**set sortrngstate** — Set the state of sort's randomizer

Description Syntax Remarks and examples Also see

# Description

Stata's `sort` command jumbles the data into a random order before sorting it. This means that tied values of the sort variables may result in observations being sorted differently across multiple runs of the same `sort` command. `set sortrngstate`, combined with the c-class system parameter `c(sortrngstate)`, allows you to hold and restore the state of the randomizer (really a jumbler) that `sort` uses prior to sorting. It affects the commands `sort` and `gsort`, as well as any commands that use sorting as part of their computation.

This is a low-level utility that even the most sophisticated programmers are unlikely to need. If you are looking to perform reproducible sorts when you have tied values of the key variables for some observations, instead see *Sorting with ties* in [D] **sort**.

# Syntax

`set sortrngstate` #

`set sortrngstate` *statecode*

`local` *statecode* `= c(sortrngstate)`

# is any number between 0 and $2^{31} - 1$ (or 2,147,483,647).

*statecode* is a random-number state previously obtained from `creturn` value `c(sortrngstate)`.

# Remarks and examples

Remarks are presented under the following headings:

*Holding and restoring the jumbler state*
*Reproducibility*

## Holding and restoring the jumbler state

`sort` prejumbles the data using a simple and fast pseudo–random process. This dramatically enhances the performance of `sort` if the original ordering has long runs of presorted key variables. Because the jumbler is a pseudo–random process, it has an internal state that determines how the jumbling will occur. This state can be fetched by typing

```
. local mysortstate = c(sortrngstate)
```

Then, after you have performed some other sorts, you can type

```
. set sortrngstate 'mysortstate'
```

to reset the state of sort's jumbler. Then sort will act as though those other sorts had not occurred and henceforth prejumble the data as though you had never performed those other sorts.

If this all seems esoteric, it is. set sortrngstate exists solely so we at StataCorp could continue to use numerous certification scripts while inserting new certification scripts amongst the existing scripts. There is a better way to do that; again see *Sorting with ties* in [D] **sort**, but the die was already cast. We wanted to "save" several thousand existing scripts.

## Reproducibility

In addition to accepting a stored state, set sortrngstate can accept a seed, much like a pseudo-random-number generator.

You can, for example, type

```
. set seed 12345
```

to set the internal state of the jumbler to a value derived from 12345.

Every time you type

```
. set seed 12345
```

the jumbler will be set to the same value and will jumble in exactly the same way.

This can be used as a crude way to coax sort into performing the same sort even when there are observations with ties in the key variables, that is to say, when the sort order is not uniquely defined by the keys. That is indeed a crude form of reproducibility but not a good form of reproducibility. Do not rely on this.

Here are just a few of the problems with such an approach.

First, it is only reproducible when your data always start in exactly the same order prior to each sort. If x1 has observations with the same value and you type

```
. set seed 845
. sort x1
...
. sort x2
...
. set seed 845
. sort x1
```

Your data will not be in the same order after the first and third sorts. How the jumbler works depends on both its state **and** the current order of the data.

Second, and a corollary of the first reason, you cannot easily recover the ordering from sort. You may not think so today, but someday you are likely to wish to retrieve the same order that one of your sorts created. You can only do that reliably if the original sort produced a unique ordering.

Third, sorts that produce a nonunique ordering of the data are obscuring something about your data, your analysis, or your method. Many data manipulations depend on sorts being specified fully and correctly, and setting sortrngstate can disguise logic problems in data management. Even if the ties "do not matter" and should be broken randomly, there are better ways to do that than to rely on a jumbler whose sole job is to be fast. sort's jumbler does not have the nice statistical properties of the random-number generator behind runiform() and other Stata random-number generators.

Fourth, it is crucial that `sort` be fast, and Stata makes no attempt to blunt that speed with false reproducibility. Stata/SE and Stata/MP use the jumbler differently and so produce different orderings of ties, even when starting from the same seed/state. What's more, Stata/MP with two processors and Stata/MP with four processors also produce different orderings of ties. The older `qsort` (prior to Stata 17) and the newer `fsort` (Stata 17 and beyond) also use the jumbler differently and produce different orderings of ties, even when starting from the same seed/state. (See [P] **set sortmethod** for a discussion of `qsort` and `fsort`.) So any reproducibility produced by `set sortrngstate` is specific to the edition of Stata that you are running and which sort method is being used.

See *Sorting with ties* in [D] **sort** for reliable ways of dealing with ties.

## Also see

[P] **creturn** — Return c-class values

[P] **set sortmethod** — Specify a sort method

[D] **sort** — Sort data

[R] **set** — Overview of system parameters