Typesetting number sequences

FIFO and some more ...

Kees van der Laan

Hunzeweg 57
9893 PB Garnwerd, The Netherlands
cgl@rug.nl

Abstract

Typesetting sequences of numerical values, represented via symbolic names which get their values on the fly, is dealt with. The sorting of the sequence is done by a linear sorting algorithm, of complexity O(n^2). Three or more consecutive numbers are typeset as a range.
The objective was to encode typesetting sequences of numbers as simple, concise, general, compatible, modular, orthogonal, and ..., as possible in TEX.[1]

Keywords: Typesetting sequences, citation lists, lists of references, linear sorting, FIFO, plain TEX, macro writing, education.

1 Introduction

To think about typesetting sequences of numerical values looks a bit peculiar. I was pulled in this direction when thinking about the 'range notation' problem, which was posed at the tex-nl@hearn discussion list.[2] The idea is that for example the numbers 1, 2, 3 should appear in print as 1–3. This is a trifle when the numbers are known a priori. When symbolic names are used, which get their numerical values on the fly, one has to resort to macros. The macros must take care of ordering the sequence and proper typesetting the numbers.

For that purpose the macro `\typseq`—mnemonics: typeset sequence—was written.

There are various choices possible with respect to the TEX implementation of symbolic names for numbers. The most direct form is `\chardef\`$\langle$*symbolic name*$\rangle$ = $\langle$*number*$\rangle$.[3]

The problem was solved in the Polya, 1957, way. First the kernel problem of typesetting an ordered sequence of numbers in range notation was considered. Collateral the independent problem of ordering a sequence had to be solved.[4] And finally the merging of both aspects and the TEX encoding.

The intended scope of readers consists of those who not only favor the use of LATEX and TEX, but also like to understand what is going on, and otherwise strive after keeping the encoding simple and concise.

Notations

`\ea`, `\nx`, and `\ag`, are used as shorthand for `\expandafter`, `\noexpand`, respectively `\aftergroup`.

2 Example of use

If we have:[5]

```
\chardef\dekker=5, \chardef\forsythe=3,
\chardef\reinsch=4, \chardef\knuth=11,
```

then

```
\typseq{\dekker,\knuth,\forsythe,\reinsch}
```

yields [3–5, 11].

3 Stepping stone

Suppose we have a non-descending sequence of numbers and we like to typeset these in range notation. For example 1, 2, 5, 7, 8, 9, 10 as [1, 2, 5, 7–10].

My solution is the invocation

```
\cpr{1,2,5,7,8,9,10},
```

backboned by[6]

[1] Not blurred by safeguarding goodies or limited by a particular application.
[2] Not dealt with is the typesetting of large amount of data via tables or graphs, with or without statistical methods, or the typesetting of encoded data like van Wijngaarden's method to typeset millions of prime numbers on an A4 or two.
[3] This is restricted by 256. If needed one can use `\def\`$\langle$*symbolic name*$\rangle${...}, or `\mathchardef`. ...
[4] For sorting general sequences in TEX, via O($n \log n$) algorithms, see van der Laan, 1993.
[5] That these names stand for incomparable outstanding (numerical) mathematicians is a mere coincidence.
[6] Declarations are omitted. For FIFO see van der Laan, 1992b, or the listing of the file typseq.tex included below.

```
\def\cpr#1{{\def\process{\processc}\bs
 \fifo#1,\ofif,\prtfl\es}}
\def\processc#1{%
           \init{#1}\def\processc##1%
 {\ifnum##1=\lst\else\ifnum##1=\slst
  \lst=\slst\advance\slst1{}\else
  \prtfl\sepn\init{##1}\fi\fi}}
\def\init#1{\frst=#1\lst=#1\slst=#1{}%
   \advance\slst1 }
\def\prtfl{\the\frst\ifnum\frst<\lst
 \advance\frst1{}\ifnum\frst=\lst\sepn
 \else\nobreak--\nobreak\fi\the\lst\fi}
\def\bs{[}\def\es{]}\def\sepn{, }
```

Explanation

`\cpr` This is an independent macro for just typesetting a sequence in range notation.

`\processc` The encoding makes use of the FIFO paradigm[7] to process each element. In order to perform the appropriate action we must look ahead, or postpone the typesetting, both to an unknown depth. It seems natural to postpone *as long as needed* the typesetting, with the first and last elements of the range so far—not necessarily different—stored in the counter variables `\frst`, respectively `\lst`.
The mechanism of redefinition is used to account for initialization. The first definition of `\processc` invokes `\init`—to give the counter variables `\frst` and `\lst` the value of the current list element—followed by a *redefinition* of `\processc`.[8] In the latter redefinition an element is skipped when it equals[9] the previous one. If it equals the successor of `\lst` then `\lst` gets the value of its successor. Otherwise 'the range' is typeset followed by the value of the separator, `\sepn`. The typesetting is handled by the macro `\prtfl`. The degenerate case, `\frst=\lst`, yields a single number. After having typeset the range we are in a pseudo initial state. `\frst` (and `\lst`) must get the value of the list element at hand. This is done again via the invocation of `\init`.

At the end of the list we must typeset appropriately the values of `\frst` and `\lst`. This is done via the invocation of `\prtfl`.

If we look at sorting as a CISO—Collective-In-Smallest-Out—process, then it is rather straightforward to combine sorting with `\processc`, because `\processc` handles the range typesetting independent of how the successive arguments are obtained.[10]

4 The macro \typseq

Purpose. The purpose of the macro `\typseq` is to typeset automatically a sequence in ascending order in range notation.

Input. The argument of `\typseq` is the sequence of *symbolic* names—representing the numerical values—separated by commas.

Result. The values of the sequence items are typeset in ascending order in range notation, separated by the value of `\sepn`, and delimited by the values of `\bs`, respectively `\es`.

Design. My encoding of automatic typesetting sequences of numbers comes down to

- Store the input sequence as a list with active list separators,
- Sort the list via appropriate definition of the active list separator,
- Typeset the list appropriately.

The file typseq.tex

```
%Shorthands
\let\ag=\aftergroup
\let\ea=\expandafter\let\nx=\noexpand
%Counters
\newcount\frst%First value of range
\newcount\lst %Last value of range
\newcount\slst%Successor \lst
%Newif-s
\newif\ifnoe%   Mnemonics: if not empty
%Parameters: separators
\def\sepn{, }%  Number separator
%Parameters: brackets
\def\bs{[}\def\es{]}
%FIFO with comma as separator
\def\fifo#1,{\ifx\ofif#1\ofif\fi%
 \process{#1}\fifo}\def\ofif#1\fifo{\fi}
%Store, sort and typeset
\def\typseq#1{{\strseqaslst{#1}\bs\srt\prtfl%
 \es}}% Local scope because of redef-s.
%Store sequence as list
\def\strseqaslst#1{\let\process=\processs
\xdef\list{\fifo#1,\ofif,}}
%ProcessS stores consecutive elements
%(preceded and) separated by \ls as a list.
\def\processs#1{\nx\ls\nx#1}
%Mod from Syntactic Sugar, MAPS92.1, p135
\def\srt{% Assumed is \list contains
%        symbolic names separated by \ls.
 \loop\ifx\empty\list\noefalse%
      \else\noetrue\fi%
%Test for NOEmpty list.
 \ifnoe \first\list%  \min=first element
```

[7] For FIFO and especially (variant) TEX encodings of the principle, see van der Laan, 1992b.

[8] This mechanism is general and elegant for coping with begin situations, where the first action is the only one different from the rest.

[9] For an ordered sequence 'less than' could be used. 'Equals' allows for non-ordered sequences too: just numbers which form a range are compressed.

[10] This prompts another approach for solving the problem: maintaining a priority queue. My case rests.

```
     \list%  Find minimum and store \min
     \processc\min%  Typset\min
%Delete minima from \list.
  {\def\ls####1{\ifx####1\min\else%
   \nx\ls\nx####1\fi}\xdef\list{\list}}%
 \repeat}%   end \srt
%List Separator.
\def\ls#1{\ifnum#1<\min\let\min=#1{}\fi}
%Pop up first element of list #1
\def\first#1{\def\lop\ls##1##2\pol{%
   \let\min=##1{}}\ea\lop#1\pol}

%Compressing sequences; it is assumed
%that elements are separated by commas.
\def\cpr#1{{\def\process{\processc}\bs%
 \fifo#1,\ofif,\prtfl\es}}
%Typeset element or keep track of range
\def\processc#1{\init{#1}\def\processc##1%
 {\ifnum\lst=##1{}\else\ifnum\slst=##1%
  \lst=\slst\advance\slst1{}\else%
  \prtfl\sepn\init{##1}\fi\fi}}
\def\init#1{\frst=#1\lst=#1\slst=#1{}%
   \advance\slst1{}}
%Print range: \frst-\lst (or \lst).
\def\prtfl{\the\frst\ifnum\frst<\lst{}%
 \advance\frst1{}\ifnum\frst=\lst\sepn%
 \else\nobreak--\nobreak\fi\the\lst\fi}
\endinput              %dec 92; cgl@rug.nl
%Test/example program. \tracingmacros=1
%Typeset sequence: \chardef\a=1
\chardef\b=27\chardef\c=134  %all <256!
\typseq{\c,\a,\b}.
\bye
```

Explanation

`\typseq` This composition macro invokes the macros for storing (`\strseqaslst`),[11] sorting (`\srt`), and typesetting (`\processc`).
`\strseqaslst` The arguments of the macro are the sequence elements separated by commas. First the data are stored in a list, via the use of the FIFO paradigm.[12] `\ls` is used as active list separator.
`\srt` We loop through the `\lists` until the `\list` is exhausted. In each step the minimum value is determined, typeset, and deleted from the list.[13]

- Finding the minimum.
  To initiate the process the first element is considered to be the minimum. Then the `\list` is invoked with the active list separator the function to find better minima.
- Delete minimum.
  The deletion of the minima from the list is TeX specific. We first redefine locally the list separator, `\ls`, with the function to delete the element if it equals the minimum value.[14] Then `\list` is `\xdef`-ed with `\list` as replacement text! This expands `\list` and provides in `\list` the non-minima, again separated by `\ls`-s.[15]
  Concise and elegant isn't it?
- Typesetting.
  We must account for the range notation—three or more consecutive numbers are represented as $\langle \mathit{first\ number}\rangle$–$\langle \mathit{last\ number}\rangle$—and other conventions like the separation symbol, and enclosing the total within square brackets. The latter have been parameterized into `\sepn`, `\bs`, respectively `\es`.

5 Variation

My earlier variant of `\storeseqaslst` made use of `\ag`, as follows

```
\def\strseqaslst#1{{\ag\def\ag\list\ag{%
 \let\process\processs\fifo#1,\ofif,}}}
%with
\def\processs#1{\ag\ls\ag#1}
```

When an application provides just numbers—for example page numbers after index items—the above `\strseqaslst` can be adapted to store these numbers in `\ls\1 \ls\2 \ls` ... `\ls\`$\langle n\rangle$, via

```
\def\strseqaslst#1{{\ag\def\ag\list\ag{%
 \let\process\processs\global\n0
 \fifo#1,\ofif,}}}
%with
\def\processs#1{\global\advance\n1
 \ea\xdef\csname\the\n\endcsname{#1}%
 \ag\ls\ea\ag\csname\the\n\endcsname}
```

The encoding of `\strseqaslst` can be made robust with respect to redundant spaces. I refrained from this at the moment.
`\strseqaslst` is superfluous when the convention is adopted to separate and precede the arguments of `\typseq` by `\ls`.

For other representations of the numbers, for example in superscript, one can redefine `\prtfl`. Penalties can be inserted in the replacement text of `\sepn`, to inhibit line breaks. The question is: How much?

6 Looking back

The data structure `\list` is peculiar. Not only obeys it to the queue access method, that is from left to right via the execution of `\list` with active list separator `\ls`, but individual elements can be accessed via their

[11] Mnemonics: store-sequence-as-list.

[12] This is in the spirit of the generation of n-stars, TeXbook Appendix D.1 p.374. No `\aftergroup` is necessary in this particular case, however.

[13] This has been published earlier and perhaps a bit hidden, in Syntactic Sugar, van der Laan, 1992a.

[14] Note that multiple occurrences of the minimum is accounted for.

[15] The four #-s in the definition of `\ls`—redefinition within `\srt`, line 36–37—look peculiar. We must account for the hidden definition of the loop `\body`, which makes that the definition of `\ls` is nested two levels deep!

names too. During the invocation of the example the data structure is modified as follows[16]

sequence :`\d,\k,\f,\r`
list :`\ls\d\ls\k\ls\f\ls\r`,

and gradually into `\list` equals `\empty` while sorting. Neat![17]

7 Epilogue

Typesetting of sequences on the fly occur when dealing with citation lists, or a list of references to figures and the like. LaTeX's `\cite` doesn't automatically order sequences or typeset ranges.

The range representation was posed as a problem at the tex-nl@hearn discussion list. A (LaTeX) style was offered in reply.[18] This style was composed by Ronald Kappert out of the work of Arseneau and Green. Kappert reported that Arseneau has improved upon the style. The improved style is available on file server niord.shsu.edu. According to Kappert it is in general use, especially by those who have long lists of citations. I myself don't use long citation lists. I am happy with the system of 'name followed by year.' As author, and also as reader, I know by heart the name and year of most of the works to refer to. This is structurally simple. It also makes the usual multi-pass processing of a list of references superfluous, because there is no information needed which will be created later. It is already there. However, in practice authors are restricted by journal conventions.

The hardest thing was not to introduce bells-and-whistles and to keep it as straight as possible. I refrained from introducing robustness with respect to arguments with unnecessary spaces.
Ronald Kappert is kindly acknowledged for his remarks and suggestions in proofing the article.

8 TeXniques used

- Creating a list of dynamical length via FIFO (and as alternative with `\aftergroup`).
- Sorting a list via repeated execution of `\xdef\list{ \list}`, and appropriate use of the active list separator.
- Handling recursion (loop) initialization, such that some action on first traversal is different from the action via the same name, on later traversals.

References

[1] Jeffreys, A (1990): Lists in TeX's mouth. *TUGboat* 11, no. (2), 237–244.
[2] Kappert, R (1992): scite.sty. (From the file server; it is compiled of work from D. Arseneau and I. Green: overcite.sty, drftcite.sty, cite.sty. Actually, I. Green made the style citesort.sty, which has been assimilated by D. Arseneau in his styles.)
[3] Knuth, D.E (1984): *The TeXbook*, Addison-Wesley.
[4] Laan, C.G. van der (1992a): Syntactic Sugar. MAPS92.2, 130–136. (Submitted TUG '93.)
[5] Laan, C.G. van der (1992b): FIFO & LIFO sing the BLUes. MAPS92.2, 139–144. (Submitted TUGboat.)
[6] Laan, C.G. van der (1992c): Tower of Hanoi, revisited. *TUGboat* 13, no. (1), 91–94. Also in MAPS92.1, 125–127.
[7] Laan, C.G. van der (1993): Sorting in BLUe, MAPS93.1. (Submitted TUG '93.)
[8] Lamport, L (1986): LaTeX, user's guide & reference manual. Addison-Wesley.
[9] Polya, G (1957): How to solve it. Anchor.

[16] Symbolic names abbreviated to their first letter.
[17] Instead of emptying the `\list` I could have maintained a property list, for example a permutation array. Another approach. My case rests.
[18] My code consists of the modules: storing, sorting, and typesetting in range notation (each ≈ 10 lines), next to the composition (1 line).

Sorting in BLUe

Kees van der Laan

Hunzeweg 57,
9893PB Garnwerd, The Netherlands
`cgl@rug.nl`

Abstract

Macros for number and lexicographic sorting are supplied. Data can originate from the copy, from file, or generated automatically. Lexicographic sorting allows words with ligatures and diacritical marks. Applications treated are: sorting with respect to report generation with TEX as a database tool, sorting and compressing `index.tex`, Knuth's index reminders file, and sorting control sequences separately.
It is illustrated by various examples that a set can be sorted within TEX once the ordering of the set is defined and encoded in a comparison macro, in compliance with the parameter macro `\cmp`.

Keywords: Sorting, index preparation, database handling, multiple sorting keys, macro writing, education.

Introduction

Sorting is a fundamental process. With respect to TEX, sorting was needed by Amy Hendrickson for sorting address labels [15], by Alan Jeffreys [16] and by Lincoln Durst [9] for sorting index items, to name but a few. Donald Arseneau, Ian Green, Ronald Kappert [19], and myself [25], have used sorting within TEX for citation lists. For aspects with respect to index generation see [8] and [31]. Available is Makeindex [6], [27], to cooperate with LATEX, and Salomon's plain TEX version of it [33].

All the sorting with respect to index items are external, outside of TEX.[1] This is practical, but sorting within plain is possible.[2] An advantage of TEX is that it allows for abstraction with respect to the kind of data.

Normally, number sorting and lexicographic sorting are done by different routines. This is necessary because the exchange and comparison are generally tied up with the data type. Within TEX the exchange is independent of the type, and the relational operator can be used as parameter by the sorting macro. Furthermore, secondary (and more) keys can be accounted for. The latter facility is not always available in the external sorters.

The efficiency of a sorting process depends upon the character of the data. A nearly sorted list, or a small number of items, can be handled effectively by a linear sorting routine. A non-increasing sorted list can better be walked through in reverse order than sorted. In general sorters of complexity O$(n \log n)$ are efficient for random data. Quick sort comes in handy when only part of an array has to be sorted.

For a discussion of the wide area of sorting and searching, see [20], and for programming templates, see [37].[3] For the Dutch speaking community there is the nice introduction [2].

The challenge is to encode O$(n \log n)$ sorting algorithms in TEX in a simple but flexible way.[4] Issues to address are

- a data structure must be chosen
- macros to fill the data structure
- abstracting from the sorting algorithm—heap sort, quick sort, ...
- parameterizing the comparison and exchange operations,
- abstracting in lexicographic sorting from the ASCII[5] ordering, and the
- handling of ligatures and diacritical marks.

In the first section the printing of sequences is treated. The storing of the data is considered in the second section. The sorting is elaborated on in the sections 3 and 4: sorting of numbers, respectively lexicographic sorting

[1] However, citations lists are sorted within TEX.
[2] If not for the encoding challenge.
[3] Any sorting macro should implement the algorithm with the comparison and exchange operator as parameters.
[4] Compatibility of number and lexicographic sorting has been strived after, where the particular sorting variant can be realized by appropriate `\let`-equals of the parameters.
[5] ASCII is the abbreviation of American Standard Code for Information Interchange. An ASCII table—associating each character with a number—is provided in the TEXbook, p. 367.

in the presence of the Dutch ij-ligature and diacritical marks.[6] In the fifth section the applications: sorting address labels, sorting and compressing Knuth's index reminders file, and sorting of control sequences separately, are dealt with. In the appendices I supplied the listings of the files: `heap.tex`, `quick.tex`, `sort.tex` and my testdriver `sort.tst`.

There are so many details in sorting and the TEX encoding of it, that I hope that the remainder is not too concise for those who are really interested in the details of the TEX encoding. On the other hand, I hope it won't contain too much for those who just like to get an idea of the possibilities of TEX with respect to sorting.[7]

Approach. The three processes: initialization, sorting and typesetting, are separately and independently designed.

For filling the data structure I considered it handy to have a few macros which store from

- copy (`\seq`...`\qes`),
- a file (`\storefrom`), or
- a process, which (randomly) generate elements (`\storerandomn`, `\storerandomw`).

For sorting I provided

- the Ben Lee User level macros (`\sortn`, `\sortaw`, `\sortw`), and
- the blue collar macros (`\heapsort`, `\quicksort`).

For typesetting the data structure I used the macros `\prtn`, respectively `\prtw`.[8]

Files. The file `sort.tex` contains the macros for storing (`\seq`...`\qes`, `\storefrom`, and `\storerandomn`, `\storerandomw`), for sorting (`\sortn`, `\sortaw`, `\sortw`), and for typesetting (`\prtn`, `\prtw`, and `\prtind`). Apart from these, the file contains the common definitions of the `\heapsort` and `\quicksort` macros, as well as variants for the parameter macros.

The files `heap.tex` and `quick.tex` contain the `\heapsort`, respectively `\quicksort`, macro along with specific auxiliaries.

My testdriver is the file `sort.tst`.

Definitions and notations. A sequence is defined as a row of numbers, respectively words, separated by spaces. The structure `\csname`$\langle k\rangle$`\endcsname`, is associated with an array with index $k = 1, 2, \ldots, n$. To denote in the documentation a value pointed by the number $\langle k\rangle$, I made use of `\val{`$\langle k\rangle$`}`, with `\def\val#1{\csname#1\endcsname}`. Macro names take suffix -n, -w, when specific for number, respectively word data. For example `\sortn` stands for sort numbers, `\prtw` stands for print words. I have typeset the in-line results of the examples in bold face.

For transferring values to a macro, I generally refrained from the (optional) parameter mechanism, as it is used in nowadays high-level programming languages. Instead I used Knuth's parameter TEXnique, which comes down to providing definitions and using these by invokations, eventually after a `\let`-equal.

I have used the shorthand notation `\ea`, `\nx`, and `\ag` for `\expandafter`, `\noexpand`, respectively `\aftergroup`. `\k` is used as counter to loop through the values $1, 2, \ldots, n$, the index domain. `\n` contains the maximum number of sequence elements, n. `\ifcontinue` is used for controlling loops. The array and the counter `\status` had to be maintained globally, because of the nesting of loops.

1 Typesetting elements

After sorting the typesetting must be done. In general this is dependent upon the application and will demand Hi-TEXnique.[9] For simplicity and in order to concentrate on the sorting aspects I typeset the sequence element after element, via `\prtn`, or `\prtw`.

Example (Typesetting a number sequence)[10]

```
\def\1{314}\def\2{1}\def\3{27}\n3 \prtn
```

yields: **314, 1, 27**.

Example (Typesetting a word sequence)

```
\def\1{ik}\def\2{j\ij}\def\3{h\ij}\n3 \prtw
```

yields: **ik jij hij**.

1.1 TEX encoding

Design choice. The elements are typeset in the default font. The separator is parameterized into `\sep`. Number sequences are typeset in range notation.

Input. The array `\`$\langle k\rangle$, $k = 1, 2, \ldots, n$, and the counter `\n` with value $\langle n\rangle$,[11] and optionally a value $\langle kzero\rangle, \geq 0$, in `\kzero`.

[6] Adaptable to other ligatures and accents.

[7] Ben Lee User, BLU for short, can always page through the provided headings and grasp 'what it is all about' from the included examples.

[8] The file `sort.tex` contains also `\prtind`, to typeset `index.tex`.

[9] Think for example of Knuth's typesetting of the index of the TEXbook, p. 261–263. It is in the chapter on OTR-s (Output Routines) with aura '... the following material will take you all the way to the rank of Grandmaster, i.e., a person who can design output routines.'

[10] In the examples `\def`-s are used to define a one digit as control symbol. `\csname...\endcsname` must be used for two or more digits.

[11] The defaults for the parameter `\sep`—`\sepn`, respectively `\sepw`—are provided in the file `sort.tex`.

Result. The array \$\langle kzero+1\rangle$: \$\langle n\rangle$ is typeset. `\kzero` is default 0.

The macros

```
\def\prts{{\k\kzero%print \1,...\n
 \def\sep{\let\sep\sepw}%
 \loop\ifnum\k<\n\advance\k1
    \sep\csname\the\k\endcsname
 \repeat}}\let\prtw\prts
%
\def\prtn{{\k\kzero%Print ranges
 \loop\ifnum\k<\n\advance\k1
  \ea\prc\csname\the\k\endcsname
 \repeat\prtfl}}
%
\def\prc#1{\init{#1}\def\prc##1{%
 \ifnum##1=\lst\else\ifnum##1=\slst
    \lst\slst\advance\slst1 \else
       \prtfl\sepn\init{##}\fi\fi}}
%
\def\prtfl{\the\frst\ifnum\frst<\lst
 \advance\frst1 \ifnum\frst=\lst\sepn
   \else\nobreak--\nobreak\fi\the\lst}
%
\def\init#1{\frst#1\lst#1\slst#1\advance
 \slst1{}}
```

Explanation. Abstraction of the lower index into `\kzero`, default 0, makes it possible to typeset parts of the array. The elements are separated by the separator given in, `\sepn`, respectively `\sepw`. The encoding is TEX specific. Each first time the loop is traversed the invokation of `\sep` redefines itself with the actual separator. On subsequent traversals the provided separator is typeset.

The replacement text of `\prtn` and `\prtw` is a group, and therefore the loop's `\body` cannot redefine the `\body` of an outer loop.

In order to account for number ranges `\prtn` uses `\prc`, a simplified version of `\processc`, borrowed from [25].

2 Storing a sequence

As data structure the following TEX-specific encoding[12] is used.

$$\texttt{\textbackslash csname}\langle k\rangle\texttt{\textbackslash endcsname},\quad k=1,2,...,n.$$

Writing to, respectively reading from, the k^{th} element goes via[13]

$$\texttt{\textbackslash ea\textbackslash def\textbackslash csname}\langle k\rangle\texttt{\textbackslash endcsname}\{\langle k^{th} elem.\rangle\},$$

and

$$\texttt{\textbackslash csname}\langle k\rangle\texttt{\textbackslash endcsname}.$$

When a counter `\k`, which takes the values $1,2,\ldots,n$, is used, then TEX requires `\the\k` for the index number $\langle k\rangle$.

To get the hang of it. The reader must be aware of the differences between

- the index number, $\langle k\rangle$
- the counter variable `\k`, with the value $\langle k\rangle$ as index number
- the control sequences \$\langle k\rangle$, $k=1,2,\ldots,n$, with as replacement texts the items to be sorted.

When we have `\def\3{4}` `\def\4{5}` `\def\5{6}` then
`\3` yields **4**,
`\csname\3\endcsname` yields **5**, and
`\csname\csname\3\endcsname\endcsname` yields **6**.

Similarly, when we have
`\k3 \def\3{name} \def\name{action}` then
`\the\k` yields **3**,
`\csname\the\k\endcsname` yields **name**, and
`\csname\csname\the\k\endcsname\endcsname` yields **action**.[14] To exercise shorthand notation the last can be denoted by `\val{\val{\the\k}}`.

Another `\csname...` will execute `\action`, which can be whatever you provided as replacement text.

2.1 From copy

Elements available in the copy of an author are stored via

`\seq`$\langle sequence\rangle$`\qes`.[15]

Example (Storing numbers from copy)
`\seq1 314 27\qes` stores the elements. For verification `\prtn` yields: **1, 314, 27**.

Example (Storing words from copy)
`\seq ik j{\ij} h\ij\qes` stores the elements. For verification `\prtw` yields: **ik jij hij**.

TEX encoding

Design choice. The sequence is stored in an array via the FIFO TEXnique [23]. The process is independent of the type. Numbers or words (text) can be stored by the same macro.

Input. Data from the user copy preceded by `\seq` and followed by the separator `\qes`. The elements must be separated by a ␣, which is not gobbled by TEX's mouth. (In practice this means that words ending with a control sequence—`\i`, `\j`, or for Dutch `\ij`—must have braces around that control sequence.)

[12] Functionally equivalent to an array. Amy Hendrickson [15] used arrays in TEX although she did not call them as such. Adrew Greene [13], while playing around in TEX's mind, associated already the array concept with `\csname...`.

[13] Actually, I used `\gdef`-s, `\xdef`-s, and `\the\k`.

[14] Confusing, but powerful!

[15] Mnemonics: sequence. This abstracts from all the `\def`-s, casu quo `\csname...\endcsname`-s, as provided in the examples.

Result. The array $\backslash\langle k\rangle,\ k = \langle kzero+1\rangle, 2, \ldots, n,$ with the sequence elements as values. The counter `\n` will contain the value $\langle n\rangle$. $\langle kzero\rangle$ is a bias, with default value 0.

The macros

```
\def\seq#1\qes{%
           \k\kzero\fifow#1 \wofif{} }
%
%and auxiliaries
\def\fifow#1 {\ifx\wofif#1\n\k\wofif\fi
 \processw{#1}\fifow}
               \def\wofif#1\fifow{\fi}
%
\def\processw#1{\advance\k1
 \ea\gdef\csname\the\k\endcsname{#1}}
```

Explanation. The idea is that the elements from the copy enclosed by `\seq` and `\qes`—and appended in the macro `\seq` by ␣`\wofif{}`␣[16]—are processed as arguments of the macro `\fifow`. This macro has a ␣ as endseparator. When `\wofif` is encountered the number of elements is stored in `\n` and the recursion is terminated by the invokation of `\wofif`. The latter macro gobbles all the tokens—in this case `\fi \processw{#1}`—up to and including the next invokation of `\fifow`. Its replacement text inserts a new `\fi`, to correct the disturbed `\if...\fi` balance.

The macro `\processw` maintains the (index) counter and actually stores each element, globally.

2.2 From a file

In applications the words (and other information like page numbers[17] for index preparation) are gathered into a file for later, usually external, processing.

Example (Storing from file)
If the file `index.tex` contains the records

```
word !3 314
word !1 27
tag !1 1
word !1 1
```

then

```
\storefrom{index.tex}
```

stores the elements from the file into the array.

For verification the array is printed by

```
\begin{quote}
\let\sepw\\\prtw\unskip.
\end{quote}
```

with result[18]

> word !3 314
> word !1 27
> tag !1 1
> word !1 1

TEX encoding

Specification. Records from a user specified file are to be read into the array. On termination the counter `\n` contains the number of stored elements.

Input. The file with the elements given per line. `\kzero` is default 0.

Result. The array $\backslash\langle k\rangle,\ k = \langle kzero\rangle + 1, 2, \ldots, n,$ with the elements as values. `\n` contains the upper bound of the array, $\langle n\rangle$.

The macro

```
\def\storefrom#1{%#1 is file name
 \openin\rec#1 \k\kzero \continuetrue
 \loop\ifeof\rec\continuefalse\fi
 \ifcontinue\advance\k1 \read\rec to\xyz
  \ea\global\ea\let\csname\the\k\endcsname\xyz
 \repeat\advance\k-1\n\k\closein\rec}
```

Explanation. The `\newread\rec` has been specified in the file `sort.tex`. TEX appends a `\par` to the opened file, therefore I had to decrement the counter `\k` by 1 at the end. After `\rec#1` a ␣ is mandatory; an empty group is not recognized as terminator. Because of the lack of an `\ifnoteof` and of the way `\loop` has been encoded—TEXbook, p. 219, an `\else` cannot be used in the body of the loop as part of the termination—I used the `\newif\ifcontinue` for controlling the loop. The bias $\langle kzero\rangle$ is handy for merging index files.

2.3 From a generator

Although the automatic generation of data is only used in the tests, it seemed worthwhile for me to include these macros too, as an example of how data can be created and stored.

Numbers

A random number generator—the macro `\rnd`—has been encoded in TEX by Reid [30]. I added `\storerandomn` to store the specified number of random numbers in the array.

Example (Storing random generated numbers)

```
\rndnum5 \storerandomn5\prtn
```

yields:[19] **1, 88, 62, 27, 1.**

[16] The empty group is needed because spaces after control sequences are gobbled. Beware!

[17] Known by the OTR—Output Routine—only. For writing the index reminders to the file `index.tex` see the TEXbook, p. 424, the macro `\writeit` and auxiliaries. A simplified encoding will be provided in Manmac BLUes, see elsewhere in this MAPS.

[18] Note that I had to add an `\unskip`. `\\` is LATEX's newline.

[19] More clearly, I could have provided `\rndnum=5`␣ and `\storerandomn{5}`, to emphasize the different syntactical roles of the number 5.

The macros. The encoding of my macro is straightforward, once I decided to use Reid's random generator macro, `\rnd` [30].

```
\def\storerandomn#1{%#1 number
                     %of r-numbers
 \n#1\k0{\loop\ifnum\k<\n\advance\k1 %
                              \rnd\ea
  \xdef\csname\the\k\endcsname{%
                    \the\rndval}%
 \repeat}}
%
\def\rnd{\global\multiply\rndnum371
 \global\advance\rndnum1
 \ifnum\rndnum>99999
  \rndtmp\rndnum \divide\rndtmp100000
  \multiply\rndtmp100000
  \global\advance\rndnum-\rndtmp
 \fi\global\rndval\rndnum
 \global\divide\rndval1000 }
```

Words

Reid [30] introduced his macros for generating random paragraphs. I added `\storerandomw` to store the specified number of random words in the array.

Example (Storing random generated words)

`\rndnum5 \storerandomw5\prtw`

yields: **ajqjjhfn fyi uednas ahw zr.**

The macros

```
\def\storerandomw#1{%#1 number of words
 \n#1\nw\n{\loop\ifnum0<\nw
  {\ag\defarr\ag{\randomword}}%
                         \advance\nw-1
 \repeat}}%end s-r-w.
%
\def\defarr{\ea\gdef%
                \csname\the\nw\endcsname}
%
\def\randomword{\rnd\nc\rndval
 \divide\nc15\advance\nc2
 \loop\ifnum0<\nc\randomchar%
                          \advance\nc-1
 \repeat}%end r-word
%
%Random character is modified
\def\randomchar{\rnd
 \multiply\rndval29\divide\rndval100
 \ifnum\rndval=26\rndval0 \fi
 \ifnum\rndval>26\rndval4 \fi
%Mod cgl: I \ag-ed the letter
 \ea\ag\ifcase\rndval
  a\or b\or c\or d\or e\or f\or g\or h\or
  i\or j\or k\or l\or m\or n\or o\or p\or
  q\or r\or s\or t\or u\or v\or w\or x\or
  y\or z\fi}
```

Explanation. Although the same approach as for storing random generated numbers has been followed, I had to modify the code due to the need for intermediate storing of each random generated letter. A random word consists essentially of random numbers, mapped onto letters. The numbers are generated in an inner loop, via Reid's macro `\randomchar`. Because of the nesting of loops I had to group the inner loop. Realizing this, prompted a TEX specific way for storing the letters generated in the inner loop. The letters are placed after the enclosing group via `\aftergroup`. When the group is ended the word is stored as replacement text of `\`$\langle nw\rangle$. (The tokens for the definition are `\ag`-ed before the inner loop; the closing brace is already after the innerloop.)

3 Sorting of numbers

Example

`\seq314 1 27\qes\sortn` yields: **1, 27, 314**.

3.1 Design choices

The backbone of my 'sorting in an array' is the data structure

$$\texttt{\textbackslash csname}\langle k\rangle\texttt{\textbackslash endcsname}\{\langle k^{th} elm.\rangle\},\quad k=1,2,\ldots,n,$$

with k the role of array index and n the number of items to be sorted.

The encoding is parameterized by `\cmp`, the comparison macro, which differs for numbers, strings, and in general when more sorting keys have to be dealt with.[20] The result of the comparison is stored globally in the counter `\status`.

3.2 TEX encoding

Input. The elements are assumed to be stored in the array `\`$\langle k\rangle$, $k = 1, 2, \ldots n$. The counter `\n` must contain the value $\langle n\rangle$.

Result. The sorted array `\1`, `\2`, ... `\`$\langle n\rangle$, with `\val1` $\leq$ `\val2` $\leq \ldots \leq$ `\val`$\langle n\rangle$.

The macros

```
\def\sortn{\let\cmp\cmpn\sort\prtn}
%
\def\cmpn#1#2{%#1, #2 must expand into numbers
%Result: \status= 0, 1, 2 if
%         \val{#1} =, >, < \val{#2}.
 \ifnum#1=#2\global\status0 \else
  \ifnum#1>#2\global\status1 \else
            \global\status2 \fi\fi}
%
\def\sort{\heapsort}.
```

Explanation. The above shows the structure of each of the Ben Lee User sorting macros.

Sorting: `\sortn.` A (pointer) `\def\sortn{...}` is introduced which has as replacement text the setting of the parameter `\cmp`, and the invokations of the actual sorting macro and the macro for typsetting the sorted sequence.

[20] For an example see the sorting of Knuth's index reminders in section 5.

Comparison operation: `\cmpn`. The result of the comparison is stored globally in the counter `\status`. The values $0, 1, 2$ denote $=, >, <$, respectively.

Exchange operation: `\xch`. The values can be exchanged via[21]

```
\def\xch#1#2{%#1, #2 counter variables
 \edef\aux{\csname\the#1\endcsname}\ea
 \xdef\csname\the#1\endcsname{\csname
   \the#2\endcsname}\ea
 \xdef\csname\the#2\endcsname{\aux}}.
```

3.3 Some testing

Apart from the examples as given above, `\sortn` has been tested on sequences of random numbers. Some idea of the efficiency was obtained and no reasonable restrictions with respect to the number of items to be sorted, other than the installation limitations, were encountered. For this purpose use has been made of Reid's random number generator in TEX, `\rnd` [30].

Timings. On my 8086 MS-DOS PC `\sortn` (without time needed to create the array, but with the time needed to write the sorted array to the dvi-file) had the (near-linear) performance

No	≈ Time
15	13 seconds
50	1 minute
200	5 minutes.

The University's VAX8650[22] needed ≈ 1.75 minutes for sorting 500 numbers.[23] The measurements were done with `\heapsort` as sorting macro.

3.4 Variation

For short sequences algorithms of complexity $O(n^2)$ are generally used.[24]

```
%O(N*N) sorting.
\def\sort{\bubblesort}
%
\def\bubblesort{%Data in \1, \2,...\<n>.
{\loop\ifnum1<\n{\k\n
  \loop\ifnum1<\k\advance\k-1 \cmp\k\n
   \ifnum1=\status\xch\k\n\fi
  \repeat}\advance\n-1
 \repeat}}%end \bubblesort
```

4 Lexicographic sorting

Given the blue collar workers `\heapsort`, respectively `\quicksort`, we have to encode the comparison macro in compliance with the parameter macro `\cmp`. But, ... lexicographic sorting is more complex than number sorting. We lack a general comparison operator for strings,[25] and we have to account for ligatures and diacritical marks.

In creating a comparison macro for words, flexibility must be built in with respect to the ordering of the alphabet, and the handling of ligatures and diacritical marks.

Example (Sorting ASCII words)

```
\seq a b aa ab bc bb aaa\qes\sortw
```

yields: **a aa aaa ab b bb bc**.

Example (Sorting words with ij-ligature)

```
\seq{\ij}st{\ij}d {\ij} {\ij}s in tik
 t\ij\qes\sortw
```

yields: **in tik tij ij ijs ijstijd**.

Example (Sorting accented words)

```
\seq b\'e b\`e \'a\'a ge\"urm geur aa a
ge{\ij}kt be ge\"\i nd gar\c con\qes
\sortw
```

yields: **a aa áá be bé bè garçon geïnd geur geürm geijkt**.

Reculer pour mieux sauter. Because of the complexity and the many details involved I recede with simplified cases as stepping stones. I'll first guide you through the encoding of the comparison macro for

- one-(ASCII)letter-words, and
- ASCII strings, of undetermined length,

after which we will come back to the main track of the encoding of the general comparison macro.

One-(ASCII)letter-words. The issue is to encode the comparison macro, in compliance with the parameter macro `\cmp`. Let us call this macro `\cmpolw`.[26] Its task is to compare one-letter words and store the result of each comparison globally in the counter `\status`. As arguments we have `\def`-s with one letter as replacement text.

```
\def\cmpolw#1#2{%#1, #2 are def-s
%Result: \status= 0, 1, 2 if
%        \val{#1} =, >, < \val{#2}.
 \ea\chardef\ea\cone\ea`#1{}%
 \ea\chardef\ea\ctwo\ea`#2{}%
 \global\status0 \lge\cone\ctwo}
%
\def\lge#1#2{%#1, #2 are letter values
```

[21] For a better and more general macro, see section 4 about lexicographic sorting. Here the definitions are completely expanded, which is not necessary and therefore inefficient.
[22] Just to give the reader an idea because VMS is a time sharing system.
[23] As expected with 0–99 as printed result. Neat!
[24] A nice example of encoding nested loops. Should be part of courseware about macro writing in TEX.
[25] It is not part of the language, nor provided in plain. Victor Eijkhout [10] supplied one. The (limited) predecessor of my comparison macro has appeared in [23]. Those macros don't abstract from the ASCII ordering or allow for accented words and ligatures.
[26] Mnemonics: compare one letter words.

```
%Result: \status= 0, 1, 2 if #1 =, >, < #2.
 \ifnum#1>#2\global\status1 \else
   \ifnum#1<#2\global\status2 \fi\fi}
%
\seq z y A B a b d e m n o p z z u v c g
q h j I i l k n t u r s f Y\qes
\let\cmp=\cmpolw\sort\prtw
```

The above yields: **A B I Y a b c d e f g h i j k l m n n o p q r s t u u v y z z z**.

Explanation `\cmpolw`**.** In order to circumvent the abundant use of `\expandafter`-s, I needed a two-level approach: at the first level the letters are ‘dereferenced,’ and the numerical value of each replacement text is provided as argument to the second level macro, `\lge`.[27]

ASCII words. The next level of complexity is to allow for strings, of undetermined length and composed of ASCII letters. Again the issue is to encode the comparison macro, in compliance with `\cmp`. Let us call the macro `\cmpaw`[28]. Its task is to compare ASCII words and store the result of each comparison globally in the counter `\status`.

The problem is how to compare strings letter by letter. Empty strings are equal. This provides a natural initialization for the `\status` counter. As arguments we have `\def`-s with words of undetermined length as replacement text.

```
\def\cmpaw#1#2{%#1, #2 are def-s
%Result: \status= 0, 1, 2 if
%        \val{#1} =, >, < \val{#2}.
 {\let\nxt\nxtaw\cmpc#1#2}}
%
\def\cmpc#1#2{%#1, #2 are def-s
%Result: \status= 0, 1, 2 if
%        \val{#1} =, >, < \val{#2}.
\global\status0 \continuetrue
{\loop\ifx#1\empty\continuefalse\fi
      \ifx#2\empty\continuefalse\fi
 \ifcontinue\nxt#1\nxtt \nxt#2\nxtu
     \lge\nxtt\nxtu
     \ifnum0<\status\continuefalse\fi
 \repeat}\ifnum0=\status
 \ifx#1\empty\ifx#2\empty\else
                  \global\status2 \fi
 \else\ifx#2\empty\global\status1 \fi
 \fi\fi}
%
\def\nxtaw#1#2{\def\pop##1##2\pop{\gdef
 #1{##2}\chardef#2`##1}\ea\pop#1\pop}
%
\seq a b aa ab bc bb aaa\qes
\let\cmp\cmpaw\sort\prtw.
```

The above yields: **a aa aaa ab b bb bc**.

Explanation

Comparison: `\cmpaw`**.** The macro is parameterized over the macro `\nxt`. The main part of `\cmpaw` has been encoded as `\cmpc`. (That part is also used in the general case.)

We have to compare the words letter by letter. The letter comparison is done by the already available macro `\lge`. The `\lge` invokation occurs within a loop, which terminates when either of the strings has become empty. I added to stop when the words considered so far are unequal. At the end the status counter is corrected if the words considered are equal and one of the #-s is not empty: into 1, if #1 is not empty, and into 2, if #2 is not empty.

Head and tail: `\nxt`**.** The parameter macro `\nxt` has the function to yield from the replacement text of its first argument the ASCII value of the first letter and deliver this value as replacement text of the second argument.[29] The actual macro `\nxtaw` pops up the first letter and delivers its ASCII value—a `\chardef`—as replacement text of the second argument. Note that the first parameter is globally redefined for the emptiness-test after the loop.

4.1 Design choices

Sorting of words with ligatures and accents is done via the ordering defined in a so-called ordering table.[30] This entails that the comparison of letters has to be generalized such that accents are recognized too. For the (accented)letter-to-number conversion the **`** is replaced by a table look-up.

Comparison operation. A special situation arises with diacritical marks. Within the context of sorting the commands for diacritical marks have been redefined with the function to provide for the control symbol together with the accompanying letter an appropriate value from the ordering table.

Note that when the ASCII ordering is sufficient, no ordering table is needed. For that case `\cmp` can be `\let`-equal to `\cmpaw`.

Ordering table. In Dutch the ij is peculiar. It is mostly used as a ligature[31] and in that role its lexicographic position is between `x` and `z`.[32] This character is not accounted for in the ASCII table, therefore we

[27] Mnemonics: letter greater or equal. A nice application of the use of `\ea`, `\chardef`, and the conversion of a character into a number: **`**. Note that the values of the upper case and lower case letters differ (by 32) in ASCII.

[28] Mnemonics: compare ASCII words.

[29] Splitting up into ‘head and tail,’ is treated in the TeXbook, Appendix D.2, p. 378, the macro `\lop`. There use has been made of token variables instead of `\def`-s.

[30] In the ordering table numbers are associated to letters, ligatures and accented letters.

[31] Also as two characters. For example in bi-jection (hyphen for emphasis) and the like.

[32] I was quite surprised to find out that my Dutch dictionary does *not* sort on the ij-ligature?!?

need an ordering table. In some other languages similar situations exist, so the idea of abstraction from the ASCII ordering is useful,[33] if not for the handling of diacritical marks. For the input I adopted the convention to supply `\ij`. The ordering table is implemented via a list of `\chardef`-s.[34] Numbers are not assigned consecutively in the ordering table, leaving room for accented letters.

I have provided the same values for upper and lower case letters. The successor values are reserved for the accents: acute, grave, umlaut, and hat. I also accounted for the cedille.

4.2 TEX Encoding

Purpose. To sort words with (Dutch) accents and ij-ligature.

Input. The elements are assumed to be stored in the array $\backslash\langle k\rangle$, $k = 1, 2, \ldots n$. The counter `\n` must contain the value $\langle n\rangle$.

The default settings are done in the file `sort.tex`. The macro `\accdef` contains the modified accent definitions, and the macro `\accstr` the string of accent control sequences.

Result. The sorted array `\1`, `\2`, … $\backslash\langle n\rangle$, with `\val1` $\leq$ `\val2` $\leq \ldots \leq$ `\val`$\langle n\rangle$.

The macros

```
%Modifications/addenda to number sorting
%Sorting and typesetting.
\def\sortw{{\accdef\let\cmp\cmpw\sort}%
                                   \prtw}
%
%Compare words.
\def\cmpw#1#2{%#1, #2 are def-s.
%Result: \status= 0, 1, 2 if
%        \val{#1} =, >, < \val{#2}.
 \let\nxt\nxtw\cmpc#1#2}
%
%Yield value of next (accented) letter.
\def\nxtw#1#2{\def\pop##1##2\pop{%
 \gdef#1{##2}\def\head{##1}}%tail and head
 \ea\pop#1\pop
 \ea\loc\head\accstr%\head in accentcs?
 \iffound\let\acs\head
  \ea\pop#1\pop%next tail and head
  \ea\let\ea#2\csname ot\acs\head\endcsname
 \else\ea\let\ea#2\csname ot\head\endcsname
 \fi}
%
\def\accstr{\`\'\"\^\c}
%
\def\accdef{%
 \def\'##1{##1g}\def\`##1{##1a}
                              %acute grave
 \def\"##1{##1t}\def\c##1{##1c}
                              %trema cedille
 \def\^##1{##1h}%hat
 \def\i{i}\def\j{j}}%dotless i, j

\def\loc#1#2{\def\locate##1#1##2\end
 {\ifx\empty##2\empty\foundfalse
 \else\foundtrue\fi}\ea\locate#2.#1\end}
%
%Ordering table
\chardef\ota32 \chardef\otA32
 \chardef\otaa33 \chardef\otag33
 \chardef\otat34 \chardef\otah35
%et cetera, see Appendix C
```

Explanation

Sorting: `\sortw`. A (pointer) `\def\sortw{...}` is introduced, which has as replacement text the insertion of the accent definitions via the invokation of `\accdef`, the setting of the parameter `\cmp`, and the invokations of the actual sorting macro and the macro for typesetting. A group is used in order to keep the temporary redefinitions of the accents local. `\accdef` yields the modified definitions for the accents and special letters like `\i`, and `\j`. The purpose of these definitions is to get the right value from the ordering table.

Comparison operation: `\cmpw`. The parameter macro `\nxt` is `\let`-equal to `\nxtw`. The comparison is done by the common `\cmpc`,[35] which stores globally the result of the comparison in the counter `\status`.

Head and tail: `\nxtw`. This macro peels a token[36] from the first argument, selects the associated value form the ordering table and delivers the latter value in the second argument, as a `\chardef`. The complication is that when we have an accent we have to consider the next token too, and select the associated numerical value for the combination.[37]

The remainder of the word, the tail, is delivered globally in the first argument, for the afterloop emptiness-test. The local macro `\pop` yields the head and the tail of a word. `\loc` determines whether the token in `\head` is an accent control symbol.[38]

Exchange operation: `\xch`. No expansion of the accents must take place, therefore the already stored data are copied via the `\let`-equal TEXnique.

```
\def\xch#1#2{%#1, #2 counter variables
```

[33] As communicated by Wlodek Bzyl, with respect to Czech.
[34] Remember that a `\chardef` name can be used as a number.
[35] For an explanation see the subsection about ASCII words.
[36] Or debraced group.
[37] A neat use of `\ea`, `\let`, and `\csname...`, with as result a `\chardef`!
[38] This is a generalization of the search for $\langle char\rangle \in \langle string\rangle$ [23]. On second thoughts, I consider this a neat generalization. The temporary redefinitions are parameterized into `\def\accdef...` .