionally possible implementations in APL are presented.

5.1 A sorting algorithm for strong local matches

5.1.1 Description

We now reproduce the dissection of the direct sum $\mathcal{M}$ for strong local matches introduced in section 4 (in particular subsection 4.3) via a sorting algorithm than can be implemented in APL. As a "base dissection" we use $\mathcal{M} = \mathcal{M}_1^{(0)}$ and create filtered dissections $\mathcal{M} = \biguplus \mathcal{M}_i^{(k)}$. In a last step we find the (index of the) first match (for each $d \in \mathcal{D}$).

In the APL algorithm we write down we use both ascending and descending sorting (through APL primitives). We try however to avoid a rather high run-time penalty for highly non-trivial sorting and much data shuffling. To achieve this some additional information must be processed.

It is of course possible (and faster) to use only ascending sorting. The presented code is however easier to read. It is also much easier to verify its correctness (the fact that it delivers the desired strong local matches).

5.1.2 Algorithm

The variables REF and DAT are matrices and represent of course $\mathcal{R}$ and $\mathcal{D}$ respectively, REL is an alphanumeric vector and stands for the relation $\leftrightsquigarrow$. We assume $\mathcal{S}_0 = \mathbb{Z}$, so that ⍋ respects the natural relation $<$. For $\mathcal{S}_0 = \mathbb{R}$ the comparison tolerance must possibly be set to 0, alphanumeric data can be processed similarly.

Algorithm "strong local match"

```
IND ← (1 ↑ ⍴DAT) ⍴ (NO_MATCH ← 1 + 1 ↑ ⍴REF)
    ⍝ result is 4 byte integer

MAT   ← REF ,[1] DAT ⍝ catenate
ROW   ← ⍳ 1 ↑ ⍴MAT   ⍝ member numbers


BEG_I ← 1 , ((¯1 + 1 ↑ ⍴MAT) ⍴ 0) ⍝ start with one group

:FOR COL :IN ⍳ 1 ↓ ⍴MAT
    GR_NR ← + \ BEG_I      ⍝ group number
    REF_I ← NO_MATCH > ROW ⍝ mark references
    :SELECT REL[COL]       ⍝ build next sorting vector
    :CASE '='
        V ← ⍋   GR_NR  , MAT[;COL] , (~REF_I) ,[1.5]   ROW
    :CASE '≤'
        V ← ⍋   GR_NR  , MAT[;COL] , (~REF_I) ,[1.5]   ROW
    :CASE '<'
        V ← ⍋   GR_NR  , MAT[;COL] ,   REF_I  ,[1.5]   ROW
    :CASE '≥'
        V ← ⍒ (-GR_NR) , MAT[;COL] ,   REF_I  ,[1.5] (-ROW)
    :CASE '>'
        V ← ⍒ (-GR_NR) , MAT[;COL] , (~REF_I) ,[1.5] (-ROW)
    :ENDSELECT
    MAT ← MAT[V;] ⍝ sort matrix
    ROW ← ROW[V ] ⍝ sort member numbers
                  ⍝ BEG_I needs no sorting

    I ← (1 ↓ V) ≠ (¯1 ↓ (V ← MAT[;COL])) ⍝ inequality?
    :SELECT REL[COL]
        ⍝ further dissect each subgroup
    :CASE     '='
        BEG_I ← BEG_I  ∨  (1 , I)
            ⍝ all inequalities count
    :CASELIST '≤≥'
        REF_I ← NO_MATCH > ROW
        BEG_I ← BEG_I  ∨  (1   ,   I ^ 1 ↓ REF_I)
```

```
            ⍝ inequalities count at reference only
    :CASELIST '<>'
        REF_I ← NO_MATCH > ROW
        BEG_I ← BEG_I  ∨  (1  ,  (I ⌈ 1 ↓ REF_I)
            ∨  ((1 ↓ REF_I)  ⌈  (¯1 ↓ ~ REF_I)))
               ⍝ inequalities count at reference only,
                 equalities if after data and at reference
    :ENDSELECT
:ENDFOR


V ← ⍋ (+ \ BEG_I) ,[1.5] ROW
    ⍝ ensure sort up and leading reference members
MAT ← MAT[V;]                ⍝ sort matrix
ROW ← ROW[V ]                ⍝ sort member numbers

DAT_I ← NO_MATCH ≤ ROW ⍝ mark data

V ← BEG_I / ⍳ 1 ↑ ⍴MAT                  ⍝ begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ⍴MAT))  -  V ⍝ members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
    ← DAT_I / MAT_NR / NO_MATCH ⌊ BEG_I / ROW
        ⍝ member number of first match in reference
```

5.1.3 Proof of correctness

In the main part of the algorithm the subgroups are created as defined in subsection 4.3. To achieve this the data is processed column-wise. It is sorted using:

`GR_NR` This part ensures that already defined subgroups are not teared apart and get not moved around. This also means, that `BEG_I` needs not to be sorted or reset, even if members in the subgroup get reshuffled.

`MAT[;COL]` This is the data proper. It is clear that it is necessary and sufficient to sort it in order to respect the relevant relations.

`REF_I` The marking of reference points ensures the desired behaviour whenever the column data proper is equal (meaning in particular that references and data points are indistinguishable in this column). In some of the cases the necessary information is also encoded in `ROW` and the variable could have been omitted.

`ROW` The use of the original member numbers (coded as rows in APL) guarantees that the sort is stable even if the algorithm underlying the primitive is not. This ensures that the first match lands at the beginning of its subgroup.

Then the already constructed subgroups are further dissected. A case by case analysis makes clear that the algorithm works:

= In this case each inequality defines a new group as it should be. Reference points come before equal data points due to the sorting used.

$\leq$ The maximum is unique (for each data point). That fore unequal reference points define new subgroups. Not so for data points, as long as they point to the same maximal reference value. The sorting ensures that reference points come before equal data points.

$<$ In contrast to $\leq$ data points must come *before* equal reference points, so they are preceded (only) by strictly smaller references. Additionally the transition between equal data and reference points must define a new subgroup.

$\geq$ Analog to $\leq$.

$>$ Analog to $<$.

So it is clear that the correct subgroups are created. The last part selects the first member of each group and uses it to define the first match of each $d \in \mathcal{D}$. To achieve this it ensures that reference points precede equal data points and uses the number of members per subgroup.

5.2 A sorting algorithm for strong global matches

5.2.1 Introduction

We now reproduce the dissection of the direct sum $\mathcal{M}$ for strong global matches introduced in section 4 (in particular subsection 4.3) via a sorting algorithm than can be implemented in APL. As a "base dissection" we use $\mathcal{M} = \mathcal{M}_1^{(0)}$.

We do not follow the definition of the dissection strictly but instead replace data points for inequalities with extremal values coordinate-wise. Based on those new values we seek matches on equality and directly create the final dissection $\mathcal{M} = \biguplus \mathcal{M}_i^{(n)}$ and find the (index of the) first match (for each $d \in \mathcal{D}$).

For the necessity of this last step consider $\mathcal{R} = \{(3,0),(0,3),(3,3)\}$, $\mathcal{D} = \{(4,4)\}$ and $\leftrightsquigarrow = (\leq, \leq)$. The search for the maximal value yields matches at the first (as well as the third) and second (as well as the third) member respectively. The desired overall match however is the third member (even more $\mathcal{R}^{str,gl}(d) = (3,3)$ contains only this one member).

It should be clear that the overall process is equivalent to the definition of strong global matches. We only sort data up (via ⍋). This means that when replacing values under the relations $\geq$ or $>$ be must use last members of groups (instead of first ones).

5.2.2 Algorithm

As in subsection 5.1 the variables `REF` and `DAT` represent $\mathcal{R}$ and $\mathcal{D}$ respectively and `REL` stands for the relation $\leftrightsquigarrow$. The remarks about $\mathbb{Z}$, $\mathbb{R}$ and alphanumeric data hold.

Algorithm "strong global match"

```
IND ← (1 ↑ ⍴DAT) ⍴ (NO_MATCH ← 1 + 1 ↑ ⍴REF)
    ⍝ result is 4 byte integer

MAT ← REF ,[1] DAT ⍝ catenate
```

```
ROW ← ⍳ 1 ↑ ⍴MAT   ⍝ member numbers


MATCH_I ← (1 ↑ ⍴DAT) ⍴ 1 ⍝ all data points may have matches

:FOR COL :IN ('=' ≠ REL) /  ⍳ 1 ↓ ⍴MAT
    ⍝ find extremal values at inequalities
    MATP ← MAT[;COL] ⍝ relevant column

    REF_I ← NO_MATCH > ROW ⍝ mark references
    :SELECT REL[COL]       ⍝ build next sorting vector
    :CASELIST '≤>'
        V ← ⍋ MATP , (~REF_I) ,[1.5] ROW
    :CASELIST '<≥'
        V ← ⍋ MATP ,   REF_I  ,[1.5] ROW
    :ENDSELECT
    MATP ← MATP[V] ⍝ sort column
    ROWP ← ROW [V] ⍝ sort member numbers

    I ← (1 ↓ MATP) ≠ (¯1 ↓ MATP) ⍝ inequality?
    REF_I ← NO_MATCH > ROWP ⍝ mark references
    :SELECT REL[COL] ⍝ dissect on one inequality
    :CASE '≤'
        GR_I ← 1  ,   (I ^  1 ↓ REF_I)
            ⍝ inequalities count at reference only
    :CASE '<'
        GR_I ← 1  ,  ((I ^  1 ↓ REF_I)
            ∨  ((1 ↓   REF_I)  ^  (¯1 ↓ ~ REF_I)))
                ⍝ inequalities count at reference only,
                  equalities if after data and at reference
    :CASE '≥'
        GR_I ←        (I ^ ¯1 ↓ REF_I)  ,  1
            ⍝ inequalities count at reference only
    :CASE '>'
        GR_I ←       ((I ^ ¯1 ↓ REF_I)
            ∨  ((1 ↓ ~ REF_I)  ^  (¯1 ↓   REF_I)))  ,  1
                ⍝ inequalities count at reference only,
                  equalities if after reference and at data
    :ENDSELECT

    DAT_I ← NO_MATCH ≤ ROWP ⍝ mark data

    V ← GR_I / ⍳ ⍴MATP ⍝ begin or end of each group
    :SELECT REL[COL]   ⍝ members per group
    :CASELIST '≤<'
        MAT_NR ← (1 ↓ V , (1 + ⍴MATP))  -  V
    :CASELIST '≥>'
        MAT_NR ← V  -  (0 , ¯1 ↓ V)
    :ENDSELECT
```

```
    MAT[DAT_I / ROWP ; COL] ← DAT_I / MAT_NR / GR_I / MATP
        ⍝ replace data with extremal values
    V ← (1 - NO_MATCH) + DAT_I / ROWP
        ⍝ member numbers of data points
    I ← NO_MATCH > DAT_I / MAT_NR / NO_MATCH ⌊ GR_I / ROWP
        ⍝ extremal value found?
    MATCH_I[V] ← MATCH_I[V] ^ I
        ⍝ may still find match?
:ENDFOR


V ← ⍋ MAT , ROW ⍝ final sorting vector
MAT ← MAT[V;]   ⍝ sort matrix
ROW ← ROW[V ]   ⍝ sort member numbers

BEG_I ← 1 , ∨ / (1 ↓[1] MAT) ≠ (¯1 ↓[1] MAT)
    ⍝ all inequalities count

DAT_I ← NO_MATCH ≤ ROW ⍝ mark data

V ← BEG_I / ⍳ 1 ↑ ⍴MAT                  ⍝ begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ⍴MAT))  -  V ⍝ members per group

IND[(1 - NO_MATCH) + MATCH_I / DAT_I / ROW]
    ← MATCH_I / DAT_I / MAT_NR / NO_MATCH ⌊ BEG_I / ROW
        ⍝ member number of first match in reference
```

5.2.3 Proof of correctness

There is not much to prove. One must only think each case (relation) through and make sure that the sort and the following definition of subgroups are the right ones for finding the wanted extremal value. We refer to subsection 5.1 for details.

The last step is to define the (final) dissection. More precisely the dissection on equalities must be further divided. At this point it is important to be sure that (all) references come either before or after (all) corresponding data points. That fore a mix of sort up and down must be used in the main part. The use of ROW ensures that our matches are first ones.

5.2.4 An alternative approach

There is another possible approach to a solution. It uses the fact, proven in subsection 4.4, that matches, as far as they exists, do not depend on their type. That fore we can try to find $\mathcal{R}^{str,gl}(d)$ starting with $\mathcal{R}^{str,loc}(d)$.

The algorithm "strong local match" presented in subsection 5.1 sorts reference and data in such a way, that the disjoint dissection necessary for strong local matches is created. Let $d \in \mathcal{D}$ be arbitrary but fix. It is contained in a subgroup $\mathcal{M}(d)$.

If and only if there exists a $r \in \mathcal{R} \cap \mathcal{M}(d)$ there is a strong local match ($\mathcal{R}^{str,loc}(d) \neq \varnothing$). Without loss of generality we assume that r is the first

match. Because of $\mathcal{R}^{str,loc}(d) \supseteq \mathcal{R}^{str,gl}(d)$ this is necessary but not sufficient for the existence of a strong global match ($\mathcal{R}^{str,gl}(d) \neq \varnothing$).

We indicate the potential for a match as $\mathbb{1}_0^{str,gl}(d)$. For each coordinate we successively examine if $r^{(k)}$ is also globally extremal. If not then there exists no match. In formulas

$$\mathbb{1}_k^{str,gl}(d) = \begin{cases} 1 & \text{if } k = 0 \text{ and } \mathcal{R}^{loc,gl}(d) \neq \varnothing \\ \mathbb{1}_{k-1}^{str,gl}(d) & \text{if } k \geq 1 \text{ and} \\ & \quad r^{(k)} = \operatorname{extr}\{(r')^{(k)}; r' \in \mathcal{R} \text{ and } (r')^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}\} \\ 0 & \text{else} \end{cases}$$

If and only if $\mathbb{1}_n^{str,gl}(d)$ is true there exists a strong global match ($\mathcal{R}^{str,gl}(d) \neq \varnothing$) and (then) r is the first match. We used a particular order to check the conditions of extremality but it is obviously arbitrary and not relevant for the end result. Furthermore each sort can clearly be applied to an arbitrary ordering of $\mathcal{M}$.

A possible algorithmic implementation in APL is obviously to use the algorithm "strong local match" with an additional sort for each column k as an add on. The latter is used to determine if $r^{(k)}$ is globally extremal with respect to d. However the presented algorithm "strong global match" is surely less circuitous and more efficient.

6 Tentative algorithms for problematic matches

This section outlines sorting algorithms for weak (local and global) matches. There is however no obvious way to implement them effectively using APL primitives. "Special sorts" are proposed as a solution to the problem.

6.1 A tentative sorting algorithm for weak local matches

6.1.1 Introduction

We now want to reproduce the dissection of $\mathcal{M}$ for weak local matches via a sorting algorithm than can be implemented in APL. As a "base dissection" we use $\mathcal{M} = \mathcal{M}_1^{(0)}$.

We want to follow the definition of the dissection. To achieve this we want to use a "special sort" of the group $\mathcal{M}$ to enforce that all $d \in \mathcal{D}$ which have the necessary relation to a specific $r \in \mathcal{R}$ (including extremality) and only those follow this r (directly). This given it is trivial to directly create the final dissection $\mathcal{M} = \biguplus \mathcal{M}_i^{(n)}$.

It should be clear that the overall process is equivalent to the definition of weak local matches. What is unclear is how to sort the data using the APL primitives ⍋ and ⍒.

6.1.2 Algorithm with a gap

As in subsection 5.1 the variables `REF` and `DAT` represent $\mathcal{R}$ and $\mathcal{D}$ respectively and `REL` stands for the relation $\leftrightsquigarrow$. The remarks about $\mathbb{Z}$, $\mathbb{R}$ and alphanumeric data hold.

We use the obvious fact (see also subsection 4.4) that equalities may be tested at the beginning of the algorithm. The intermediate dissection thus created is only a technical trick to separate those and has not much in common with the ones in the definition of the match.

We write SPEC_SORT as a place holder for a special type of sort. It uses a boolean flag vector in addition to the data proper. The algorithm sorts data in a way that all but creates the desired final dissection (all members are correctly "positioned" and only the beginning of each group must be determined).

To do this it must ensure that (all) references come either before or after (all) corresponding data points. That fore sort up and down may both by needed. Note that the resulting order of each sort is the basis of the next one but the subgroups corresponding to each sort are not and *cannot be* used subsequently.

Algorithm "weak local match"

```
IND ← (1 ↑ ⍴DAT) ⍴ (NO_MATCH ← 1 + 1 ↑ ⍴REF)
    ⍝ result is 4 byte integer

MAT ← REF ,[1] DAT ⍝ catenate
ROW ← ⍳ 1 ↑ ⍴MAT   ⍝ member numbers

COL_EQ   ← (I ← '=' = REL) / (V ← ⍳ ⍴REL) ⍝ separate equalities
COL_INEQ ← (I ← ~I) / V                   ⍝ and inequalities

V ← (2 / 1 ¯1) ['≤<≥>' ⍳ I / REL] ⍝ sign
REF_I ← NO_MATCH > ROW ⍝ mark references
V ← ⍋ MAT[;COL_EQ] , (V ×[2] MAT[;COL_INEQ])
    , (~REF_I) ,[1.5] ROW
        ⍝ preparatory sort, ensure lexicographic order
MAT ← MAT[V;] ⍝ sort matrix
ROW ← ROW[V]  ⍝ sort member numbers

BEG_I ← 1 , ∨ / (1 ↓[1] V) ≠ (¯1 ↓[1] (V ← MAT[;COL_EQ]))
    ⍝ inequality?
GR_NR ← + \ BEG_I ⍝ dissect at equalities

:FOR COL :IN COL_INEQ
    REF_I ← NO_MATCH > ROW ⍝ mark references
    :SELECT REL[COL] ⍝ build next sorting vector
    :CASE '≤'
        V ← SPEC_SORT ((  GR_NR  , MAT[;COL] , (~REF_I)
            ,[1.5]   ROW )  REF_I  'Up'  )
    :CASE '<'
        V ← SPEC_SORT ((  GR_NR  , MAT[;COL] ,   REF_I
            ,[1.5]   ROW )  REF_I  'Up'  )
    :CASE '≥'
        V ← SPEC_SORT (((-GR_NR) , MAT[;COL] ,   REF_I
            ,[1.5] (-ROW))  REF_I  'Down')
    :CASE '>'
        V ← SPEC_SORT (((-GR_NR) , MAT[;COL] , (~REF_I)
```

```
            ,[1.5] (-ROW))  REF_I  'Down')
    :ENDSELECT
    MAT ← MAT[V;] ⍝ sort matrix
    ROW ← ROW[V]  ⍝ sort member numbers
:ENDFOR

:IF 0 < ⍴COL_INEQ
    I ← ∨ / (1 ↓[1] V) ≠ (¯1 ↓[1] (V ← MAT[;COL_INEQ]))
        ⍝ inequality?
    REF_I ← NO_MATCH > ROW ⍝ mark references
    BEG_I ← BEG_I  ∨  (1  ,  (I ^ 1 ↓ REF_I)
        ∨  ((1 ↓ REF_I)  ^  (¯1 ↓ ~ REF_I)))
            ⍝ inequalities at reference only,
              equalities if after data and at reference
:ENDIF

DAT_I ← NO_MATCH ≤ ROW ⍝ mark data

V ← BEG_I / ⍳ 1 ↑ ⍴MAT                  ⍝ begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ⍴MAT))  -  V ⍝ members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
    ← DAT_I / MAT_NR / NO_MATCH ⌊ BEG_I / ROW
        ⍝ member number of first match in reference
```

6.2 Correctness of algorithm

6.2.1 Prerequisites for correctness

The algorithm completely separates equalities and inequalities. The first are trivial. We can therefore assume that we deal only with inequalities. The algorithm begins with a lexicographic sort (using `ROW` to ensure that at the end we get the desired first matches). It is necessary to keep the lexicographic order of the references upright until the end.

The end part is also easy. Assuming that

- each reference comes before all data points that belong in the same dissection and
- references are in lexicographic order

the data points are *immediately* preceded by the corresponding reference.

We have analyzed the building of dissections on inequalities in subsection 5.1. There we used an additional condition on data points followed by references in the case of strict inequalities.

One must observe that this is superfluous but not harmful in the case of non-strict inequalities: then such sequences never contain identical members. Therefore it can be used for all inequalities (as done in the proposed algorithm).

Let us further assume that `SPEC_SORT`

- sorts by exchanging consecutive pairs or members and

- respects flags by never exchanging a pair of members if the second one is flagged.

Those conditions are conservative and can be relaxed a bit. They are meant to use the references as a "scaffold" or in other words

- keep the reference members in a stable position relative to each other and
- allow data points to slide up past references as far as necessary but not more or the other way round.

We (that fore) flag the reference points. It must be noted that this "special sort" produces an order that does not depend only on the members but also on their starting order.

Further it must be noted, that for *every* $d \in \mathcal{D}$ *all* $r \in \mathcal{R}_{\leftrightsquigarrow}(d)$ precede d. This however does not mean that every $r \in \mathcal{R}$ which precedes a certain d is also contained in $\mathcal{R}_{\leftrightsquigarrow}(d)$. Such a sort does not exist in general.

As an elementary example consider $\leftrightsquigarrow = (\leq, \leq)$, $\mathcal{D} = \{(2,5),(4,3)\}$ and $\mathcal{R} = \{(1,1),(2,4),(3,3)\}$. We have $\mathcal{R}_{\leftrightsquigarrow}((2,5)) = \{(1,1),(2,4)\}$ and $\mathcal{R}_{\leftrightsquigarrow}((4,3)) = \{(1,1),(3,3)\}$. There is no way to sort $\mathcal{M}$ as desired.

6.2.2 Proof of correctness

Let $d, d' \in \mathcal{D}$ and $r, r' \in \mathcal{R}$ be arbitrary but fix.

data versus reference We want to establish

$$\{r \leftrightsquigarrow d\} \text{ or } \{\exists \tilde{r} \in \mathcal{R},\ \tilde{r} \leftrightsquigarrow d \text{ and } \tilde{r} \text{ ends up after } r \text{ and before } d\}$$

$$\Leftrightarrow \quad r \text{ ends up before } d.$$

We implicitly use the restriction about consecutive pairs on the sort algorithm when we directly compare r with d.

1. We first want to prove that if $r \leftrightsquigarrow d$ holds then r ends up before d. If $r \leftrightsquigarrow d$ holds then $r \leftrightsquigarrow_{lex} d$ also holds. That fore r starts *before* d because of the preliminary sort.
   In each step $r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}$ is true. Should is be checked (should the members be compared), at strict inequalities this determines that r stays before d. At non-strict inequalities the use of the variable `REF_I` as part of the sort ensures that this is the case even if $r^{(k)} = d^{(k)}$ holds. It follows that r stays before d. Note that especially for $r \in \mathcal{R}^{wk,loc}(d)$ the relation $r \leftrightsquigarrow d$ holds per definition.
2. Next let r be such that $r \leftrightsquigarrow_{lex} d$ does *not* hold. Because of the nature of lexicographic sort $d \leftrightsquigarrow_{lex} r$ holds and r starts *after* d. The condition on flags ensures that this stays so.
3. Next we select a r such that $r \leftrightsquigarrow d$ does *not* hold but $r \leftrightsquigarrow_{lex} d$ does. r starts before d because of the preliminary sort. Let k be minimal with $r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}$ false. Up to the $(k-1)$-th sort r stays before d. Let us assume that $r^{(k)}$ *is* compared with $d^{(k)}$.
   If $\leftrightsquigarrow^{(k)}$ is a non-strict inequality, then $r^{(k)} \neq d^{(k)}$ and the corresponding sort exchanges the pair (this is allowed as d is *not* flagged). If $\leftrightsquigarrow^{(k)}$ is strict then the use of `REF_I` ensures the switch.

After that *no* switch can occur because r is flagged. This means that r ends up after d.

4. Finally if in the previous setting $r^{(k)}$ *is not* compared with $d^{(k)}$ this means that another member $\tilde{r} \in \mathcal{R}$ blocks the comparison. But $d \leftrightsquigarrow_{lex} \tilde{r}$ cannot hold, as we have shown, and for the combination $\tilde{r} \leftrightsquigarrow_{lex} d$ true but $\tilde{r} \leftrightsquigarrow d$ false we can repeat the same argument.

   So $\tilde{r} \leftrightsquigarrow d$ must hold and we have found the desired intermediate reference member.

connectedness of $\mathcal{R}^{wk,loc}(d)$ We now want to show that all members of each subgroup $\mathcal{R}^{wk,loc}(d)$ end up in consecutive (uninterrupted) order. We assume $\mathcal{R}^{wk,loc}(d) \cap \mathcal{R}^{wk,loc}(d') = \varnothing$, $r \in \mathcal{R}^{wk,loc}(d)$ and $r \neq r'$.

1. We first note that $r \leftrightsquigarrow_{lex} r'$ is equivalent to r starting before r' because of the preliminary sort. Reference points are however flagged and never exchanged. That fore

   $$r \leftrightsquigarrow r'_{lex} \quad \Leftrightarrow \quad r \text{ ends up before } r'$$

   also holds.

2. Now let $r \leftrightsquigarrow_{lex} r'$ hold. Per assumption $r \leftrightsquigarrow d$ also holds, so r ends up before r' and d. Let r' end up before d. Then, as we have established, $r' \leftrightsquigarrow d$ holds (or at least we can replace it without loss of generality with an appropriate $\tilde{r}$).

   Let k be the first coordinate with $r^{(k)} \leftrightsquigarrow (r')^{(k)}$. Because $r \neq r'$ we may assume that this is a strict inequality. But this means that $r^{(k)}$ is *not* extremal with respect to d, in contradiction to $r \in \mathcal{R}^{wk,loc}(d)$. It follows that r, d and r' end up in this order.

3. If $r' \leftrightsquigarrow_{lex} r$ holds then of course r' ends up before r and therefore d.

4. We finally examine d'. If $\mathcal{R}^{wk,loc}(d') = \varnothing$ is empty, then d' must end up before *all* members of $\mathcal{R}$ and therefore before r. If it is not then we may without loss of generality assume that $r' \in \mathcal{R}^{wk,loc}(d')$. Then d' ends up after r', which ends up after d.

We have proved that our algorithm dissects $\mathcal{M}$ correctly. So the only think that remains is to create `SPEC_SORT`!

6.3 Special sort algorithm with scaffold

We present some algorithms that can be used for the special sort required to effectively find weak local matches.

6.3.1 Basic facts about sort algorithms

We consider a primitive algorithm (lets call it "Primitive Sort"), Bubble Sort, Merge Sort and Natural Merge Sort (based on https://de.wikipedia.org/wiki/Sortierverfahren) as sort algorithms. The primitive algorithm is a dumb version of Bubble Sort that loops through the *whole* array until it gets stationary. Furthermore all those algorithms are stable.

It is clear that Primitive Sort and Bubble Sort fulfil the requirements laid out in subsection 6.2. It is also clear that (and how) they can be modified to respect the condition on flags.

Merge Sort splits the data into smaller lists and repeatedly zips together consecutive lists. Natural Merge Sort is the same expect for the definition of the lists. So both algorithms violate the strict criteria (about comparing consecutive pairs of members) we laid out in subsection 6.1.

6.3.2 A false special sort algorithm

We want to show that a modification of Merge Sort can be used as a special sort. To better illustrate the problem we introduce a *false* start. Let us consider the following adaption to (Natural) Merge Sort:

1. We start with two lists, `S` with the starting and `E` with the ending members, and an empty result.
2. As long as both lists are not empty and the first member of `E` is *not* flagged we zip members from the lists together as in Merge Sort and append them to the result.
3. Then we append the rest of `S` to the result.
4. Then we append the rest of `E` to the result.

This preserves the stability of the relative position of flagged members and the property of not-flagged members to slide up past flagged ones but not the other way round. The non-flagged members can however slide to "high".

To see this consider the lists `S` = $\{0, 1, 3, 0, 1, 3\}$ and `E` = $\{2, 4\}$. The first and third as the well as the second members respectively are flagged. The first member of `E` lands at the third position of the result.

In the setting that interests us however the members of `S` belong to two subgroups defined by a previous step of the dissection. The obtained result destroys the dissection because it groups the data point together with a reference that does not have the right (extremal) value at a coordinate already processed.

6.3.3 A modification of (Natural) Merge Sort as special sort

Instead we propose the following modified (Natural) Merge Sort:

1. We start with two lists, `S` with the starting and `E` with the ending members, and an empty result.
2. As long as both lists are not empty and `E` contains a flagged member we append the result to the *last* member of `E`.
3. As long as both lists are not empty we zip trailing members from the lists together as in Merge Sort and append the result to them. In particular we reverse the comparison of two members in a way that preserves stability.
4. Then we append the result to the rest of `S`.
5. Then we append the result to the rest of `E`.

It is clear that the "inverse zip" as such does not alter the results of Merge Sort.

To establish assertions about the results of our special sort algorithm it is always enough to consider one member in `S` and one in `E` at some step of the process. In all other cases the relative positions of members are preserved.

Because the end of `E`, beginning with the first flagged member of `E`, lands at the end and the rest is compared normally the sort

- preserves the relative position of flagged members,
- allows non-flagged members to slide up past flagged ones but not the other way round and
- ensures that the position of two members relative to each other is only changed after comparing them.

6.3.4 Modified proof of correctness

In particular the sort does not (in contrast to the normal Merge Sort) assume that each list is "correctly" sorted. Using this we can slightly modify and generalize the proof of correctness given in subsection 6.2 (check the latter for further details).

We write the full proof down again, although in an abbreviated form. Point 1 of "data versus reference" is (only) formally modified; point 3 and 4 use the property that members are only switched around after comparison; the rest is the same.

Let $d, d' \in \mathcal{D}$ and $r, r' \in \mathcal{R}$ be again arbitrary but fix.

data versus reference We want to establish

$$\{r \leftrightsquigarrow d\} \text{ or } \{\exists \tilde{r} \in \mathcal{R},\ \tilde{r} \leftrightsquigarrow d \text{ and } \tilde{r} \text{ ends up after } r \text{ and before } d\}$$

$$\Leftrightarrow \quad r \text{ ends up before } d.$$

1. We first assume $r \leftrightsquigarrow d$ holds and prove that r ends up before d. Then r starts *before* d and in each step $r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}$ holds.
   The critical step comes when $r^{(k)}$ is a member of `S` and $d^{(k)}$ one of `E`. But this means at worst a normal comparison of values, the order is preserved.
2. Next let r be such that $r \leftrightsquigarrow_{lex} d$ does *not* hold. Then r starts *after* d and the condition on flags ensures that this stays so.
3. Next we select a r such that $r \leftrightsquigarrow d$ does *not* hold but $r \leftrightsquigarrow_{lex} d$ does. Let k be minimal with $r^{(k)} \leftrightsquigarrow^{(k)} d^{(k)}$ false and let us assume that $r^{(k)}$ *is* compared with $d^{(k)}$.
   We consider the step where $r^{(k)}$ is a member of `S` and $d^{(k)}$ one of `E`. A comparison means a normal one, as d is not flagged. The two members are switched.
   After that *no* switch can occur because r is flagged. This means that r ends up after d.
4. Finally if in the previous setting $r^{(k)}$ *is not* compared with $d^{(k)}$ this means that another member $\tilde{r} \in \mathcal{R}$ blocks the comparison. Then $\tilde{r} \leftrightsquigarrow d$ must hold and we have found the desired intermediate reference member.

connectedness of $\mathcal{R}^{wk,loc}(d)$ We now want to show that all members of each subgroup $\mathcal{R}^{wk,loc}(d)$ end up in consecutive (uninterrupted) order. We assume $\mathcal{R}^{wk,loc}(d) \cap \mathcal{R}^{wk,loc}(d') = \varnothing$, $r \in \mathcal{R}^{wk,loc}(d)$ and $r \neq r'$.

1. We first note that $r \leftrightsquigarrow_{lex} r'$ is equivalent to r starting before r' and that reference points are never exchanged. That fore

$$r \leftrightsquigarrow r'_{lex} \quad \Leftrightarrow \quad r \text{ ends up before } r'$$

   holds.

2. Now let $r \leftrightsquigarrow_{lex} r'$ hold. Then r ends up before r' and d. Let r' end up before d so that it may be assumed that $r' \leftrightsquigarrow d$ holds.
   Let k be the first coordinate with $r^{(k)} \leftrightsquigarrow (r')^{(k)}$. Because $r \neq r'$ it follows that $r^{(k)}$ is *not* extremal with respect to d, in contradiction to $r \in \mathcal{R}^{wk,loc}(d)$. It follows that r, d and r' end up in this order.

3. If $r' \leftrightsquigarrow_{lex} r$ holds then of course r' ends up before r and therefore d.

4. We finally examine d'. If $\mathcal{R}^{wk,loc}(d') = \varnothing$ is empty, then d' must end up before r. If not then we may assume $r' \in \mathcal{R}^{wk,loc}(d')$ and d' ends up after r', which ends up after d.

We have proved that our algorithm dissects $\mathcal{M}$ correctly. So it can be used as `SPEC_SORT`.

6.4 Implementation of the special sort with scaffold

We present some implementations of the special sort required to effectively find weak local matches in APL. It seems however that a "correct" and smooth solution should involve some enhancement (like an external function) on the side of the vendor.

6.4.1 Special sort in APL

The special sort can of course be implemented in APL itself. We give an example. It uses Primitive Sort. Its runtime behavior is bad regardless of the implementation.

Under APL runtime is abysmal. Still the algorithm guarantees that it only checks and exchanges consecutive pairs of members (and that it is stable). It can be used as a very transparent reference point.

Algorithm "special sort in APL"

```
DATA_V ← ¯1 ↓ (SORT_V ← ⍳ 1 ↑ ⍴DATA)
SIGN   ← ('Up' 'Down' ⍳ ⊂SORT) ⊃ 1 ¯1
SORT_I ← 1

:WHILE SORT_I ⍝ another loop?
    SORT_I ← 0
    :FOR NR :IN DATA_V ⍝ work through all pairs
        NR ← 0 1 + NR ⍝ pair numbers
        :IF    ~ (¯1 ↑ NR) ⊃ FLAG_I
```

```
            ⍝ special condition:
              closing flagged members are not exchanged
        :ANDIF (SIGN  ×  (> /[1] DATA[NR;])  ⍳  1)
                  <   (SIGN  ×  (< /[1] DATA[NR;])  ⍳  1)
                      ⍝ normal, stable sorting
            SORT_I ← 1 ⍝ another loop necessary
            DATA  [NR;] ← DATA  [V ← ⍋NR;]
            FLAG_I[NR ] ← FLAG_I[V      ]
            SORT_V[NR ] ← SORT_V[V      ]
        :ENDIF
    :ENDFOR
:ENDWHILE
```

6.4.2 Special sort in .Net

It is of course also possible to create a special sort in .Net, for example in C#. The author tried out Primitive Sort, Bubble Sort, Merge Sort and Natural Merge Sort and modified each to accept the flags as required.

It was rather difficult for an APL programmer to handle simple algorithms whose code run over whole pages. Still .Net solutions that contain the algorithms were created and can be provided.

Their runtime behaviour on simple 4 byte integer data was comparable to the version 13 APL+Win Grade Up primitive ⍋. In the case of real numbers the "correct" interaction with comparison tolerance was not even researched.

The algorithms were used by APL+Win in the form of a dll. Probably they could be bound in Dyalog directly as C# code. What the author (as an absolute .Net novice) could not provide was an overload which appears as *one* method in APL+Win and that can handle different data types and ranks without increasing runtime by 1–3 orders of magnitude...

We present the core of the modified Merge Sort as an example for a possible implementation under .Net.

Algorithm "special sort in .Net"

```
index = len - 1;
// Sort data by merging 2 established consecutive runs at a time.
while (index > (increment - 1))
{
    i_r = index;
    i_l = index - increment;
    begin_r = i_l + 1;
    begin_l = begin_r - increment;
    if (begin_l < 0)
    {
        begin_l = 0;
    }
    i_z = i_r;

    // Put trailing elements of rigth run into result,
       starting with first flagged element.
```

```
if (run_flagged[i_r])
{
    end_r = begin_r;
    while (flags_unsorted[end_r] == false)
    {
        end_r = end_r + 1;
    }
    end_r = end_r - 1;
    while (i_r > end_r)
    {
        data_sorted[i_z] = data_unsorted[i_r];
        flags_sorted[i_z] = flags_unsorted[i_r];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_r]; }
        i_r = i_r - 1;
        i_z = i_z - 1;
    }
}
run_flagged[index] =
    (run_flagged[index - increment] | run_flagged[index]);

if (sort_up) { i_comp_l = i_l; i_comp_r = i_r; }
else { i_comp_l = i_r; i_comp_r = i_l; }

// Put last element of left run into result if it is
   (strictly) greater than the last element of the rigth run,
   else use the latter.
while ((i_l >= begin_l) && (i_r >= begin_r))
{
    if (data_unsorted[i_comp_l] > data_unsorted[i_comp_r])
    {
        data_sorted[i_z] = data_unsorted[i_l];
        flags_sorted[i_z] = flags_unsorted[i_l];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_l]; }
        i_l = i_l - 1;
        if (sort_up) { i_comp_l = i_comp_l - 1; }
        else { i_comp_r = i_comp_r - 1; }
    }
    else
    {
        data_sorted[i_z] = data_unsorted[i_r];
        flags_sorted[i_z] = flags_unsorted[i_r];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_r]; }
        i_r = i_r - 1;
        if (sort_up) { i_comp_r = i_comp_r - 1; }
        else { i_comp_l = i_comp_l - 1; }
    }
    i_z = i_z - 1;
```

```
    }
    // Put residual elements of left run into result.
    while (i_l >= begin_l)
    {
        data_sorted[i_z] = data_unsorted[i_l];
        flags_sorted[i_z] = flags_unsorted[i_l];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_l]; }
        i_l = i_l - 1;
        i_z = i_z - 1;
    }
    // Put residual elements of rigth run into result.
    while (i_r >= begin_r)
    {
        data_sorted[i_z] = data_unsorted[i_r];
        flags_sorted[i_z] = flags_unsorted[i_r];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_r]; }
        i_r = i_r - 1;
        i_z = i_z - 1;
    }
    index = index - (2 * increment);
}
```

6.5 A sorting algorithm for weak global matches

6.5.1 Outline of a possible algorithm

We now search for weak local matches. As in subsection 5.1 the variables `REF` and `DAT` represent $\mathcal{R}$ and $\mathcal{D}$ respectively and `REL` stands for the relation $\leftrightsquigarrow$. The remarks about $\mathbb{Z}$, $\mathbb{R}$ and alphanumeric data hold.

We want to use the alternate approach to strong global matches presented in subsection 5.2 to find the desired matches. We can find weak matches using the algorithm "weak local match" and then decide if those are also (weak) global (matches). To these means we start with an arbitrary but fix $d \in \mathcal{D}$ and denote with r the first weak local match (if it exists.)

We indicate the potential for a weak global match as $\mathbb{1}_0^{wk,gl}(d)$. For each coordinate we successively examine if $r^{(k)}$ is also globally extremal. If not then there exists no match. In formulas

$$\mathbb{1}_k^{wk,gl}(d) = \begin{cases} 1 & \text{if } k = 0 \text{ and } \mathcal{R}^{wk,loc}(d) \neq \varnothing \\ \mathbb{1}_{k-1}^{wk,gl}(d) & \text{if } k \geq 1 \text{ and} \\ & \quad r^{(k)} = \operatorname{extr}\{(r')^{(k)}; r' \in \mathcal{R}_{\leftrightsquigarrow}(d)\} \\ 0 & \text{else} \end{cases}$$

If and only if $\mathbb{1}_n^{wk,gl}(d)$ is true there exists a weak global match ($\mathcal{R}^{wk,gl}(d) \neq \varnothing$) and (then) r is the first match. (We used a particular order to check the conditions of extremality but it is obviously arbitrary and not relevant for the end result.)

6.5.2 Outline of implementation of trivial case in APL

We try to find an implementation of the proposed algorithm suitable for APL. The problem is the restriction of the extremal values to $\mathcal{R}_{\leftrightsquigarrow}(d)$. We have seen in subsection 4.4 that it is not (in general) possible to sort $\mathcal{M}$ in such a way that *each* d is preceded *exactly* by $\mathcal{R}_{\leftrightsquigarrow}(d)$.

If $\mathcal{D} = \{d\}$ contains only one element it is easy to see a solution. Successively for each coordinate in their natural order

1. drop all members following d and then
2. sort using Grade Up or Grade Down.

This uses the fact that $\mathcal{R}_{\leftrightsquigarrow}(d)$ precedes d at the end of the algorithm "weak local match". Unfortunately $\mathcal{R}_{\leftrightsquigarrow}(d)$ depends on d in a way that does not allow for easy generalisation.

Of course the problem can be solved by looping through the data points d. These however will not yield a practically useful algorithm. More generally the solution will at best behave like $\mathcal{O}(M^2 \cdot \log(M))$ with $M = |\mathcal{M}|$.

6.5.3 Discussion of the problem

We cannot of course use a normal sort to find the relevant extremal value for an arbitrary coordinate. The sort cannot distinguish between $\mathcal{R}_{\leftrightsquigarrow}(d)$ and the rest of $\mathcal{R}$. In that sense it is too flexible.

On the other side the special sort with scaffold is too rigid. Using it on the result of "weak local match" leaves $\mathcal{M}$ completely stationary. More general it is easy to see that a reference can block a data point, not allowing it to reach another reference with higher value that precedes the first.

It is possible to define an intermediate sort variation that

- compares adjacent pairs m and m',
- exchanges a leading reference m and/or a trailing data point m' if $m \leftrightsquigarrow m'$ does not hold (ordinary sort condition) and
- never exchanges a leading data point with a trailing reference.

This new special asymmetric sort algorithm comes near to a solution but is not good enough. To pinpoint the problem consider first the case of one coordinate. Let d be a data point preceded by a reference r that happens to be the relevant extremal one.

It can happen that there is a data point d' with $r \leftrightsquigarrow d'$ and a reference r' with $r' \leftrightsquigarrow d$. Let further $r' \leftrightsquigarrow r$ hold. If the 4 members start in the order $(\ldots, r, d', r', d, \ldots)$ then they are stationary under the new algorithm, so the right extremal value cannot be matched to d.

As an example let $d = (2,3)$, $d' = (1,4)$, $r = (1,3)$ and $r' = (2,2)$ be 4 members with 2 coordinates. The result of "weak local match" leaves $\mathcal{M}$ in the order (r, d', r', d). Although $\mathcal{R}^{wk,loc}(d) = \{r'\}$ and $\mathcal{R}^{wk,gl}(d) = \varnothing$ evidently hold sorting the second coordinate encounters the problem just described.

Furthermore $\mathcal{R}^{wk,gl}(d') = \{r\}$ holds and the example shows that there is no obvious way to "sort" the 4 members the "right way" with respect to the second coordinate. The desired result is (r, d, d', r') and there does not seem to be a rule that would lead to the pair (r', d) getting exchanged.

6.5.4 A possibly suboptimal circumvention

To give any algorithm for weak global matches at all we use the symmetry of their definition. Both $\mathcal{R}_{\leftrightarrow}(d)$ and the extremal values defining $\mathcal{R}^{wk,gl}(d)$ do not depend on the order of the coordinates.

For each $d \in \mathcal{D}$ and for each permutation $\pi \in S_n$ we can define a weak local match $\mathcal{R}_{\pi}^{wk,loc}(d)$. If there is a weak global match then $\mathcal{R}^{wk,gl}(d) = \mathcal{R}_{\pi}^{wk,loc}(d)$ must hold for every π.

On the other hand the *first coordinate* of each weak local match is a *global* extremal value with respect to d. For the existence of a weak global match a reference must be found whose coordinates are all extremal with respect to d (and that fore $\mathcal{R}_{\leftrightarrow}(d)$). If *all* $\mathcal{R}_{\pi}^{wk,loc}(d)$ are *identical* they definitely fulfil the requirement.

Furthermore it is clearly enough to examine the n cyclic permutations $\pi \in S_n$ of length n because each of them makes another coordinate the first one. We may that fore circumvent the sorting problem:

1. for each $\pi \in S_n$ cyclic of length n find $\mathcal{R}_{\pi}^{wk,loc}(d)$ and note the index of the first match for all $d \in \mathcal{D}$
2. compare the n indices for each d
3. if and only if all indices are identical and less or equal than $|\mathcal{R}|$ (see subsection 4.2 on the convention for non-existent matches) for a specific d there is a weak global match $\mathcal{R}^{wk,gl}(d)$

This algorithm can obviously be implemented in APL as a loop on weak local matches. It is not however obvious if there exists a significantly better solution.