The Arraysort Package*

Robert J Lee
latex@rjlee.homelinux.org

September 4, 2013

Abstract

The \texttt{arraysort} package allows the user to sort an array (defined with the \texttt{arrayjobx} package), or a portion of such an array, without using external files or commands or requiring a second run of \LaTeX. Basic comparators are provided for sorting by ASCII-code or for sorting numeric values. Options to tweak performance of the sort are also provided, should they be needed.

Introduction

This package implements an in-place Quick Sort algorithm for \LaTeX. QuickSort is a recursive and highly configurable algorithm for sorting of arrays.

1 Usage

The \texttt{arraysort} package requires the \texttt{arrayjobx} package, which provides several methods to define an array of values. This document assumes that you are familiar with \texttt{arrayjobx}. A number of examples are provided to sort a small array named \textit{A}; the package can sort much larger arrays than those shown including arrays whose contents are stored unexpanded (although they will be expanded when comparing them using the provided comparators).

Sorting by Text

\texttt{\texttt{\textbackslash sortArray}}

To sort the first 10 elements of \textit{A}, you can simply write: \texttt{\textbackslash sortArray}{10}\{A\}

\texttt{\textbackslash sortArray} takes two mandatory parameters: the first is the number of elements to sort, the second is the name of the array.

\footnote{This document corresponds to Arraysort v1.0, dated 2013/09/04.}
Example: \texttt{\textbackslash sortArray with words}

\begin{verbatim}
Brown Dog Fox Jumps Lazy Over
Quick The the
\end{verbatim}

Note that the default is to sort by character code order, so the lower-case “the” is sorted after the words starting with upper-case letters. This is a limitation of the default sorting method, but other ways of sorting are possible.

### Sorting Numbers

The default sorting method can have surprising results when sorting arrays of numbers:

Example: \texttt{\textbackslash sortArray with numbers}

\begin{verbatim}
1 28 4 6 78 85
\end{verbatim}

Here, the numbers are still sorted in dictionary order, which was probably not the intent, as the number 28 would usually be sorted after the number 6, even though the first digit of 28 is smaller than the first digit of 6.

To solve this problem, an alternative comparator can be used:

Example: \texttt{\textbackslash sortArray with arraysortcomparenum}

\begin{verbatim}
1 4 6 28 78 85
\end{verbatim}

The \texttt{arraysortcomparenum} comparator is passed as the first optional argument; this option sorts by numerical order in the intuitive way — but it will error if a value in the array is not a number.

This option requires additional package options; see section below.
Sorting part of an array

A second optional argument is accepted for the start of the range to be sorted. So you can easily sort only a segment of an array:

Example: \texttt{\sortArray with range}
1 newarray\{A\}
2 \readarray\{A\}\{P\&Y\&F\&G\&\%
3 A\&O\&E\&U\&I\&D\&H\&T\&N\&S\&\%
4 Q\&J\&K\&X\&B\&M\&W\&V\&Z\}
5 \sortArray[arraysortcomparestr][8]{21}\{A\}
6 \{A(1)\}\{A(2)\}\{A(3)\}\{A(4)\}\{A(5)\}\{A(6)\}\{A(7)\}\%
7 \texttt{textit}\{\{A(8)\}\{A(9)\}\{A(10)\}\{A(11)\}\{A(12)\}\%
8 \{A(13)\}\{A(14)\}\{A(15)\}\{A(16)\}\{A(17)\}\{A(18)\}\%
9 \{A(19)\}\{A(20)\}\{A(21)\}\{A(22)\}\{A(23)\}\%

The start of the range must be the second optional parameter, so you need to specify the comparator as well. The default comparator is \texttt{arraysortcomparestr} as shown here.

The example shows sorting the italicised region of 8–21 letters, originally arranged in the relatively jumbled order of the Dvorak keyboard.

Sorting by Custom Order

While it is useful to sort an array into numbers or alphabetically, it is possible to sort an array into any order. To do this, you need to write a comparator; this is a macro that is passed 2 values from the array and evaluates which should appear first.

A custom comparator macro can be passed by appending its name as the optional argument is passed at the end of the \texttt{\sortarray} macro call.

Your comparator macro must set the value of two toggles:
arraysortresult Set \texttt{arraysortresult} to true if the first parameter is less than the second (ie if the parameters are presented in sorted order).
arraysortrequal \texttt{arraysortrequal} should be set by a comparator if both values are equal.

It is not necessary to set \texttt{arraysortresult} if \texttt{arraysortrequal} is set to true.

Both toggles can be set using the macros \texttt{\toggletrue{togglename}} and \texttt{\togglefalse{togglename}} defined in the \texttt{etoolbox} package, where \texttt{togglename} is the name of the toggle to be set or cleared.

The name of the comparator is passed to the sort algorithm as the first optional parameter; the leading \texttt{\} should be omitted.
Example: \sortArray with custom comparator

\newcommand{\cmpnumbersfirst}{\%
  \edef\cmpA{\#1}\%
  \edef\cmpB{\#2}\%
  \if\IsPositive{\cmpA}\%
    \if\IsPositive{\cmpB}\%
      \arraysortcomparenum{\cmpA}{\cmpB}\%
    \else\%
      \togglefalse{\arraysortrequal}\%
      \toggletrue{\arraysortresult}\%
    \fi\%
  \else\%
    \toggletrue{\arraysortrequal}\%
    \togglefalse{\arraysortresult}\%
  \fi\%
}

2 12 -4 apple rabbits

\newarray{A}
\readarray{A}{apple&2&rabbits&12&-4}
\sortArray{\cmpnumbersfirst}{5}{A}
\A(1) \A(2) \A(3) \A(4) \A(5)

This example uses the following definition of \IsPositive from the cite package\footnote{See \url{http://www.tex.ac.uk/cgi-bin/texfaq2html?label=isitanum}}

\def\IsPositive#1{\%
  T\fi
  \ifcat_\ifnum0<0\ifnum#1<0 _\else A\fi
}

This custom sort places positive integers first, in numerical order, then everything else in default order.

The \cmpnumbersfirst macro tests each parameter to see if is a positive number. If both parameters are positive integers, then it delegates to the arraysortcomparenum macro so that integers are sorted in sequence. If both parameters are not positive integers then the default sort is used. Otherwise, arraysortresult is set to true if \#1 is a positive integer and \#2 is not, or false if it is the other way around; this guarantees that positive numbers are sorted first.
TIP: Most comparators will fully expand their arguments only once per comparison, to ensure that the sorting order remains appropriate. That is the reason for the `\edef` in the above example, which expands and copies the parameter into a temporary macro. This is useless when passing individual string arguments, as in this example, but prevents unstable behaviour when arguments could change when reevaluated, such as when a macro contains the current time or a pseudorandom value.

Changing the Partitioning Scheme

The partitioning scheme does not affect the final sorting order (unless you write your own that does not use the comparator argument) but may affect how long it takes \LaTeX{} to sort your array. In general it is recommended to use the default scheme, unless you are sorting a very large array and find the performance is unacceptable.

To change the partitioning scheme, an optional argument can be added to the end of the `\sortArray` macro, thus:

```
\example\sortArray with custom partition
\begin{verbatim}
 1 newarray{A}
 2 \readarray{A}{e&d&c&b&a}
 3 \sortArray{5}{A} %
 4 [sortArrayPartitionRand]
 5 A(1) \ A(2) \ A(3) \ A(4) \ A(5)
\end{verbatim}
```

Here, the writer of the package knows that \( A \) is not randomised before it is sorted, so uses the `sortArrayPartitionRand` to partition the array at random. This avoids the worst-case performance of the sorting algorithm.

The performance of each partitioning method is discussed in detail where it is defined; you should also read the section on the in-place quick-sort algorithm to understand the purpose of each algorithm.

The partition name is just the name of a macro, so you can easily write your own. It must take four parameters:

1. The name of a comparator macro, described above
2. The start index (inclusive) of the array segment to be partitioned
3. The end index (inclusive) of the array segment to be partitioned
4. The name of the array to be partitioned

---

5
The macro should generate no output as it may be called multiple times with different array segments to partition. It should set `arraysort@partpos` to the current value of the partition element.

Values that are equal to the partition value may be sorted into either segment. A sorting algorithm that retains the relative order of equal values is known as a stable sorting algorithm; if this is required, then the partitioning algorithm must retain the relative position of each element in each sub-array, not just those which are equal to the pivot. **The supplied partitioning algorithms make no claim to be a stable sort**, and stable sort semantics of the partitioning algorithm should **not be relied on for future versions**.

In general, it is sufficient to identify the partition element within the array and swap it with the first element, then use `\sortArrayPartitionFirst`

**Note:** It is important to ensure that all elements are expanded only once during the partition, as it is theoretically possible for a macro to expand to different values each time it is expanded.

### All Package Options

Package options are passed on the `\usepackage` line near the top of your document. A comma-separated list may be supplied, like this:

`\usepackage[comparestr,comparenum,randompart]{arraysort}`

**comparestr**  
The `comparestr` option requires the `pdftexcmds` package to be installed and to run `pdflatex`. It is currently the default sort option, so you must either supply the `comparestr` option or specify a comparator explicitly.

**comparenum**  
The `comparenum` option defines the `arraysortcomparenum` comparator, which allows you to sort arrays comparing numbers by numeric value instead of by name.

**randompart**  
The `randompart` macro requires the `lcg` package to be installed. It defines the `\sortArrayPartitionRand` option to partition arrays using a pseudorandom sequence. This option repeatedly calls `\reinitrand`, which resets the value of the pseudorandom sequence as well as the maximum and minimum values generated, so you should take care if using the `lcg` package outwith this package. At minimum, you should call `\reinitrand` yourself after every sort, and possibly within macros if `\rand` is used. Be aware that this will output whitespace unless care is taken to consume it before it is output. `lcg` will output warnings about reusing an existing counter; these can be safely ignored as `\sortArrayPartitionRand` intentionally reuses the counter to prevent exhaustion of counters with large sorts.
2 Method: The In-Place Quick-Sort Algorithm

Quick-sort is a practical example of a recursive algorithm.

Definition of terms:

**Partition** A single element from the array.

**Segment** A contiguous group of elements from the array.

The general approach is to divide the array into two smaller arrays, then sort each smaller array in turn.

The initial array segment is the entire array.

The basic steps are:

1. Determine $A_P$, the index of the partition element within the current array segment. In practice, this must be done in constant time $O(1)$ or the sort becomes slow.

2. Partition the array into two array segments, separated by a partition, in linear time $O(N)$.

3. Quick-sort the first array segment

4. Quick-sort the second array segment

The first step is choosing the partition element, $A_P$. This may be any element from the array segment, although the fastest results are achieved by selecting the median value. Many algorithms exist to perform this “best guess”.

The next step is partitioning. Other than the partition element, every element in the array is iterated over in turn. Any value less than the partition value $P$ is moved to the left of the partition (lower index than $A_P$), and any value greater than the partition is moved to its right (higher index than $A_P$).

So, if there are $N$ elements in an array $A$, the original array is given by:

$$A_0 \ldots A_n$$

After the partitioning stage, the partitioned array is given by:

$$A_0 \ldots A_{(P-1)}, A_P, A_{(P+1)} \ldots A_N$$

where $A_P$ is the partition and $P$ is the index of the partition.

$A_0 \ldots A_{(P-1)}$ defines the first array segment to be sorted and $A_{(P+1)} \ldots A_N$ defines the second.

Each array segment is considered to be sorted if it contains one element or less; otherwise, each are presented to the entire quick-sort algorithm to be sorted.

Each iteration through the algorithm divides the array into smaller segments, each of which is always sorted relative to the other segments. Once the segment size is as small as one element, the entire array is sorted.
3 Possible Future Improvements

- It should be possible to sort macro contents by their unexpanded values.

- It may also be possible to sort macro contents in a case-insensitive manner (depending on language). Note that sorting mixed-language, mixed-alphabet content in a standard-compliant manner is not always possible.

- Further speed improvements are possible; in particular, it is often faster to defer to a lower-overhead $O(n^2)$ sorting algorithm when the number of array segment elements is smaller than some threshold value (quick-sort is inefficient at sorting small arrays, but often produces many of them to be sorted).

- More sorting and partitioning options. I am undecided about passing options versus simply defining multiple macros; certainly the chance of a name collision is very small with the naming convention used so far.

- Support partition values not in the array:

Some implementations of quicksort allow for a partition value $P$ that does not correspond to a value in the array, dividing the array into:

$$s \ A_0 \ldots A_i, A_{i+1} \ldots A_N$$

where $i$ is arbitrary. This is usually less efficient, as there is an additional value to be sorted with each iteration and hence a greater number of iterations in the best case.

In some cases, it is not possible to know the best partition value within an array segment, or a single partition may not be applicable (such as an array containing only two distinct values); however, there may be some knowledge of the array’s distribution. In the best case, a pre-calculated median of the array’s contents might already be available prior to sorting, which would be the ideal partition value for the first iteration but would be unlikely to be a value in the array, let alone a value of known position.

This would likely require significant changes to the algorithm.

4 Large-Scale Sorting

*Example:* $\texttt{\textbackslash sortArray on a large scale}$
Before sort:

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1663422405</td>
<td>1198227383</td>
<td>1653451356</td>
</tr>
<tr>
<td>1118531306</td>
<td>83797298</td>
<td>1779381895</td>
</tr>
<tr>
<td>214224337</td>
<td>1285822781</td>
<td>695523700</td>
</tr>
<tr>
<td>913318473</td>
<td>2077933796</td>
<td>1454225052</td>
</tr>
<tr>
<td>649045651</td>
<td>1440796438</td>
<td>440113088</td>
</tr>
<tr>
<td>1046972942</td>
<td>2140699517</td>
<td>1943227222</td>
</tr>
<tr>
<td>888599772</td>
<td>1095069960</td>
<td>905946124</td>
</tr>
<tr>
<td>577432032</td>
<td>418706180</td>
<td>2038322882</td>
</tr>
<tr>
<td>1781310659</td>
<td>418706180</td>
<td>2038322882</td>
</tr>
<tr>
<td>1433524024</td>
<td>619218869</td>
<td>505761115</td>
</tr>
<tr>
<td>586768173</td>
<td>567759781</td>
<td>345221316</td>
</tr>
</tbody>
</table>

After sort:

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>544921</td>
<td>597045</td>
<td>654834</td>
</tr>
<tr>
<td>1288256</td>
<td>1641621</td>
<td>1720755</td>
</tr>
<tr>
<td>108306978</td>
<td>984869468</td>
<td>204464444</td>
</tr>
<tr>
<td>641923787</td>
<td>2002712422</td>
<td>2076460317</td>
</tr>
<tr>
<td>311783616</td>
<td>287118626</td>
<td>206975567</td>
</tr>
<tr>
<td>1862313270</td>
<td>324957059</td>
<td>502359486</td>
</tr>
<tr>
<td>139768039</td>
<td>109443781</td>
<td>...</td>
</tr>
</tbody>
</table>

This example uses the `multido` and `lcg` packages.

5 Implementation

The \texttt{\LaTeX} package-option support does not allow conditional includes of packages. So, instead, we build up the required `\RequiresPackage` statements inside the \texttt{\arraysort@extrapkgs} macro.

\begin{verbatim}
  \newcommand*{\arraysort@extrapkgs}{% \arraysortextrapkgs}

1 \newcommand*{\arraysortextrapkgs}{}\end{verbatim}
\ DeclareOption{comparestr}{
\ g@addto@macro{\arraysort@extrapkgs}{
\ RequirePackage{pdftexcmds}% for comparison. TODO: use compare.sty optionally
}
\ arraysortcomparestr

Called with two arguments, guaranteed to be re-evaluatable. must set arraysortre-
sequal if arguments are considered equal, otherwise must set arraysortresult true
if #2 is to be sorted after #1, otherwise must set both flags false.

Basic ASCII-like comparison
\ newcommand*{\arraysortcomparestr}[2]{%
\ protected@edef{\arraysort@left}{#1}%
\ protected@edef{\arraysort@right}{#2}%
\ arraysortcomparestr%
}%

The following macro performs the comparison. The parameters must (it seems)
be passed by macro as passing by parameter #1 and #2 did not cause the expected
results, hence the extra macro.
\ newcommand*{\arraysortcomparestr}{%
\ protected@edef{\arraysort@compresult}{\pdf@strcmp{\arraysort@left}{\arraysort@right}}%
\ ifthenelse{equal{\arraysort@compresult}{0}}{%
\ toggletrue{arraysortresequal}%
%}
\ togglefalse{arraysortresequal}%
\ ifthenelse{#2 > #1}{%
\ toggletrue{arraysortresult}% #2 > #1
%}
\ togglefalse{arraysortresult}%
% #2 < #1
%
}%}

\ arraysortcomparenum

\ DeclareOption{comparenum}{
\ newcommand*{\arraysortcomparenum}[2]{%
\ ifthenelse{equal{#1}{#2}}{%
\ toggletrue{arraysortresequal}%
%}
\ togglefalse{arraysortresequal}%
\ ifthenelse{#2 > #1}{%
\ toggletrue{arraysortresult}% #2 > #1
%}
\ togglefalse{arraysortresult}%
% #2 < #1
%
}%}
All partitioning algorithms should complete in O(1) time; that is, they should not iterate over the array, or do anything that takes longer the more elements there are.

\texttt{\textbackslash sortArrayPartitionMed}\hspace{1em} Partition the segment consisting of indexes #2–#3 (inclusive) of array named #4, using comparator #1

Use the median of the first, last and middle values. While this has extra overhead compared to \texttt{sortArrayPartitionFirst}, it is guaranteed to avoid the worst-case performance of that method. If the array is randomly shuffled prior to sorting, this usually offers the best performance. This is the default method.

Performance depends on the comparison macro. May not work well if there are many duplicate values.

```latex
\newcommand*{\sortArrayPartitionMed}[4]{%
\setcounter{arraysort@temp1}{(#2 + #3) / 2}%
\edef\arraysort@midpos{\arabic{arraysort@temp1}}%
\testarray{#4}(#2)\protected@edef\arraysort@left{\temp@macro}%
\testarray{#4}(#3)\protected@edef\arraysort@right{\temp@macro}%
\csname#1\endcsname{\arraysort@left}{\arraysort@mid}%
\iftoggle{arraysortresequal}{%
left = mid if any two are the same, there can be no median, so may as well leave alone
}%
\iftoggle{arraysortresult}{%
left < mid
}%
\testarray{#4}(#2)\protected@edef\arraysort@left{\temp@macro}%
\csname#1\endcsname{\arraysort@left}{\arraysort@right}%
\iftoggle{arraysortresequal}{%
(left = right) < mid
}%
\iftoggle{arraysortresult}{%
left < mid, left < right
}%
\csname#1\endcsname{\arraysort@mid}{\arraysort@right}%
\iftoggle{arraysortresequal}{%
left < (mid = right)
}%
\iftoggle{arraysortresult}{%
left < mid < right
}%
\arraysort@swap{#4}{#2}{\arraysort@midpos}%
}%
```
left < right < mid

\arraysort\swap{#4}{#2}{#3}\
\%\
\%
\%

left < mid, left > right
left is already in the middle; leave alone

\%
\%
\%

left > mid
\csname#1\endcsname{\arraysort\mid}{\arraysort\right}\
\iftoggle{arraysort\resequal}{%\left > (mid = right)\%
\%
\iftoggle{arraysort\result}{%\left > right > mid\
\arraysort\swap{#4}{#2}{#3}\
\}\
\%
\%
\%
\sortArrayPartitionFirst{#1}{#2}{#3}{#4}\
}\%
\%
\%
\%
\sortArrayPartitionRand

Partition the sub-array consisting of indexes #2–#3 (inclusive) of array named #4, using comparator #1.
Use the lcg package to generate a (pseudo)-random partition value. This should perform reasonably well most of the time, and you can simply re-run LaTeX if the performance is unacceptable.
Caution: this macro will re-initialise the LCG package.

\begin{Verbatim}
\DeclareOption{randompart}{\g@addto@macro{\arraysort@extrapkgs}{
Store for later execution the fact that we will need the lcg package for random numbers
\RequirePackage[quiet]{lcg}
\}\
\newcommand*{\sortArrayPartitionRand}[4]{%}
\end{Verbatim}
It is necessary to change the start and end values of the sequence; the only way to do this is by reinitialising `lcg`. There are 2 possible problems; firstly, reinitrand outputs whitespace; and secondly it prints out a warning about a re-used counter. It’s actually best to re-use the counter, but there’s no way to silence the warning.

```
\reinitrand[\text{counter=arraysort@temp1, first=#2, last=#3, quiet=y}]%
\rand%
\arraysort@swap{#4}{#2}{\arabic{arraysort@temp1}}%
\sortArrayPartitionFirst{#1}{#2}{#3}{#4}%
```

\textbf{\texttt{\sortArrayPartitionMid}} 
Partition the sub-array consisting of indexes \texttt{#2–#3} (inclusive) of array named \texttt{#4}, using comparator \texttt{#1}

This implementation uses the middle value in the array segment. This is generally the best option if you don’t know anything about the array’s contents; in particular, it offers reasonable speed when attempting to re-sort previously-sorted arrays.

```
\newcommand*{\sortArrayPartitionMid}[4]{%
\setcounter{arraysort@temp1}{(#2 + #3) / 2}%
\arraysort@swap{#4}{#2}{\arabic{arraysort@temp1}}%
\sortArrayPartitionFirst{#1}{#2}{#3}{#4}%
}
```

\textbf{\texttt{\sortArrayPartitionFirst}} 
Partition the array segment consisting of indexes \texttt{#2–#3} (inclusive) of array named \texttt{#4}, using comparator \texttt{#1}

This implementation uses the first value in the array segment. This is fastest in theory, but only if the array is pre-shuffled. This has the worst performance when attempting to sort an already-sorted array.

```
\newcommand*{\sortArrayPartitionFirst}[4]{%
\setcounter{arraysort@partpos}{#2}%
\setcounter{arraysort@temp1}{#2 + 1}%
\setcounter{arraysort@endpos}{#3 + 1}%
\arraysort@repeats{arraysort@temp1}{\value{arraysort@temp1}}{\value{arraysort@endpos}}{1}{%
\testarray{#4}{\arabic{arraysort@temp1}}%
if the value \texttt{A_{\text{temp1}} is less than partition A_{\text{P}}, decrement the partition counter by 1 and swap.}
\let \texttt{copies without expanding:}
\let \texttt{\arraysort@cur\temp@macro}%
\let \texttt{\testarray{#4}{\arabic{arraysort@partpos}}}%
Expand the macros only once just in case they would be different on subsequent expansion:
\protected@edef \arraysort@left{\arraysort@cur}%
\protected@edef \arraysort@right{\temp@macro}%
\csname#1\endcsname{\arraysort@left}{\arraysort@right}% #2 = cur, #3 = partition
\setcounter{arraysort@temp2}{\value{arraysort@partpos} + 1}%
\iftoggle{arraysortresequal}{% #2 = #3
```
Must be moved before pivot. Swap $A_P$ with $A_{P+1}$ then swap (the new) $A_P$ with current ($A_{\text{temp}2} = A_{P+1}$)

\arraysort@swap{#4}{{\arabic{arraysort@partpos}}}{{\arabic{arraysort@temp2}}}%
\arraysort@swap{#4}{{\arabic{arraysort@partpos}}}{{\arabic{arraysort@temp1}}}%

Increment partition; otherwise the next non-equal pivot will break
\stepcounter{arraysort@partpos}%

Just swap part with current value; they are adjacent
\arraysort@swap{#4}{{\arabic{arraysort@partpos}}}{{\arabic{arraysort@temp1}}}%
\arraysort@swap{#4}{{\arabic{arraysort@partpos}}}{{\arabic{arraysort@temp2}}}%
\arraysort@swap{#4}{{\arabic{arraysort@partpos}}}{{\arabic{arraysort@temp2}}}%
\arraysort@swap{#4}{{\arabic{arraysort@partpos}}}{{\arabic{arraysort@temp1}}}%

Increment partition; one more in left array segment
\stepcounter{arraysort@partpos}%

$A_{\text{arraysort@cur}} > A_P$ and already after it; leave it alone
\}%
\}%
\}%
\}%

\ProcessOptions
This processes the package include options, defining whichever of the above macros the user has asked for, and adding a list of any optional packages into \arraysort@extrapkgs.

\ProcessOptions\relax

Package includes for required packages:
For loading the arrays that we will sort
\RequirePackage{arrayjobx}
For easier syntax on counter operations
\RequirePackage{calc}
For comparisons
\RequirePackage{ifthen}
Toggles etc.
\RequirePackage{etoolbox}
To declare macros with multiple optional arguments (ie \sortArray)
\RequirePackage{xargs}
For partitioning
\RequirePackage{macroswap}
now process any conditional includes

\arraysort@extrapkgs

Now we are done including packages, so discard the macro:

\let\arraysort@extrapkgs\relax

\sortArray

Sort the elements at index \#2–\#3 of array named \#4, using comparator \#1.

\#5 is the partitioning algorithm to use.

\eg \sortArray[1]{3}{ABC}

Defined using the \texttt{xargs} package

\newcommandx*{\sortArray}[5][1=arraysortcomparestr,2=1,5=sortArrayPartitionMed]{%

\ifcsname#1\endcsname%
\ifthenelse{#2>0}{%
\ifthenelse{#3>#2}{%
\ifcsname total@#4\endcsname%
\arraysort@sort{#1}{#2}{#3}{#4}{#5}%
\else%
\PackageError{arraysort}{Cannot sort unknown array \#4}{%}
\fi%
\else%
\PackageError{arraysort}{Cannot sort; to index \#3 greater than from index \#2}{%}
\fi%
\}%
\PackageError{arraysort}{Cannot sort; Invalid from index \#2}{%}
\else%
\PackageError{arraysort}{Cannot sort by undefined comparator \#1}{%}
\fi%
%
\arraysort@sort

As \texttt{\sortArray}, except that it doesn’t validate its parameters, hence the @ in the name (signifying an internal macro).

Use with caution as error messages may be misleading.

Sort the elements at index \#2–\#3 of array named \#4, using comparator \#1

\#5 is the partitioning algorithm to use.

\newcommand*{\arraysort@sort}[5]{%

\csname#5\endcsname{#1}{#2}{#3}{#4}%

Keep the position on the local stack!

\edef\arraysort@partition{\value{arraysort@partpos}}%

\setcounter{arraysort@temp1}{\arraysort@partition - 1}%

\ifthenelse{\#2 = \value{arraysort@temp1} \OR \#2 > \value{arraysort@temp1}}{%
\edef\arraysort@to{\arabic{arraysort@temp1}}%
\arraysort@sort[#1]{#2}{\arabic{arraysort@to}}{#4}{#5}%
\}%

\edef\arraysort@to{\arabic{arraysort@temp1}}%

\setcounter{arraysort@temp1}{\arraysort@partition + 1}%

\ifthenelse{\value{arraysort@temp1} = \#3 \OR \#3 < \value{arraysort@temp1}}{%
\}%

\edef\arraysort@from{\arabic{arraysort@temp1}}%
\arraysort{\#1}{\arraysort@from}{\#3}{\#4}{\#5}\
\}

Counters used for sorting

\newcounter{arraysort@partpos}
current position of the partition
\newcounter{arraysort@temp1}
partition position +1
\newcounter{arraysort@temp2}
used for partitioning
\newcounter{arraysort@endpos}
Toggles used by the comparator macro
set by comparison if \#1 < \#2
\newtoggle{arraysortresult}
set by comparison if \#1 = \#2
\newtoggle{arraysortresequal}

\arraysort@repeats
Don’t use \whiledo here because it uses up TeX’s capacity, so rolling own basic
repeat loop...
For counter \#1 from \#2 to \#3 step \#4, do \#5
\newcommand*{\arraysort@repeats}[5]{%
\setcounter{#1}{#2}%
\ifthenelse{\equal{\value{#1}}{#3}}{%
\addtocounter{#1}{#4}%
\arraysort@repeats{#1}{\arabic{#1}}{#3}{#4}{#5}%
\}}%
\}

\arraysort@swap
Globally swap array values \#1(#2) with \#1(#3)
ica
\newcommand{\arraysort@swap}[3]{%
\gmacroswap{#1#2\string~}{#1#3\string~}%
\}
arrayjobx does not provide a way to assign an array element to the contents of
another element (or macro) without expanding it. This macro simply swaps the
definitions of the two macros used internally by arrayjobx:
\gmacroswap{#1#2\string``}{#1#3\string``}%
\}

That is all
Change History

v1.0
General: Initial version . . . . . . . . 1
Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

| A | \arrayort@repeats . 176 |
|   | \arraysort@comparestr     |
|   | \arraysort@cur            |
|   | \arraysort@extrapkgs      |
|   | \arraysort@from           |
|   | \arraysort@left           |
|   | \arraysort@mid            |
|   | \arraysort@midpos         |
|   | \arraysort@partition      |
|   | \arraysort@right          |
|   | \arraysort@sort           |
|   | \arraysort@swap           |
|   | \arraysort@to             |

| B | \arraysortcomparenum       |
| C | \arraysortcomparestr       |
| D | \arraysortcomresult        |
| G | \comparenum                |
|   | \comparestr                |
| H | \DeclareOption             |
|   | \pdf@strcmp                |
|   | \PackageError             |
|   | \pdfstrcmp                 |
|   | \ProcessOptions           |
|   | \protected@edef           |

| P | \rand                      |
|   | \randompart               |
|   | \reinitrand               |
|   | \requirepackage           |
|   | \temp@macro               |
|   | \testarray                |
|   | \togglefalse              |
|   | \toggletrue               |

18