The Arraysort Package*

Robert J Lee
latex@rjlee.homelinux.org

September 4, 2013

Abstract

The `arraysort` package allows the user to sort an array (defined with the `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 `arraysort` package requires the `arrayjobx` package, which provides several methods to define an array of values. This docment assumes that you are familiar with `arrayjobx`. A number of examples are provided to sort a small array named 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

`\sortArray`

To sort the first `10` elemnets of A, you can simply write: `\sortArray{10}{A}`

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

---

[*] This document corresponds to Arraysort v1.0, dated 2013/09/04.

Example: `\sortArray with words`

Brown Dog Fox Jumps Lazy Over Quick The the

```
\newarray{A}
\readarray{A}{The&Quick&Brown&%
Fox&Jumps&Over&the&Lazy&Dog}
\sortArray{9}{A}
\A(1) \A(2) \A(3) \A(4) \A(5)
\A(6) \A(7) \A(8) \A(9)
```

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: `\sortArray with numbers`

1 28 4 6 78 85

```
\newarray{A}
\readarray{A}{78&4&85&1&28&6}
\sortArray{6}{A}
\A(1) \A(2) \A(3) \A(4) \A(5) \A(6)
```

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: `\sortArray with arraysortcomparenum`

1 4 6 28 78 85

```
\newarray{A}
\readarray{A}{78&4&85&1&28&6}
\sortArray[arraysortcomparenum]{6}{A}
\A(1) \A(2) \A(3) \A(4) \A(5) \A(6)
```

The `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: \\`sortArray with range`

PYFGAOE*BDHIJKMNQ*STUWXVZ

```
\newarray{A}
\readarray{A}{P&Y&F&G&%
A&O&E&U&I&D&H&T&N&S&%
Q&J&K&X&B&M&W&V&Z}
\sortArray[arraysortcomparestr][8]{21}{A}
\A(1)\A(2)\A(3)\A(4)\A(5)\A(6)\A(7)%
\textit{\A(8)\A(9)\A(10)\A(11)\A(12)%
\A(13)\A(14)\A(15)\A(16)}\A(17)\A(18)%
\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 `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 comparotor macro can be passed by appending its name as the optional argument is passed at the end of the `\sortarray` macro call.

Your comparator macro must set the value of two toggles:

`arraysortresult` Set `arraysortresult` to true if the first parameter is less than the second (*ie* if the parameters are presented in sorted order).

`arraysortresequal` `arraysortresequal` should be set by a comparator if both values are equal. It is not necessary to set `arraysortresult` if `arraysortresequal` is set to true.

Both toggles can be set using the macros `\toggletrue{`*togglename*`}` and `\togglefalse{`*togglename*`}` defined in the `etoolbox` package, where *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 \ should be omitted.

Example: `\sortArray with custom comparator`

```
\newcommand{\cmpnumbersfirst}[2]{%
  \edef\cmpA{#1}%
  \edef\cmpB{#2}%
  \if\ IsPositive {\cmpA}%
    \if\ IsPositive {\cmpB}%
      \arraysortcomparenum{\cmpA}{\cmpB}%
    \else%
      \ togglefalse {arraysortresequal}%
      \toggletrue{ arraysortresult }%
    \fi %
  \else%
    \if\ IsPositive {\cmpB}%
      \ togglefalse {arraysortresequal}%
      \ togglefalse { arraysortresult }%
    \else%
      \arraysortcomparestr{\cmpA}{\cmpB}%
    \fi %
  \fi %
}
\newarray{A}
\readarray{A}{apple&2&rabbits&12&-4}
\sortArray[cmpnumbersfirst]{5}{A}
\A(1) \A(2) \A(3) \A(4) \A(5)
```

```
2 12 -4 apple rabbits
```

This example uses the following definition of `\IsPositive` from the `cite` package[1]

```
\def\IsPositive#1{%
  TT\fi
  \ifcat_\ifnum0<0#1 _\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.

---

[1] See `http://www.tex.ac.uk/cgi-bin/texfaq2html?label=isitanum`

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

```
a b c d e
```

```
\newarray{A}
\readarray{A}{e&d&c&b&a}
\sortArray{5}{A}%
          [sortArrayPartitionRand]
\A(1) \A(2) \A(3) \A(4) \A(5)
```

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

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 inital 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-complient manner is not always possible.
- Further speed improvemnts 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 partitining 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, divding the array into:

$$s \quad A_0 \ldots A_i, A_{(i+1)} \ldots \; A_N$$

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: `\sortArray on a large scale`

```
Before sort:
1663422405  1198227383  1653451356
1118531306    83797298  1779381895
214224337   1285822781   695523700
913318473   2077933796  1454225052
649045651   1440796438   440113088
1046972942  2140699517  1943227222
888599772   1095069960   905946124
577432032    421544225   345221316
1781310659   418706180  2038322882
1433524024   619218869   505761115
586768173    567759781  1068778840
1412723566  1065755724  2142820482
1083063978   984869468  2044664441
641923787   2002712422  2076460317
311783616    287118626   206975567
1862313270   324957059   502359486
1397648039  1094443781 ...

After sort:
544921    597045    654834    837130
1288256   1641621   1720755   2735418
2783139   2787241   2969413   3018282
3421638   3520977   3686589   3729320
3765849   4015197   4270493   4780611
4962951   5124989   5711085   6038417
6110593   6218449   6388226   6528730
6656722   7326515   7328075   7429679
7823021   7859575   8222801   8255806
8491502   8499115   8807286   8875754
9141432   9176485   9657300   9853381
10025395      10166867      10244461
10351310  10527702  10693570 ...
```

```
\newarray{A}
\expandarrayelementtrue
\reinitrand[counter=rand,quiet=y]
\newcommand{\asize}{10000}
\multido{\i=1+1}{\asize}{
 \rand
 \A(\i)={\arabic{rand}}
}

\textbf{Before sort:}

\multido{\i=1+1}{50}{
 \A(\i)
}\dots

\sortArray[arraysortcomparenum]{%
  \asize}{A}

\line(1,0){100}

\textbf{After sort:}

\multido{\i=1+1}{50}{
 \A(\i)
}\dots
```

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

---

5 Implementation

`\arraysort@extrapkgs`

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

```
\newcommand*{\arraysort@extrapkgs}{}
```

`comparestr`

```
\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 arraysortresequal 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}%
    \arraysort@comparestr%
  }
```

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*{\arraysort@comparestr}{%
    \protected@edef\arraysort@compresult{\pdf@strcmp{\arraysort@left}{\arraysort@right}}%
    \ifthenelse{\equal{\arraysort@compresult}{0}}{%
      \toggletrue{arraysortresequal}%
    }{%
      \togglefalse{arraysortresequal}%
      \ifthenelse{\equal{\arraysort@compresult}{-1}}{%
        \toggletrue{arraysortresult}% #2 > #1
      }{%
        \togglefalse{arraysortresult}% #2 < #1
      }%
    }%
  }
}
```

`comparenum`

```
% Numeric comparison, as |\arraysortcomparestr| but used with arrays compairng numbers
\DeclareOption{comparenum}{
```

`\arraysortcomparenum`

```
  \newcommand*{\arraysortcomparenum}[2]{%
    \ifthenelse{\equal{#1}{#2}}{%
      \toggletrue{arraysortresequal}%
    }{%
      \togglefalse{arraysortresequal}%
      \ifthenelse{#2 > #1}{%
        \toggletrue{arraysortresult}%
      }{%
        \togglefalse{arraysortresult}%
      }%
    }%
  }
}
```

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.

`\sortArrayPartitionMed` 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 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.

```
\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}(\arraysort@midpos)\protected@edef\arraysort@mid{\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

```
  }{%
```

left $\neq$ mid

```
    \iftoggle{arraysortresult}{%
```

left $<$ mid

```
      \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{arraysortresequal}{%
```

left > (mid = right)

```
   }{%
    \iftoggle{arraysortresult}{%
```

left > right > mid

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

swap right & left, so left is median

```
    }{%
```

left > mid > right

swap right & mid, so left is median

```
     \arraysort@swap{#4}{#2}{\arraysort@midpos}%
    }%
   }%
  }%
 }%
 \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.

`randompart`

```
\DeclareOption{randompart}{
 \g@addto@macro{\arraysort@extrapkgs}{
```

Store for later execution the fact that we will need tho `lcg` package for random numbers

```
  \RequirePackage[quiet]{lcg}
 }
 \newcommand*{\sortArrayPartitionRand}[4]{%
```

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

`\sortArrayPartitionMid`

Partition the sub-array consisting of indexes `#2`–`#3` (inclusive) of array named `#4`, using comparator `#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 ararys.

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

`\sortArrayPartitionFirst`

Partition the array segment consisting of indexes `#2`–`#3` (inclusive) of array named `#4`, using comparator `#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 A_{temp1} is less than partition A_P, decrement the partition counter by 1 and swap.

`\let` copies without expanding:

```
    \let\arraysort@cur\temp@macro%
    \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 ($A_{\text{temp2}} = A_{P+1}$)

```
   \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}%
  }{%
   \iftoggle{arraysortresult}{% #3 > #2
    \ifthenelse{\equal{\arabic{arraysort@temp2}}{\arabic{arraysort@temp1}}}{%
```

Just swap part with current value; they are adjacent

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

Swap A_P with A_{P+1} then swap (the new) A_P with current ($\text{temp}_2 = A_{p+1}$)

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

Increment partition; one more in left array segment

```
    \stepcounter{arraysort@partpos}%
   }{%
```

$A_{\texttt{arraysort@cur}} > A_P$ 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 `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%
    }{%
    \PackageError{arraysort}{Cannot sort; to index #3 greater than from index #2}{}%
    }%
  }{%
    \PackageError{arraysort}{Cannot sort; Invalid from index #2}{}%
  }%
  \else%
    \PackageError{arraysort}{Cannot sort by undefined comparator #1}{}%
  \fi%
}
```

`\arraysort@sort`

As `\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}{\arraysort@to}{#4}{#5}%
  }%
  \setcounter{arraysort@temp1}{\arraysort@partition + 1}%
  \ifthenelse{\value{arraysort@temp1} = #3 \OR #3 < \value{arraysort@temp1}}{%
  }{%
    \edef\arraysort@from{\arabic{arraysort@temp1}}%
    \arraysort@sort{#1}{\arraysort@from}{#3}{#4}{#5}%
  }%
}
```

Counters used for sorting
current position of the partition

```
\newcounter{arraysort@partpos}
```

current position in loop

```
\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}
```

`\arrayort@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}}{%
  }{%
    #5%
    \addtocounter{#1}{#4}%
    \arraysort@repeats{#1}{\arabic{#1}}{#3}{#4}{#5}%
  }%
}
```

`\arraysort@swap` Globally swap array values `#1(#2)` with `#1(#3)`

ie

`#1` is the macro name
`#2` and `#3` are the numeic indexes of the array elements to be swapped.

```
\newcommand\arraysort@swap[3]{%
```

`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.