A Comprehensive Approach to Algorithmic Machine Sorting of Library of Congress Call Numbers

Scott Wagner and
Corey Wetherington

ABSTRACT

This paper details an approach for accurately machine sorting Library of Congress (LC) call numbers which improves considerably upon other methods reviewed. The authors have employed this sorting method in creating an open-source software tool for library stacks maintenance, possibly the first such application capable of sorting the full range of LC call numbers. The method has potential application to any software environment that stores and retrieves LC call number information.

BACKGROUND

The Library of Congress Classification (LCC) system was devised around the turn of the twentieth century, well before the advent of digital computing.[1] Consequently, neither it nor the system of Library of Congress (LC) call numbers which extend it were designed with any consideration to machine readability or automated sorting.[2] Rather, the classification was formulated for the arrangement of a large quantity of library materials on the basis of content, gathering like items together to allow for browsing within specific topics, and in such a way that a new item may always be inserted (interfiled) between two previously catalogued items without disruption to the overall scheme. Unlike, for instance, modern telephone numbers, ISBNs, or UPCs—identifiers which pair an item with a unique string of digits having a fixed and regular format, largely irrespective of any particular characteristics of the item itself—LC call numbers specify the locations of items relative to others and convey certain encoded information about the content of those items.

The Library of Congress summarizes the essence of the LCC in this way:

> The system divides all knowledge into twenty-one basic classes, each identified by a single letter of the alphabet. Most of these alphabetical classes are further divided into more specific subclasses, identified by two-letter, or occasionally three-letter, combinations. For example, class N, *Art*, has subclasses NA, *Architecture*; NB, *Sculpture*, ND, *Painting*; as well as several other subclasses. Each subclass includes a loosely hierarchical arrangement of the topics pertinent to the subclass, going from the general to the more specific. Individual topics are often broken down by specific places, time periods, or bibliographic forms (such as periodicals, biographies, etc.). Each topic (often referred to as a *caption*) is assigned a single number or a span of numbers. Whole numbers used in LCC may range from one to four digits in length, and may be further extended by the use of decimal numbers. Some subtopics appear in alphabetical, rather than hierarchical, lists and are represented by

Scott Wagner (smw284@psu.edu) is Information Resources and Services Support Specialist, Penn State University Libraries. **Corey Wetherington** (cjw36@psu.edu) is Open and Affordable Course Content Coordinator, Penn State University Libraries.

> decimal numbers that combine a letter of the alphabet with a numeral, e.g., .B72 or .K535. Relationships among topics in LCC are shown not by the numbers that are assigned to them, but by indenting subtopics under the larger topics that they are a part of, much like an outline. In this respect, it is different from more strictly hierarchical classification systems, such as the *Dewey Decimal Classification*, where hierarchical relationships among topics are shown by numbers that can be continuously subdivided.[3]

As this description suggests, LCC cataloging practices can be quite idiosyncratic and inconsistent across different topics and subtopics, and sorting rules for properly shelf-ordering LC call numbers can be correspondingly complex, as we will see below.[4]

For the purposes of discussion in what follows, we divide LC call number strings into three principal substrings: the *classification*, the *Cutter*, and what we will term the *specification*. The classification categorizes the item on the basis of its subject matter, following detailed schedules of the LCC system published by the Library of Congress; the Cutter situates the item alongside others within its classification (often on the basis of its title and/or author[5]); and the specification distinguishes a specific edition, volume, format, or other characteristic of the item from others having the same author and title:

$$\overbrace{\texttt{HC125}}^{a}\overbrace{\texttt{.G25313}}^{b}\ \overbrace{\texttt{1997}}^{c}$$

In the above example, the classification string (a) denotes the subject matter (in this case, General Economic History and Conditions of Latin America), the Cutter string (b) locates the book within this topic on the basis of author and/or title (following a specific encoding process), and the specification string (c) denotes the particular edition of the text (in this case, by year). Each of these general substrings may contain further substrings having specific cataloging functions, and though each is constructed following certain rigid syntactical rules, a great deal of variation in format may be observed within the basic framework. The following is an inexhaustive summary of the basic syntax of each of the three call number components:

- The classification string always begins with one to three letters (the class/subclass), almost always followed by one to four digits (the caption number), possibly including an additional decimal. The classification may also contain a date or ordinal number following the caption number.
- The beginning of the Cutter string is always indicated by a decimal point followed by a letter and at least one digit. While the majority of call numbers contain a Cutter, it is not always present in all cases. Among the sorting challenges posed by LC call numbers, we note in particular the "double Cutter"—a common occurrence in certain subclasses—wherein the Cutter string changes from alphabetic to numeric, then back to alphabetic, and finally again to numeric. Triple Cutters are also possible, as are dates intervening between Cutters. Certain Cutter strings (e.g., in juvenile fiction) end with an alphabetic "work mark" composed of two or more letters.
- The specification string (which may be absent on older materials) is always last, and usually contains the date of the edition, but may also include volume or other numbering, ordinal numbers, format/part descriptions (e.g., "DVD," "manual," "notes"), or other distinguishing information.

Figure 1 shows example call numbers, all found within the catalog of Penn State University Libraries, suggesting the wide variety of possibilities:

`B1190 1951`	*no Cutter string*
`DT423.E26 9th.ed. 2012`	*compound specification*
`E505.5 102nd.F57 1999`	*ordinal number in classification*
`HB3717 1929.E37 2015`	*date in classification*
`KBD.G189s`	*no caption number, no specification*
`N8354.B67 2000x`	*date with suffix*
`PS634.B4 1958-63`	*hyphenated range of dates*
`PS3557.A28R4 1955`	*"double Cutter"*
`PZ8.3.G276Lo 1971`	*Cutter with "work mark"*
`PZ73.S758345255 2011`	*lengthy Cutter decimal*

Figure 1. Example call numbers.

As one might expect given this irregularity in syntax, systematic machine-sorting of LC call numbers is by no means trivial. To begin with, sorting procedures within the LCC system are to a certain degree contextual—that is, the sorter must understand how a given component of a call number operates within the context of the entire string in order to determine how it should sort. Both integer and decimal substrings appear in LC call numbers, so that a numeral may properly precede a letter in one part of a call number (a '1' sorts before an 'A' in the classification portion, for example: `H1` precedes `HA1`), while the contrary may occur in another part (within the Cutter, in particular, an 'A' may well precede a '1': `HB74.P65A2` precedes `HB74.P6512`). Similarly, letters may have different sorting implications depending on where and how they appear. Compare, for instance, the call numbers `V23.K4 1961` and `U1.P32 v.23 1993/94`. The `V` in the former denotes the subclass of general nautical reference works and simply sorts alphabetically, whereas the `v` in the latter call number functions in part as an indicator that the numeral `23` refers to a specific volume number and is to be sorted as an integer rather than a decimal. Such contextual cues are often tacitly understood by a human sorter, but can present considerable challenges when implementing machine sorting procedures. Furthermore, the lack of uniformity or regularity in the format of call number strings poses various practical obstacles for machine sorting. Taken together, these assorted complexities suggest the insufficiency of a single alphanumeric sorting procedure to adequately handle LC call numbers as unprocessed, plain text strings.

LITERATURE REVIEW

A thorough review of information science literature revealed little formal discussion of the algorithmic sorting of LC call numbers. If the topic has been more widely addressed in the scholarly or technical literature, we were unable to discover it. Nevertheless, the general problem appears to be fairly well known. This is evident both from informal online discussions of the topic (e.g., in blog posts, message board threads, and coding forums) and from the existence of certain features of library management system (LMS) and integrated library system (ILS) software designed to address the issue. In this section we examine methods proffered by some of these sources, and detail how each fails to fully account for all aspects of LC call number sorting.

In a brief article archived online, Conley and Nolan outline a method for sorting LC call numbers through the use of function programming in Microsoft Excel.[6] Given a column of plain-text LC call numbers, their approach entails successive processing of the call numbers across several spreadsheet columns with the aim of properly accounting for the sorting of integers. The fully-processed strings are then ultimately ready for sorting in the rightmost column using Excel's built-in sorting functionality. We note that Conley and Nolan's method (hereafter "CNM") only attempts to sort what the authors refer to as the "base call number" (i.e., the classification and Cutter portions), and does not address the sorting of "volume numbers, issue numbers, or sheet numbers" (what we refer to here as the "specification").[7]

CNM stems from the tacit observation that ordinary, single-column sorting of LC call numbers is clearly inadequate in an environment like Excel's. For instance, in the following example, standard character-by-character sorting fails at the third character position, since

`PZ30.A1` erroneously sorts before
`PZ7.A1`

(as `3` is compared to `7` in the third character position), contrary to the correct order (`7` before `30`). To address this, CNM normalizes the numeric portion of the class number with leading zeros so that each numeric string is of equal length, ensuring that the proper digits are compared during sorting. This entails a transformation,

`PZ30.A1` $\Rightarrow$ `PZ0030.A1`
`PZ7.A1` $\Rightarrow$ `PZ0007.A1`

following which the strings will in fact sort correctly in an Excel column. This technique appears adequate until we compare call numbers having subclasses of different length:

`P180.A1` $\Rightarrow$ `P0180.A1`
`PZ30.A1` $\Rightarrow$ `PZ0030.A1`

Here, while standard Excel sorting will in fact properly order the resulting strings, in other applications, depending on the sorting hierarchy employed, sorting may fail in the second position if letters are sorted before numbers. Hierarchy aside, it is not difficult to see the potential issues that may arise from sorting unlike portions of the call number string against one another in this way, particularly when comparing characters within the Cutter string or in situations involving a "double Cutter." For instance, the call numbers

`B945.D4B65 1998` and
`B945.D41 1981b`

are listed here in their proper sorting order, but are in fact sorted in reverse by CNM when, in the eighth character position, `1` is sorted before `B` in accordance with Excel's default sorting priority. This again illustrates an essential problem of character-by-character sorting: in certain substrings we require a letters-before-numbers sorting priority, while in others a numbers-before-letters order is needed. This impasse makes clear that no single-column sorting methodology can succeed for all types of LC call numbers without significant modification to the call number string.

In a blog post, Dannay observed that CNM does not account for certain call number formats, particularly those of legal materials within the `K` classification having 3-letter class strings.[8] (The same would also be true in the `D` classification, where 3-letter strings also appear.) Although minor modification of portions of the function code (e.g., replacing certain '2's with '3's) would be sufficient to alleviate this particular issue, Dannay proposed instead to employ placeholder characters to normalize the classification string and avoid instances of alphabetic characters being compared against numeric ones. Dannay's Method (DM) normalizes various parts of the classification string, including the subclass, caption, and decimal portions:

```
Q171.T78  ⇒   Q**0171.0.T78
QA9.R8    ⇒   QA*0009.0.R8
```

(Here, of course, it is imperative that the chosen placeholder character sort before all letters in the sorting hierarchy.) DM thus successfully avoids the issue of comparing classification strings of unequal length or format.

Nevertheless, despite the improvements of DM over CNM, both approaches are ultimately unable to properly process certain types of common LC call numbers. For example, call numbers with dates preceding the Cutter (e.g., `GV722 1936.H55 2006`) and call numbers without Cutters (e.g., `B1205 1958`) both result in errors, as do those containing the aforementioned "double Cutters." Furthermore, as we previously noted, neither DM nor CNM were designed to handle any portion of the specification string following the Cutter, where the presence of ordinal and volume-type numbering is commonplace. Hence neither method is able to properly order the quite ordinary pair of call numbers `AC1.G7 v.19` and `AC1.G7 v.2`, since the first digit of each's volume number is compared and ordered numerically (i.e., character-by-character), resulting in a mis-sort. Though neither DN nor CNM is ultimately comprehensive (nor designed to be), both methods contain valuable insights and strategies that inform our own approach to the problem.

SOFTWARE REVIEW

Available software solutions for sorting LC call numbers appear to be nearly as scant as literature on the subject. While GitHub contains a handful of programs that attempt to address the problem, we found none which could be considered comprehensive. Table 1 is a summary of those programs we discovered and were able to examine.

The "sqlite3-lccn-extension" program is an extension for SQLite 3 which provides a collation for normalizing LC call numbers, executing from a SQLite client shell. We discovered several limitations in its ability to sort certain call number formats similar to those discussed above in the literature review. For instance, the program cannot correctly sort specification integers (e.g., it sorts `v.13` before `v.3`) or call numbers lacking Cutter strings (e.g., it sorts `B 1190.A1 1951` before `B 1190 1951`). We found similar issues with "js-loc-callnumbers," a JavaScript program with a web interface into which a list of call numbers can be pasted. The program transforms the call numbers into normalized strings, which are then sorted and displayed to the user. However, we observed that it does not account for dates or ordinal numbers in the classification string, nor can it correctly sort call numbers lacking caption numbers.[9]

Program and Author	App-Type, Interface	Repository URL	Last Update
"sqlite3-lccn-extension" by Brad Dewar	database extension, shell	https://github.com/macdewar/sqlite3-lccn-extension	Dec. 2013
"js-loc-callnumbers" by Ray Voelker	JavaScript, web	https://github.com/rayvoelker/js-loc-callnumbers	Feb. 2017
"Library-of-Congress-System" by Luis Ulloa	Python tutorial, command line	https://github.com/ulloaluis/Library-of-Congress-System	Sep. 2018
"lcsortable" by mbelvadi2	Google Sheets script	https://github.com/mbelvadi2/lcsortable	May 2017
"library-callnumber-lc" by Library Hackers	Perl, Python	https://github.com/libraryhackers/library-callnumber-lc/tree/master/perl/Library-CallNumber-LC	Dec. 2014
"lc_call_number_compare" by SMU Libraries	JavaScript, command line	https://github.com/smu-libraries/lc_call_number_compare	Dec. 2016
"lc_callnumber" by Bill Dueber	Ruby	https://github.com/billdueber/lc_callnumber	Feb. 2015

Table 1. List of GitHub software involving LC call number sorting.

Several of the programs are rather narrow in scope. The "lcsortable" script is a Google Sheets scheme for normalizing LC call numbers into a separate column for sorting, very much like CNM and DM. Its normalization routine appears to conflate decimals and integers, though, leading to transformations such as

$$\texttt{HF5438.5.P475 2001} \quad \Rightarrow \quad \texttt{HF5438.0005.P04752001}$$

which would clearly result in a great deal of incorrect sorting across a wide array of LC call number formats. The command-line-based Python program "library-callnumber-lc" processes a call number and returns a normalized sort key, but is not intended to store or sort groups of call numbers. It cannot adequately handle compound specifications or Cutters containing consecutive letters (e.g., `S100.BC123 1985`), and does not appear to preserve the demarcation between a caption integer and caption decimal (i.e., the decimal point), thereby commingling integer and decimal sorting logic. Lastly, "Library-of-Congress-System" is a tutorial/training program written in Python that runs from the command line and supplies a list of call numbers for the user to sort. It does not draw call numbers from a static collection nor allow call numbers to be input by the user; rather, it randomly generates call numbers within certain parameters and following a prescribed pattern. As such, we were not able to satisfactorily test its sorting capabilities for the kind of use-case scenario under discussion.

We did not evaluate the remaining two GitHub programs, "lc_call_number_compare" and "lc_callnumber," as we could not get the former, a JavaScript ES6 module, to execute, and as the latter, a Ruby application which we did not attempt to install, evidently remains unfinished: its GitHub documentation lists "Normalization: create a string that can be compared with other normalized strings to correctly order the call numbers" as the among tasks yet to be completed.

In addition to these open resources, we examined LC sorting capability within the commercial LMS/ILS software we had at hand. The MARC (Machine-Readable Cataloging) 21 protocol, a widely used international standard for formatting bibliographic data, provides a specific syntax for cataloging LC call numbers for the purposes of machine parsing.[10] Symphony WorkFlows, the LMS licensed by Penn State University Libraries from SirsiDynix (and thus the only one available for our direct examination), contains within its search module a call number browsing feature which attempts to sort call numbers in shelving order via "Shelving IDs," call number strings rendered from each item's MARC 21 "050" data field for sorting purposes. While these Shelving IDs are not visible within WorkFlows (that is, they operate in the background), they can be accessed as plain text strings via BLUEcloud Analytics, a separate, SirsiDynix-branded data assessment and reporting tool peripheral to the LMS. Examination of these sort keys revealed integer normalization strategies similar to those of DM and CNM, with additional processing of volume-type numbering within the specification string. However, these Shelving IDs are similarly unable to correctly sort "double Cutter" substrings and other syntactic complexities, such as ordinal numbers appearing in the classification. The following Shelving ID transformations of two call numbers in the Penn State University Libraries catalog, for instance, fail to properly account for the ordinal numbers which appear within the classification:

```
E507.5 36th.V47 2003   ⇒   E 000507.5 36TH.V47 2003
E507.5 5th.C36 2000    ⇒   E 000507.5 5TH.C36 2000
```

Consequently, and as expected, these two call numbers sort incorrectly within WorkFlows' call number browsing panes.[11]

PROPOSED PARSING AND SORTING METHODOLOGY

Given the sorting difficulties inherent in the single-column approaches outlined above, we suggest a multi-column, tiered sorting procedure in which only like portions of the call number are compared to one another. This requires the call number to be processed, its various components identified, and each component appropriately sorted according to its specific type. This, in turn, requires a sorting algorithm which can identify like substrings by scanning for specific patterns and cues.

"Shelf reading" is a term for the common practice of verifying the correct ordering of items filed on a library shelf, typically unaided by technology, and our approach is primarily informed by the kind of mental procedures one undertakes when performing such sorting "in one's head."[12] Perhaps the most significant component of this process involves recognizing and interpreting the role and logic of specific types of substrings and identifying their positions within the sorting hierarchy. The overall design of the LC classification, from class to subclass to caption, constitutes a left-to-right progression from general to specific, and the classification portion of a call number can be interpreted as a series of containers holding items of increasingly narrow scope, some of which may be empty (that is, absent). This creates a structure that has a linear, hierarchical aspect, but also contains within it subcategories that share a common position within the structure. The priority that a subcategory (or container) is afforded in the sorting process depends first upon its position in the linear hierarchy, and subsequently on the depth ascribed to it relative to other subcategories that share the same position. Call numbers indicate a subcategory's position in the linear dimension by including or expanding sections; its depth within a given position is encoded in the character or series of characters chosen to represent it. Thus, the sorting process may be regarded as a comparison of the paths that two call numbers denote through this structure, and the point at which the paths diverge is then the decisive point in determining an item's position relative to others. This inflection point may occur at any juncture of the comparison, from the first character to the last.

Given these observations, a comprehensive machine-sorting strategy must observe the following provisions:

1. Characters in call numbers should only be compared to characters that occupy an equivalent section of another call number. ("Like compared to like.")
2. Within these designated sections, characters should only be compared to characters that occupy a corresponding position (place value) within that section.
3. If call numbers are identical up to the point that one of them lacks a section that the other call number possesses, the one with the "missing" section is ordered first. This is in keeping with the convention that items occupying a more general level in the hierarchy are ordered before those occupying a more specific one. (This principle is often summarized in shelf-reading tutorials as "nothing before something.")
4. If call numbers are identical up to the point that one of them lacks a character in a given position within a particular section that the other call number possesses, the one missing the character is ordered first. Again, this preserves the general to specific scheme of LCC sorting. (Another instance of "nothing before something.")
5. Whole numbers (e.g., caption integers, volume numbers) must be distinguished from decimals. For character-by-character sorting to work in sections of the call number containing integers, the length of whole numbers must be normalized to assure each digit is compared to another of equal place value.

APPLICATION OF METHODOLOGY

ShelfReader is a software application designed by the authors to improve the speed and accuracy of the shelf-reading process in collections filed using the Library of Congress system—and, to our knowledge, is the first such application to do so. It was coded by Scott Wagner in PHP and JavaScript, uses MySQL for data storage and sorting, and is deployed as a web application.

ShelfReader allows the user to scan library items in the order they are shelved and receive feedback regarding any mis-shelved items. The program receives an item's unique barcode identification via a barcode scanner, assembles a REST request incorporating the barcode, and sends it to an API connected to the LMS. The application then processes the response, retrieving the title and call number of the item, along with information about the item's status (for example, if it has been marked as lost or missing). The call number is passed off to the sorting algorithm, which processes it and assigns it a position among the set of call numbers recorded during that session. A user interface then presents a "virtual shelf" to the user displaying a graphical representation of the items in the order they were scanned. When items are out of place on the shelf, the program calculates the fewest number of moves needed to correct the shelf and presents the necessary corrections for the user to perform until the shelf is properly ordered. A screenshot depicting the ShelfReader GUI during a typical shelf-reading session is presented in figure 2.

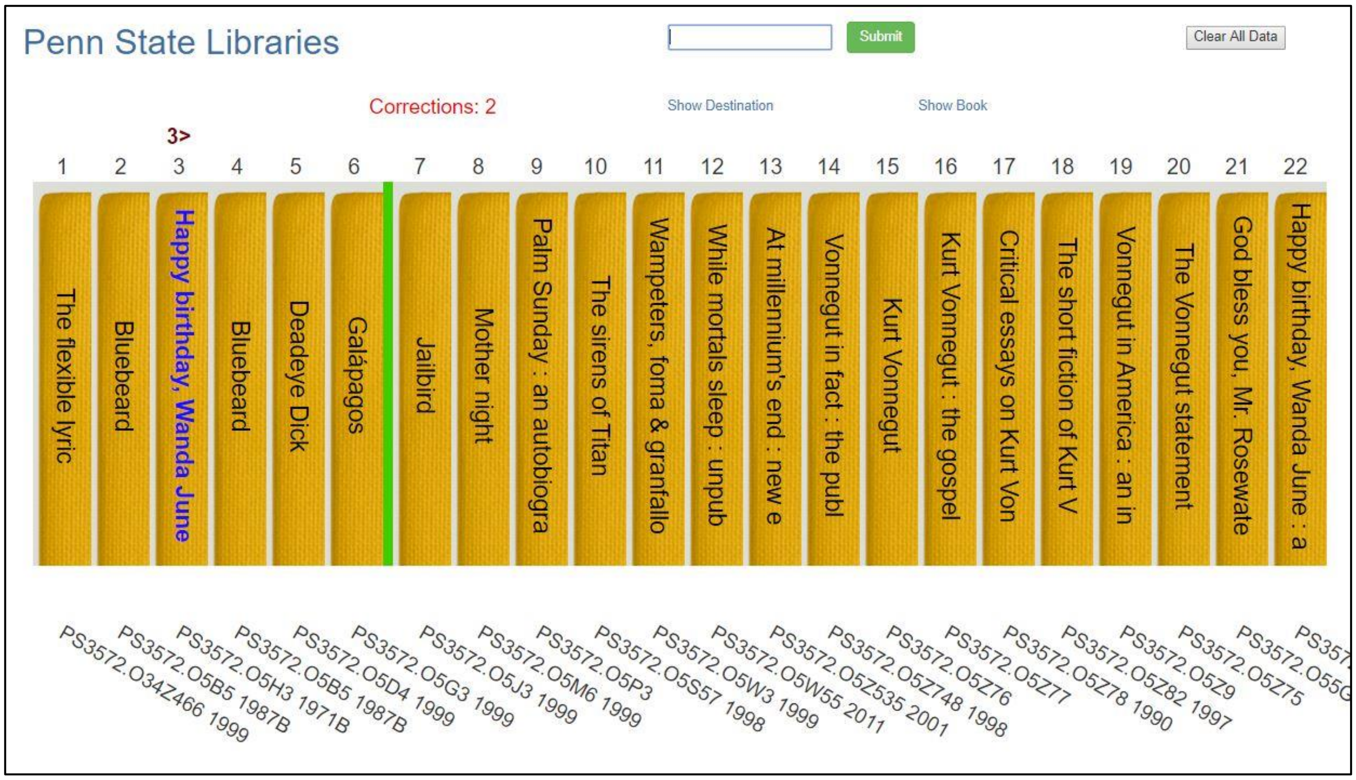

Figure 2. A screenshot of the ShelfReader GUI, showing an incorrectly filed item (highlighted in blue text) and its proper filing position (represented by the green band).

ShelfReader's sorting strategy consists of breaking call numbers into elemental substrings and arranging those parts in a database table so that any two call numbers may be compared exclusively on the basis of their corresponding parts. To this end, a base set of call number components was established. These are shown in table 2, along with their abbreviations (for ease in reference), maximum length, and corresponding MySQL data types.

The specific MySQL data type determines the kind of sorting employed in each column:

- *varchar* Accepts alphanumeric string data. Sorting is character by character, numbers before letters.
- *integer* Accepts numerical data; numbers are evaluated as whole numbers.
- *decimal* Accepts decimal values. Specifying the overall length of the column and the number of characters to the right of the decimal point has the effect of adding zeros as placeholders in any empty spaces to the right of the last digit. The values are then compared digit by digit.

- *timestamp* A date/time value that defaults to the date and time the entry is made. This orders call numbers that are identical (i.e., multiple copies of the same item) in the order they are scanned.

Section, Component	Abbreviation	Max. Length	MySQL Data Type
Classification			
class/subclass	sbc	3	varchar
caption number, integer part	ci	4	integer
caption number, decimal part	cdl	16	decimal
caption date	cdt	4	varchar
caption ordinal	co	16	integer
caption ordinal indicator	coi	2	varchar
Cutter			
first Cutter, alphabetical part	c1a	3	varchar
first Cutter, numerical part	c1n	16	decimal
first Cutter date	cd	4	integer
second Cutter, alphabetical part	c2a	3	varchar
second Cutter, numerical part	c2n	16	decimal
second Cutter date	cd2	4	integer
third Cutter, alphabetical part	c3a	3	varchar
third Cutter, numerical part	c3n	16	decimal
Specification			
specification	sp	256	varchar
timestamp	—	—	MySQL timestamp

Table 2. ShelfReader call number components and data types.

When parsing a call number, it must be assumed that each call number may contain all of the components identified above. The following is a general outline of the parsing algorithm which processes the call number:

1. An array is created from the call number. Each character, including spaces, is an element of the array.
2. A second array is then created to serve as a template for each call number, replacing the actual characters with ones indicating data type. For example, all integers are replaced with 'I's. This makes pattern matching and data-type testing simpler.
3. Pattern matching is used to identify the presence or absence of landmarks such Cutters, spaces, volume-type numbering, etc.
4. When landmarks are identified, their beginning and ending positions in the call number string are noted.
5. Component strings are created by looping through the appropriate section of the call number template, constructing a string in which the template characters are replaced by the actual characters in the call number string and continuing until a space, the end of the string, or an incompatible character is encountered.
6. Where needed, whole numbers strings are normalized to uniform length.

Dividing a call number into its component parts and placing those parts in separate columns in a database table, then, effectively creates a sort key that may be used for ordering. This key occupies a row of the table, and is an inflated representation of the call number insofar as it makes use of the maximum possible string length of each component type. It contains the characters of each component the call number possesses, and any empty columns serve as placeholders for components it does not possess. When two call numbers are compared, sorting proceeds through each successive column, each component (and each character within each component) serving as a potential break point within the sorting process.

We note that every column (with the exception of the specification) contains exclusively alphabetic or numeric data, so that numbers and letters are never compared in those sections of the call number string. (The use of spaces in the specification string effectively accounts for the mixed alphanumeric data type.)

Some additional points of clarification regarding the algorithm's multi-column approach to sorting are worth mentioning:

1. Any lowercase alphabetic characters are converted to uppercase before processing in order to ensure that letter case does not affect sorting.
2. Components are arranged in the database table from left to right in the order they occur in the call number.
3. If a call number does not contain a given component, the column is left empty (in the case of a varchar column) or is assigned a zero value (in the case of numeric columns).
4. Empty columns and zero columns sort before columns containing data.
5. In columns designated as varchar columns, numbers are compared as whole numbers. This means that, in order to sort correctly, the length of any number stored must be normalized to a uniform length (6 places) by adding leading zeros. For example, 17 must be normalized to "000017."
6. Sorting proceeds column by column, provided the call numbers are identical. When the first difference is encountered, sorting is complete.

Table 3 shows two randomly selected call numbers of rather common configuration, along with the corresponding sort keys created by ShelfReader:

$$\left.\begin{array}{l}\texttt{E169.1.B634 2002}\\ \texttt{E169.1.B653 1987}\end{array}\right\}\Rightarrow$$

sbc	ci	cdl	cdt	co	coi	c1a	c1n	cd	c2a	c2n	cd2	c3a	c3n	sp
E	0169	0.10000		0		B	0.6340000000							0002002
E	0169	0.10000		0		B	0.6530000000							0001987

Table 3. Example ShelfReader sort-key processing of two similar call numbers.

In this first example, sorting is complete when `3` is compared to `5` in the first numerical Cutter (c1n) column. (Note that we have here truncated the length of certain strings for space and readability.)

To illustrate how the application handles call numbers having heterogenous formats, table 4 shows the sort keys created from two call numbers in an example mentioned above, one with a "double Cutter" and one without:

$$\left.\begin{array}{l}\texttt{B945.D4B65 1998}\\ \texttt{B945.D41 1981b}\end{array}\right\}\Rightarrow$$

sbc	ci	cdl	cdt	co	coi	c1a	c1n	cd	c2a	c2n	cd2	c3a	c3n	sp
B	0945	0.0		0		D	0.400000		B	0.650000				0001998
B	0945	0.0		0		D	0.410000			0.000000				0001981B

Table 4. ShelfReader sort-key processing of a "double Cutter" call number and a nearby, single Cutter call number.

By pushing the second cutter (`B65`) in the first call number into the c2a and c2n columns, the issue of comparing incompatible sections of the call number is avoided, as the `1` in the second call number is compared to the placeholder `0` in the first. When the sorting routine reaches this position, it terminates, and any subsequent characters are ignored.

Aspects of this multi-column approach may seem counterintuitive at first, but the method mimics what we do when we order call numbers mentally. One compares two call numbers character by character within these component categories until encountering a difference, or until a character or entire category in one of the call numbers is found to be absent.

RESULTS

ShelfReader's sorting method is powerful, accurate, and has been extensively tested without issue in a number of different academic libraries within Penn State's statewide system. The application accurately sorts all valid LC call numbers (with the exception of those for certain cartographic materials in the `G1000` – `G9999` range, which sometimes employ a different syntax and sorting order) as well those of the National Library of Medicine classification system (which augments

LCC with class `W` and subclasses `QS` – `QZ`) and the National Library of Canada classification (which adds to LCC the subclass `FC`, for Canadian history). While there may conceivably be valid LC or LC-extended call numbers having exotic formats that would fail to correctly sort in ShelfReader, we are not aware of any examples (outside of, once again, the `G1000` – `G9999` range), nor have we received reports of any from users.

In addition to verifying proper shelf-ordering, ShelfReader contains a number of other features useful for stacks maintenance. The program can identify shelved items that are still checked out to patrons, have been marked missing or lost, or are flagged as in transit between locations, and often reveals items which have been inadvertently "shadowed" (i.e., excluded from public-facing library catalogs) or have shelf labels which do not match their catalogued call numbers. The GUI has different modes to accommodate the user's preferred view (both single shelf and multi-shelf, stacks views), and allows for a good deal of flexibility in how and when the user wishes to make and record shelf corrections. A reports module is also included, which tracks shelving statistics and other useful information for later reference.

The ShelfReader application code (including the full sorting algorithm) is freely available via an MIT license at https://github.com/scodepress/shelfreader. While ShelfReader was developed and tested using the collections and systems of Penn State University Libraries, its architecture could be adapted and configured for use with other library APIs and adjusted to suit local practices within the general confines of the LC call number structure.[13] We can also envision a wide array of potential applications of the sorting functionality within other software environments, and we welcome and encourage users to pursue innovative adaptations of the method.

REFERENCES AND NOTES:

[1] Leo E. LaMontagne, *American Library Classification: With Special Reference to the Library of Congress* (Hamden, CT: The Shoe String Press, 1961). The lengthy development of the LCC is described in detail in chapters XIII and XIV (pp. 221-51).

[2] Indeed, as LaMontagne asserts, "The Classification was constructed [. . .] to provide for the needs of the Library of Congress, with no thought to its possible adoption by other libraries. In fact, the Library has never recommended that other libraries adopt its system . . . " (*ibid*., p. 252). Nevertheless, LCC is employed by the overwhelming majority of academic libraries in the United States (Brady Lund and Daniel Agbaji, "Use of Dewey Decimal Classification by Academic Libraries in the United States," *Cataloging & Classification Quarterly* 56, no. 7 (December 2018): 653-61, https://doi.org/10.1080/01639374.2018.1517851).

[3] "Library of Congress Classification," Library of Congress, https://www.loc.gov/catdir/cpso/lcc.html. Italics in original.

[4] For a summary of LC sorting rules, see "How to Arrange Books in Call Number Order Using the Library of Congress System," Rutgers University Libraries, https://www.libraries.rutgers.edu/rul/staff/access_serv/student_coord/LibConSys.pdf. Note that this summary is not comprehensive and does not cover all contingencies.

[5] Here we emphasize that our definition of the Cutter string may differ from that of others, including (at times) that of the Library of Congress. For instance, the schedules for certain LCC

subclasses regard the first portion of a Cutter as part of the classification itself. Since this paper concerns sorting rather than classification, we favor the simpler and more convenient definition.

[6] J.F. Conley and L.A. Nolan, "Call Number Sorting in Excel," https://scholarsphere.psu.edu/downloads/9cn69m421z.

[7] Conley and Nolan, "Call Number Sorting in Excel."

[8] Tim Danny, "Sorting LC Call Numbers in Excel," https://medium.com/@tdannay/sorting-lc-call-numbers-in-excel-75de044bbb04.

[9] While there is in fact a "hack" or partial patch built into the program which identifies call numbers beginning with the subclass KBG and parses them separately, there is no general support for other call numbers in this category.

[10] For the details of this syntax, see "050 - Library of Congress Call Number (R)," Library of Congress, https://www.loc.gov/marc/bibliographic/bd050.html.

[11] Testing was conducted on SirsiDynix Symphony WorkFlows Staff Client version 3.5.2.1079, build date June 5, 2017.

[12] For an overview, see "Student Library Assistant Training Guide: Shelving Basics," Florida State College at Jacksonville, https://guides.fscj.edu/training/shelving.

[13] Shelfreader was written to receive real-time data directly from a SirsiDynix API connected to Penn State University Libraries' LMS, a great improvement over drawing from a static collections database. This does, however, present a challenge for making the program easily adaptable to libraries using distinct web services. A strategy to adapt the program would need to account for potential differences in barcode structure, structure and naming conventions in the REST request, and structure and naming conventions within the server response from institution to institution. It is possible that these issues could be resolved via a configuration file made available to the user, but no attempt to address this issue has been undertaken as of yet.