Paradigms: Sorting

Kees van der Laan

1 BLUe's Design IX

Hi folks. A strong and *unique* point of BLUe's format system is its indexing on the fly. Be it for a total document or just for a chapter. One of the requisites for indexing on the fly is the possibility to sort within TEX.

Sorting has always been an important topic in computer science. In TEX I needed sorting on several occasions especially for sorting numbers such as citation lists, words such as addresses, and index entries.

This note is devoted to paradigms encountered while implementing and applying sorting in TEX.

Sorting can be characterized by
- the set to be sorted (numbers, word. etc.)
- the addressing of elements of the set
- the ordering for the set
- the comparison operation, and
- the exchange operation.

To do some sorting of your own please load from `blue.tex` the index macros via `\loadindexmacros`. Below parts have been extracted from that collection of macros to make this note as intelligible as possible. `\ea` is my shortcut for `\expandafter`.

2 Linear sorting

A simple sorting method is repeatedly searching for the smallest element. In the example below the set is defined as a def with list element tag `\\`.

```
\def\lst{\\\ia\\\ib\\\ic}
\def\ia{314} \def\ib{1} \def\ic{27}
%
\def\dblbsl#1{\ifnum#1<\min\let\min=#1\fi}
%
\loop\ifx\empty\lst\expandafter\break\fi
   \def\\{\let\\=\dblbsl\let\min=} %space
   \lst%find minimum
   \min%typeset minimum
   {\def\\#1{\ifx#1\min \else\nx\\%
    \nx#1\fi}\xdef\lst{\lst}}%
\pool%Inspired upon van der Goot's
     %Midnight macros.
\def\loop#1\pool{#1\loop#1\pool}
\def\break#1\pool{}
```

The coding implements the looping of the basic steps
- find minimum (via `\lst`, and suitable definition of DeK's list element tag `\\`)
- typeset minimum (via `\min`)
- delete minimum from the list (via another appropriate definition of the list element tag, and the use of `\xdef`.

Remark. The kludge for using `\ifx` instead of `\ifnum` in the deletion part is necessary because TEX inserts a `\relax`.

3 Sorting in an array

If we adopt array addressing in TEX for the elements to be sorted then we can implement bubble sort in TEX too.[1]

3.1 Array addressing

When we think of associating values to (index) numbers—`1` → `\value{1}` —then we are talking about an array. A mapping of the natural numbers on ... for example the natural numbers. The `\value` control sequence can be implemented as follows.

```
\def\value#1{\csname#1\endcsname}
```

The writing to the array elements can be done via

```
\def\1{<value1>} \def\2{<value2>}...
```

In general this must be done via

```
\ea\def\csname<number>\endcsname{<valuenumber>}
```

3.1.1 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 `\<k>`, $k = 1, 2, \ldots, n$, 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**.[2] To exercise shortcut notation the last can be denoted by `\value{\value{\the\k}}`.

[1] The above example of linear sorting can be seen as sorting in a so-called associative array.

[2] Confusing, but powerful.

Another `\csname...` will execute `\action`, which can be whatever you have provided as replacement text.[3]

4 Bubble sort

This process looks repeatedly for the biggest element which is swapped to the end. This is done for the complete array, the array of size $n-1$ et cetera. The pseudo code reads as follows.

for $n :=$ *upb* **downto** 2 **do**

begin for $k :=$ $n-1$ **downto** 1 **do**

if $a[n] < a[k]$ **then**

exchange(a[*n*], *a*[*k*]*)*;

end;

The TEX macro reads as follows.

```
\def\bubblesort{%Data in defs \1, \2,...\<n>.
%Result: \1<=\2<=...<=\<n>.
{\loop\ifnum1<\n{\k\n
  \loop\ifnum1<\k \advance\k-1
     \cmp{\deref\k}{\deref\n}%
   \ifnum\status=1 \xch\k\n\fi
  \repeat}\advance\n-1
 \repeat}}%end \bubblesort
%with auxiliaries
\def\deref#1{\csname\the#1\endcsname}
\let\cmp\cmpn %from blue.tex or provide
%\def\cmp#1#2{%Comparison. Yields
% \status=0, 1, 2 for =, >, <
%...}
%\def\xch#1#2{%exchange
%#1, #2 counter variables
%...}
```

5 Heap sort

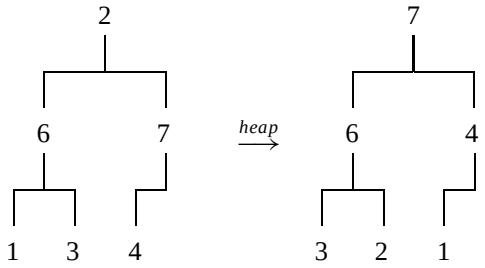
We can organize the array as a heap. A heap is an ordered tree. Loosely speaking for each node the siblings are smaller or equal than the node.

The process consists of two main steps
- creation of a heap
- sorting the heap

with a sift operation to be used in both.

In comparison with my earlier release of the code in MAPS 92.2, I adapted the notation with respect to sorting in *non-decreasing* order.[4]

What is a heap? A sequence $a_1, a_2, \ldots, a_n$, is a heap if $a_k \geq a_{2k} \wedge a_k \geq a_{2k+1}, k = 1, 2, \ldots, n \div 2$, and because a_{n+1} is undefined, the notation is simplified by defining $a_k > a_{n+1}, k = 1, 2, \ldots, n$.
A tree and one of its heap representations of 2, 6, 7, 1, 3, 4 read



In PASCAL-like notation the algoritm, for sorting the array a[1:n], reads

{ heap creation }

$l :=$ n **div** 2 $+$ 1;

while $l \neq 1$ **do**

begin $l :=$ $l-1$; *sift(a, l, n)* **end**;

{ sorting }

$r :=$ n;

while $r \neq 1$ **do**

begin *swap(a*[1], *a*[*r*]*)*;

$r :=$ $r-1$; *sift(a*, 1, *r)*

end;

{ sift arg1 through arg2 }

$j :=$ *arg*1;

while $2j \geq$ *arg*2 **and**

(a[*j*] $<$ *a*[2*j*] **or** *a*[*j*] $<$ *a*[2*j* + 1]*)*

do begin *mi* $:=$ $2j$ $+$ **if** *a*[2*j*] $>$ *a*[2*j* + 1]

then 0 **else** 1;

exchange(a[*j*], *a*[*mi*]*)*; $j :=$ *mi*

end;

5.1 Purpose

Sorting values given in an array.

5.2 Input

The values are stored in the control sequences `\1`, ..., `\<n>`. The counter `\n` must contain the value ⟨*n*⟩. The parameter for comparison, `\cmp`, must be `\let`-equal to
- `\cmpn`, for numerical comparison,
- `\cmpw`, for word comparison,

[3] My other uses of the `\csname` construction are: to let TEX accept an outer defined macro name in a replacement text, to check whether a name has already been defined, and to mimic a switch selector.

[4] It is true that the reverse of the comparison operation would do, but it seemed more consistent to me to adapt the notation of the heap concept with the smallest elements at the bottom.

- `\cmpaw`, for word comparison obeying the ASCII ordering, or
- a comparison macro of your own.

5.3 Output

The sorted array `\1`, `\2`, ... `\<n>`, with `\value1` ≤ `\value2` ≤ ... ≤ `\value`⟨*n*⟩.

5.4 Source

```
%Non-descending sorting
\def\heapsort{%data in \1 to \n
\r\n\heap\ic1
{\loop\ifnum1<\r\xch\ic\r
   \advance\r-1 \sift\ic\r
\repeat}}
%
\def\heap{%Transform \1..\n into heap
 \lc\n\divide\lc2{}\advance\lc1
 {\loop\ifnum1<\lc\advance\lc-1
  \sift\lc\n\repeat}}
%
\def\sift#1#2{%#1, #2 counter variables
 \jj#1\uone#2\advance\uone1 \goontrue
 {\loop\jc\jj \advance\jj\jj
  \ifnum\jj<\uone
   \jjone\jj \advance\jjone1
   \ifnum\jj<#2 \cmpval\jj\jjone
         \ifnum2=\status\jj\jjone\fi\fi
   \cmpval\jc\jj\ifnum2>\status\goonfalse\fi
 \else\goonfalse\fi
\ifgoon\xch\jc\jj\repeat}}
%
\def\cmpval#1#2{%#1, #2 counter variables
%Result: \status= 0, 1, 2 if
%values pointed by
%              #1 =, >, < #2
 \ea\let\ea\aone\csname\the#1\endcsname
 \ea\let\ea\atwo\csname\the#2\endcsname
 \cmp\aone\atwo}
%
\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\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}}.
%with auxiliaries
\newcount\n\newcount\lc\newcount\r
\newcount\ic\newcount\uone
\newcount\jc\newcount\jj\newcount\jjone
\newif\ifgoon
```

Explanation.

`\heapsort` The values given in `\1`,... `\<n>`, are sorted in non-descending order.

`\heap` The values given in `\1`,... `\<n>`, are rearranged into a heap.

`\sift` The first element denoted by the first (counter) argument has disturbed the heap. Sift rearranges the part of the array denoted by its two arguments, such that the heap property holds again.

`\cmpval` The values denoted by the counter values, supplied as arguments, are compared.

Example (Numbers, words)

`\cmpn`, and `\cmpw` stand for compare numbers and words. `\prtn`, and `\prtw` stand for print numbers and words, and work the way you expect. `\accdef` takes care that accents are properly defined.

```
\def\1{314}\def\2{1}\def\3{27}\n3
\let\cmp\cmpn\heapsort
\beginquote\prtn,\endquote
%
\def\1{ab}\def\2{c}\def\3{aa}\n3
\let\cmp\cmpaw\heapsort
\beginquote\prtw,\endquote
and
\def\1{j\ij}\def\2{ge\"urm}\def\3{gar\c con}
\def\4{\'el\'eve}\n4
\let\cmp\cmpw {\accdef\heapsort}
\beginquote\prtw\endquote
```

yields within the context of `blue.tex`

1, 27, 314,

aa ab c,

and élève, garçon, geürm, jij.

6 Quick sort

The quick sort algorithm has been discussed in many places, The following code is borrowed from Bentley.[5]

procedure *QSort(low,up)*;

if *low* < *up* **then**

begin

{ choose suitable median }

Swap(X[*low*], *X*[*RandInt(low,up)*]);

T := *X*[*low*]; *M* := *low*;

{ Invariant loop

X[low + 1..M] < T and X[M + 1..I − 1] ≥ T }

for *I* := *low* + 1 **to** *up* **do**

if *X*[*I*] < *T* **then**

begin *M* := *M* + 1;

Swap(X[*M*], *X*[*I*]);

end;

{ exchange median }

Swap(X[*low*], *X*[*M*]);

{ X[low..M − 1] < X[M] ≤ X[M + 1..up] }

QSort(low, M − 1); *QSort(M* + 1, *up)*;

end;

[5] Programming Pearls, Addison-Wesley. It contains also diagrams which keep track of the invariants.

6.1 Purpose

Sorting of the values given in the array \<low>, ..., \<up>.

6.2 Input

The values are stored in \<low>, ..., \<up>, with $1 \leq low \leq up \leq n$. The parameter for comparison, \cmp, must be \let-equal to

- \cmpn, for number comparison,
- \cmpw, for word comparison,
- \cmpaw,for word comparison obeying the ASCII ordering, or
- a comparison macro of your own.

6.3 Output

The sorted array \<low>, ..., \<up>, with \val⟨*low*⟩ ≤ ... ≤ \val⟨*up*⟩.

6.4 Source

```
\def\quicksort{%Values given in
%\low,...,\up are sorted, non-descending.
%Parameters: \cmp, comparison.
 \ifnum\low<\up\else\brk\fi
%\refval, a reference value selected
%at random.
 \m\up\advance\m-\low%Size-1 of array part
 \ifnum10<\m\rnd\multiply\m\rndval
   \divide\m99 \advance\m\low \xch\low\m
 \fi
 \ea\let\ea\refval\csname\the\low\endcsname
 \m\low\k\low\let\refvalcop\refval
 {\loop\ifnum\k<\up\advance\k1
   \ea\let\ea\oneqs\csname\the\k\endcsname
   \cmp\refval\oneqs\ifnum1=\status
      \global\advance\m1 \xch\m\k\fi
   \let\refval\refvalcop
  \repeat}\xch\low\m
 {\up\m\advance\up-1 \quicksort}%
 \low\m\advance\low1 \quicksort}
%
\def\brk#1\quicksort{\fi}
```

Explanation. At each level the array is partitioned into two parts. After partitioning the left part contains values less than the reference value and the right part contains values greater than or equal to the reference value. Each part is again partitioned via a recursive call of the macro. The array is sorted when all parts are partitioned.

In the TEX coding the reference value as estimate for the mean value is determined via a random selection of one of the elements.[6] Reid's \rnd has been used. The random number is mapped into the range [*low* : *up*], via the linear transformation \low + (\up − \low) ∗ \rndval/99.[7]

The termination of the recursion is coded in a TEX peculiar way. First, I coded the infinite loop. Then I inserted the condition for termination with the \fi on the same line, and not enclosing the main part of the macro. On termination the invocation \brk gobbles up all the tokens at that level to the end, to its separator \quicksort, and inserts its replacement text, a new \fi, to compensate for the gobbled \fi.

6.5 Auxiliaries

Sorting is parameterized by comparison and exchanging. Also needed is a random number generator. The latter is not supplied here.

```
\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\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}}
```

6.6 Ordering

The ordering is parameterized in the ordering table.

Example (Numbers, words)

\cmpn, and \cmpw stand for compare numbers and words. \prtn, and \prtw stand for print numbers and words, and work the way you expect. \accdef takes care that accents are properly defined.

```
\def\1{314}\def\2{1}\def\3{27}\n3
\low1\up\n\let\cmp\cmpn
\quicksort
\beginquote\prtn,\endquote
%
\def\1{ab}\def\2{c}\def\3{aa}
\def\4{\ij}\def\5{ik}\def\6{z}\def\7{a}\n7
\low1\up\n\let\cmp\cmpw
\quicksort
\beginquote\prtw,\endquote
and
\def\1{j\ij}\def\2{ge\"urm}\def\3{gar\c con}
\def\4{\'el\'eve}\n4
\low1\up\n\let\cmp\cmpw
{\accdef\quicksort}
\beginquote\prtw.\endquote
```

yields similar results as with heap sorting.

7 Use

I needed sorting within TEX for indexing and for sorting address labels.

7.1 Sorting address labels

Suppose we wish to sort addresses on the secondary key membership number. In order to do so the index must point to the name of the database entry and the name must point to its membership number, that is

$$1\,2\,\ldots \rightarrow \texttt{<name>}_x\ \texttt{<name>}_y\ \ldots \rightarrow \texttt{<no>}_x\ \texttt{<no>}_y\ \ldots$$

This can be coded as follows.

[6] If the array is big enough. I chose rather arbitrarily 10 as threshold.

[7] Note that the number is guaranteed within the range.

```
\loadindexmacros
%
\def\lst#1#2{\advance\k1
   \ea\def\csname\the\k\endcsname{#1}%
   \ea\def\ea#1\gobbletono#2}
\def\gobbletono#1\no{}
\k0
\input toy.dat %The test database
\n\k           %number of items
Membershipno unsorted: \1, \2, ...
%
\let\cmp\cmpn\sort

Sorted on membershipno: \1, \2, ...
```

The amazing thing is that we don't have to do much extra because the name will expand to the number, which will be used in the comparison. I used that `\no` was the last element of the database entry, but that is not essential. Each database entry consist of a triple `\lst`, `\<name>`, and entry proper within braces.

7.1.1 Typesetting

Now we have to redirect the pointer from the name away from the number to the complete entry, that is

$$1\,2\,\ldots \rightarrow \texttt{\<name>}_1\ \texttt{\<name>}_2\ \ldots \rightarrow \texttt{entry}_1\ \texttt{entry}_2\ \ldots$$

This is done as follows.

```
\def\lst#1#2{\def#1{#2}}
\input toy.dat
\1 \2 \3 \4 \5 \6
```

7.2 Sorting index entries

One of the processes in preparing an index is sorting the Index Reminders, IRs. This is again a sorting process on secondary keys, even tertiary keys.

Given the sorting macros we just have to code the special comparison macro in compliance with `\cmpw`: compare two 'values' specified by `\def`s. Let us call this macro `\cmpir`.[8] Each value is composed of

- a word (action: word comparison)
- a digit (action: number comparison), and
- a page number (action: (page) number comparison).

The macros read as follows.

```
\def\cmpir#1#2{%#1, #2 defs
%Result: \status= 0, 1, 2 if
%        \val{#1} =, >, < \val{#2}
 \ea\ea\ea\decom\ea#1\ea;#2.}
%
\def\decom#1 !#2 #3;#4 !#5 #6.{%
 \def\one{#1}\def\four{#4}\cmpaw\one\four
 \ifnum0=\status%Compare second key
   \ifnum#2<#5\global\status2 \else
     \ifnum#2>#5\global\status1 \else
       %Compare third key
       \ifnum#3<#6\global\status2
       \else\ifnum#3>#6\global\status1 \fi
       \fi
     \fi
   \fi
 \fi}
```

Explanation. I needed a two-level approach. The values are decomposed into their components by providing them as arguments to `\decom`.[9] The macro picks up the components

- the primary keys, the ⟨*word*⟩
- the secondary keys, the ⟨*digit*⟩, and
- the tertiary keys, the ⟨*page number*⟩.

It compares the primary keys, and if necessary successively the secondary and the tertiary keys. The word comparison is done via the already available macro `\cmpaw`.

To let this work with `\sort`, we have to `\let`-equal the `\cmp` parameter to `\cmpir`.

8 Sorting in the mouth

Alan Jeffrey and Bernd Raichle have provided macros for this. The following variant of the linear sorting given at the beginning of this note is inspired upon Bernd's 'Quick Sort in the Mouth,' EuroTEX 94. The idea is that a sequence is split in its smallest element and the rest by an invoke of `\fifo`. The rest is treated recursively as a similar sequence. Another example of (multiple) nested FIFO.

```
\def\fifo#1%accumulated rest
         #2%smallest
         #3%next
{\ifx\ofif#3 #2\ofif{#1}\fi
 \ifnum#3<#2
   \p{\fifo{#1{#2}}{#3}}\else
   \q{\fifo{#1{#3}}{#2}}\fi}
%repeat or terminate
\def\ofif#1\fi#2\fi{\fi
   \if*#1*\endsort\fi
   \fifo{}#1\ofif}
%auxiliaries
\def\p#1\else#2\fi{\fi#1}
\def\q#1\fi{\fi#1}
%terminator
\def\endsort#1\ofif{\fi}
%test
\fifo{}3{123}8{1943}\ofif
```

To assure yourself that it is all done in the mouth `\write` the test.[10]

However, in sorting within TEX I prefer a uniform approach not in the least parameterized over the ordering table.

Have fun, and all the best

[8] Mnemonics: compare index reminders

[9] Mnemonics: decompose. In each comparison the defs are 'dereferenced,' that is their replacement texts are passed over. This is a standard TEXnique: a triad of `\ea`s, and the hop-overs to the second argument.

[10] I don't know how to ensure correctness. It is tricky to get the braces right. I used `\tracingmacros=1`.