With modern computing power, the differences between sorting algorithms will be infinitesimal and unnoticeable when sorting a small number of input (e.g. a few hundred elements at most), but become apparent when sorting massive lists. Since this is a common case for spreadsheet users, our primary concern should focus on performance when sorting a large number of elements. With this in mind, I strongly suggest LSD Radix sort for our spreadsheet application, as it is far more optimal than selection sort or quicksort in these scenarios (see THEORY).

For a gigantic list of short numbers (such as area codes), a sort that might take LSD Radix sort 1 second to process might take quick sort 4 seconds or take selection sort over 2 *days.* (see PERFORMANCE EXAMPLE)

In addition, LSD Radix sort behaves in a consistent manner users expect. For instance, if the spreadsheet user sorts their employee data first by name (see STRINGS) *then* by supervisor ID, any employees with the same supervisor ID will still be ordered by their individual employee ID. The fastest versions of quicksort do not have this quality (see STABLE)

Perhaps the *only* possible noticeable downside to using LSD Radix sort in our application is if we’re expecting users to frequently sort a small number of *extremely* lengthy inputs, such having entire pages or books worth of text in a single cell. In such an extreme case, LSD radix sort would perform significantly and noticeably slower than quicksort (see INPUT LENGTH). However it is safe to say that such lies well outside the typical user behavior, and as long as the length of each element does not dwarf the number of elements, LSD Radix sort is ideal for spreadsheet sorting.

Therefore, as LSD Radix sort is the fastest sort for the typical use case, and performs in a consistent, predictable way, it should be the algorithm of choice for our spreadsheet program.

(THEORY)

Selection sort is a naïve algorithm that searches the entire list once for each index of the list, resulting in a Θ(n^2) asymptotic runtime.

Quicksort is a divide-and-conquer algorithm that breaks the list into smaller and smaller parts recursively and sorts from the lowest level up. The worst case runtime is O(n^2) but the average runtime is O(nlog(n)). Throughout this paper we will give “the benefit of the doubt” to quicksort and show that even if it is Θ(nlogn), LSD radix sort is still a better option.

LSD Radix sort functions by splitting the elements into “buckets” starting with their Least Significant Digit (hence LSD), then rejoining all the buckets in ascending order. Then it repeats the process for each digit (or character, or bit, depending on implementation). Thus, the runtime of LSD Radix sort is Θ(nw), where n is the number of elements and w is the length of the longest element.

It is reasonable to assume that spreadsheet users will not be inputting numbers with hundreds of digits, or entire pages of text into a single cell, so Θ(nw) will be asymptotically much smaller (and therefore faster) than quicksort’s Θ(nlogn) or selection sort’s Θ(n^2). Thus, LSD Radix sort is the fastest of the 3 in theory.

(PERFORMANCE EXAMPLE)

For a list of 1,000,000 5-digit zip codes, Θ(n^2) selection sort will run in approximately (10,00,000^2 =) 1 trillion units of time, Θ(nlog(n)) quicksort will run in (1,000,000 \* log2(1,000,000) = 1,000,000 \* 19.9 =) 19.9 million units of time, and Θ(nw) LSD Radix sort will run in only (1,000,000 \* 5) = 5 million units of time. Note that these are only *rough estimates* based on the asymptotic runtimes of the algorithms and these calculations are only applicable for comparison. If we arbitrarily set 5 million units = 1 second, then LSD Radix sort would run in 1 second, quicksort in 4 seconds (19.9 million units \* 1s/5million units ≅ 4s), and selection sort would run in (1 trillion units \* 1s/5million units ≅ 200,000s ≅ 2.3 days).

(STRINGS)

We must note that LSD radix sort by default will not properly sort strings, as it will return the following list as sorted: {a, b, c, d, act, cat}. This is because the algorithm only works for *fixed length* strings. However, it is simple to remedy by simply interpreting each string as fixed length equal to the length of the longest element by adding *trailing* spaces where necessary. With this modification, our previous example would now return {a--, act, b--, c--, cat, d--}, a properly alphabetically sorted list (hyphens instead of spaces used for clarity). It is then a simple task to remove the trailing spaces before returning the correct result to the user.

(STABLE)

During each phase of the algorithm, LSD Radix sort preserves the relative order of any elements that have the same digit. In other words, since the algorithm parses the list in order and copies elements to the next available space in the corresponding “bucket”, any two elements going to the same “bucket” stay in the order they were. Since this property is upheld at every step of the process, LSD Radix Sort is a stable sort.

(INPUT LENGTH)

In cases when one or more of the inputs are exceedingly long (i.e. the entire text of a book is in a cell) LSD Radix sort will take much longer than quicksort (since w will be much larger than log(n), nw will be much larger than nlog(n)). However, such cases where quicksort is noticeably faster are so specific and atypical for spreadsheet users that it’s far more economical to focus on the common use cases where LSD Radix sort outperforms quicksort.