Technical Reports

L2/26-190

Proposed Update Unicode® Technical Standard #10

UNICODE COLLATION ALGORITHM

Version	18.0.0 (draft 3)
Editors	Ken Whistler, Markus Scherer
Date	2026-05-18
This Version	https://www.unicode.org/reports/tr10/tr10-54.html
Previous Version	https://www.unicode.org/reports/tr10/tr10-53.html
Latest Version	https://www.unicode.org/reports/tr10/
Latest Proposed Update	https://www.unicode.org/reports/tr10/proposed.html
Revision	54

Summary

This report is the specification of the Unicode Collation Algorithm (UCA), which details how to compare two Unicode strings while remaining conformant to the requirements of the Unicode Standard. The UCA also supplies the Default Unicode Collation Element Table (DUCET) as the data specifying the default collation order for all Unicode characters.

Status

*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time. Publication does not imply endorsement by the Unicode Consortium. This is not a stable document; it is inappropriate to cite this document as other than a work in progress.*

> ***A Unicode Technical Standard (UTS)** is an independent specification. Conformance to the Unicode Standard does not imply conformance to any UTS.*

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this document is found in the References. For the latest version of the Unicode Standard, see [Unicode]. For a list of current Unicode Technical Reports, see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Contents

1 Introduction
- 1.1 Multi-Level Comparison
  - 1.1.1 Collation Order and Code Chart Order
- 1.2 Canonical Equivalence
- 1.3 Contextual Sensitivity
- 1.4 Customization
- 1.5 Other Applications of Collation
- 1.6 Merging Sort Keys
- 1.7 Performance
- 1.8 What Collation is Not
- 1.9 The Unicode Collation Algorithm
  - 1.9.1 Goals

1.9.2 Non-Goals
2 Conformance
2.1 Basic Conformance Requirements
2.2 Additional Conformance Requirements
3 Definitions and Notation
3.1 Collation Weights, Elements, and Levels
3.2 Ignorables
3.3 Mappings
3.3.1 Simple Mappings
3.3.2 Expansions
3.3.3 Contractions
3.3.4 Many-to-Many Mappings
3.4 Collation Element Tables
3.5 Input Matching
3.6 Sortkeys
3.7 Comparison
3.7.1 Equality
3.7.2 Inequality
3.7.3 Notation for Collation Element Comparison
3.7.4 Notation for String Comparison
3.8 Parametric Settings
3.8.1 Backward Accents
4 Variable Weighting
4.1 Examples of Variable Weighting
4.2 Interleaving
5 Well-Formedness of Collation Element Tables
6 Default Unicode Collation Element Table
6.1 Contractions in DUCET
6.1.1 Rearrangement and Contractions
6.1.2 Omission of Generic Contractions
6.2 Weighting Considerations in DUCET
6.3 Overall Order of DUCET
6.4 Exceptional Grouping in DUCET
6.5 Tailoring of DUCET
6.6 Default Values in DUCET
6.7 Tibetan and Well-Formedness of DUCET
6.8 Stability of DUCET
7 Main Algorithm
7.1 Normalize Each String
7.2 Produce Collation Element Arrays
7.3 Form Sort Keys
7.4 Compare Sort Keys
7.5 Rationale for Well-Formed Collation Element Tables
8 Tailoring
8.1 Parametric Tailoring
8.2 Tailoring Example
8.3 Use of Combining Grapheme Joiner
8.4 Preprocessing
9 Implementation Notes
9.1 Reducing Sort Key Lengths
9.1.1 Eliminating Level Separators
9.1.2 L2/L3 in 8 Bits
9.1.3 Machine Words
9.1.4 Run-Length Compression
9.2 Large Weight Values
9.3 Reducing Table Sizes
9.3.1 Contiguous Weight Ranges
9.3.2 Leveraging Unicode Tables
9.3.3 Reducing the Repertoire

9.3.4 Memory Table Size
9.4 Avoiding Zero Bytes
9.5 Avoiding Normalization
9.6 Case Comparisons
9.7 Incremental Comparison
9.8 Catching Mismatches
9.9 Handling Collation Graphemes
9.10 Sorting Plain Text Data Files
10 Weight Derivation
10.1 Derived Collation Elements
10.1.1 Handling Ill-Formed Code Unit Sequences
10.1.2 Unassigned and Other Code Points
10.1.3 Implicit Weights
10.1.4 Trailing Weights
10.1.5 Reserved Weights
10.1.6 Hangul Collation
10.2 Tertiary Weight Table
11 Searching and Matching
11.1 Collation Folding
11.2 Asymmetric Search
11.2.1 Returning Results
12 Data Files
12.1 Allkeys File Format
12.2 Conformance Tests
Appendix A: Deterministic Sorting
A.1 Stable Sort
A.1.1 Forcing a Stable Sort
A.2 Deterministic Sort
A.3 Deterministic Comparison
A.3.1 Avoid Deterministic Comparisons
A.3.2 Forcing Deterministic Comparisons
A.4 Stable and Portable Comparison
Appendix B: Synchronization with ISO/IEC 14651
Acknowledgements
References
Migration Issues
Modifications

---

1 Introduction

Collation is the general term for the process and function of determining the sorting order of strings of characters. It is a key function in computer systems; whenever a list of strings is presented to users, they are likely to want it in a sorted order so that they can easily and reliably find individual strings. Thus it is widely used in user interfaces. It is also crucial for databases, both in sorting records and in selecting sets of records with fields within given bounds.

Collation varies according to language and culture: Germans, French and Swedes sort the same characters differently. It may also vary by specific application: even within the same language, dictionaries may sort differently than phonebooks or book indices. For non-alphabetic scripts such as East Asian ideographs, collation can be either phonetic or based on the appearance of the character. Collation can also be customized according to user preference, such as ignoring punctuation or not, putting uppercase before lowercase (or vice versa), and so on. Linguistically correct *searching* needs to use the same mechanisms: just as "ä" and "æ" sort as if they were the same base letter in Swedish, a loose search should pick up words with either one of them.

Collation implementations must deal with the complex linguistic conventions for ordering text in specific languages, and provide for common customizations based on user preferences. Furthermore,

algorithms that allow for good performance are crucial for any collation mechanisms to be accepted in the marketplace.

Table 1 shows some examples of cases where sort order differs by language, usage, or another customization.

Table 1. Example Differences

Language	Swedish:	z < ö
	German:	ö < z
Usage	German Dictionary:	of < öf
	German Phonebook:	öf < of
Customizations	Upper-First	A < a
	Lower-First	a < A

Languages vary regarding which types of comparisons to use (and in which order they are to be applied), and in what constitutes a fundamental element for sorting. For example, Swedish treats *ä* as an individual letter, sorting it after *z* in the alphabet; German, however, sorts it either like *ae* or like other accented forms of *a*, thus following *a*. In Slovak, the digraph *ch* sorts as if it were a separate letter after *h*. Examples from other languages and scripts abound. Languages whose writing systems use uppercase and lowercase typically ignore the differences in case, unless there are no other differences in the text.

It is important to ensure that collation meets user expectations as fully as possible. For example, in the majority of Latin languages, ø sorts as an accented variant of o, meaning that most users would expect ø alongside o. However, a few languages, such as Norwegian and Danish, sort ø as a unique element after z. Sorting "Søren" after "Sylt" in a long list, as would be expected in Norwegian or Danish, will cause problems if the user expects ø as a variant of o. A user will look for "Søren" between "Sorem" and "Soret", not see it in the selection, and assume the string is missing, confused because it was sorted in a completely different location. In matching, the same can occur, which can cause significant problems for software customers; for example, in a database selection the user may not realize what records are missing. See *Section 1.5, Other Applications of Collation*.

With Unicode applications widely deployed, multilingual data is the rule, not the exception. Furthermore, it is increasingly common to see users with many different sorting expectations accessing the data. For example, a French company with customers all over Europe will include names from many different languages. If a Swedish employee at this French company accesses the data from a Swedish company location, the customer names need to show up in the order that meets this employee's expectations—that is, in a Swedish order—even though there will be many different accented characters that do not normally appear in Swedish text.

For scripts and characters not used in a particular language, explicit rules may not exist. For example, Swedish and French have clearly specified, distinct rules for sorting ä (either after z or as an accented character with a secondary difference from a), but neither defines the ordering of characters such as Ж, ʊ, ♫, ∞, ◊, or ⌂.

1.1 Multi-Level Comparison

To address the complexities of language-sensitive sorting, a *multilevel* comparison algorithm is employed. In comparing two words, the most important feature is the identity of the base letters—for example, the difference between an *A* and a *B*. Accent differences are typically ignored, if the base letters differ. Case differences (uppercase versus lowercase) are typically ignored, if the base letters or their accents differ. Treatment of punctuation varies. In some situations a punctuation character is treated like a base letter. In other situations, it should be ignored if there are any base, accent, or case differences. There may also be a final, tie-breaking level (called an *identical* level), whereby if there are no other differences at all in the string, the (normalized) code point order is used.

Table 2. Comparison Levels

Level	Description	Examples
L1	Base characters	role < roles < rule
L2	Accents	role < rôle < roles
L3	Case/Variants	role < Role < rôle
L4	Punctuation	role < “role” < Role
Ln	Identical	role < ro□le < “role”

The examples in *Table 2* are in English; the description of the levels may correspond to different writing system features in other languages. In each example, for levels L2 through Ln, the differences on that level (indicated by the underlined characters) are swamped by the stronger-level differences (indicated by the blue text). For example, the L2 example shows that difference between an o and an accented ô is swamped by an L1 difference (the presence or absence of an *s*). In the last example, the □ represents a format character, which is otherwise completely ignorable.

The primary level (L1) is for the basic sorting of the text, and the non-primary levels (L2..Ln) are for adjusting string weights for other linguistic elements in the writing system that are important to users in ordering, but less important than the order of the basic sorting. In practice, fewer levels may be needed, depending on user preferences or customizations.

1.1.1 Collation Order and Code Chart Order

Many people expect the characters in their language to be in the "correct" order in the Unicode code charts. Because collation varies by language and not just by script, it is not possible to arrange the encoding for characters so that simple binary string comparison produces the desired collation order for all languages. Because multi-level sorting is a requirement, it is not even possible to arrange the encoding for characters so that simple binary string comparison produces the desired collation order for any particular language. Separate data tables are required for correct sorting order. For more information on tailorings for different languages, see [CLDR].

The basic principle to remember is: ***The position of characters in the Unicode code charts does not specify their sort order.***

1.2 Canonical Equivalence

There are many cases in Unicode where two sequences of characters are canonically equivalent: the sequences represent essentially the same text, but with different actual sequences. For more information, see [UAX15].

Sequences that are canonically equivalent must sort the same. *Table 3* gives some examples of canonically equivalent sequences. For example, the *angstrom sign* was encoded for compatibility, and is canonically equivalent to an *A-ring*. The latter is also equivalent to the decomposed sequence of *A* plus the *combining ring* character. The order of certain combining marks is also irrelevant in many cases, so such sequences must also be sorted the same, as shown in the second example. The third example shows a composed character that can be decomposed in four different ways, all of which are canonically equivalent.

Table 3. Canonical Equivalence

1	Å	U+212B ANGSTROM SIGN
	Å	U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
	A ◌̊	U+0041 LATIN CAPITAL LETTER A, U+030A COMBINING RING ABOVE

2	x ◌̛◌̣	U+0078 LATIN SMALL LETTER X, U+031B COMBINING HORN, U+0323 COMBINING DOT BELOW
	x ◌̣ ◌̛	U+0078 LATIN SMALL LETTER X, U+0323 COMBINING DOT BELOW, U+031B COMBINING HORN
3	ự	U+1EF1 LATIN SMALL LETTER U WITH HORN AND DOT BELOW
	ụ ◌̛	U+1EE5 LATIN SMALL LETTER U WITH DOT BELOW, U+031B COMBINING HORN
	u ◌̛◌̣	U+0075 LATIN SMALL LETTER U, U+031B COMBINING HORN, U+0323 COMBINING DOT BELOW
	ư ◌̣	U+01B0 LATIN SMALL LETTER U WITH HORN, U+0323 COMBINING DOT BELOW
	u ◌̣ ◌̛	U+0075 LATIN SMALL LETTER U, U+0323 COMBINING DOT BELOW, U+031B COMBINING HORN

1.3 Contextual Sensitivity

There are additional complications in certain languages, where the comparison is context sensitive and depends on more than just single characters compared directly against one another, as shown in *Table 4*.

The first example of such a complication consists of ***contractions***, where two (or more) characters sort as if they were a single base letter. In the table below, *CH* acts like a single letter sorted after *C*.

The second example consists of ***expansions***, where a single character sorts as if it were a sequence of two (or more) characters. In the table below, an *Œ* ligature sorts as if it were the sequence of *O* + *E*.

Both contractions and expansions can be combined: that is, two (or more) characters may sort as if they were a different sequence of two (or more) characters. In the third example, for Japanese, a length mark sorts with only a tertiary difference from the vowel of the previous syllable: as an *A* after *KA* and as an *I* after *KI*.

Table 4. Context Sensitivity

Contractions	H < Z, *but* CH > CZ
Expansions	OE < Œ < OF
Both	カー < カア, *but* キー > キア

Some languages have additional oddities in the way they sort. Normally, all differences in sorting are assessed from the start to the end of the string. If all of the base letters are the same, the first accent difference determines the final order. In row 1 of *Table 5*, the first accent difference is on the *o*, so that is what determines the order. In some French dictionary ordering traditions, however, it is the *last* accent difference that determines the order, as shown in row 2.

Table 5. Backward Accent Ordering

Normal Accent Ordering	cote < coté < côte < côté
Backward Accent Ordering	cote < côte < coté < côté

1.4 Customization

In practice, there are additional features of collation that users need to control. These are expressed in user-interfaces and eventually in APIs. Other customizations or user preferences include the following:

- *Language.* This is the most important feature, because it is crucial that the collation match the expectations of users of the target language community.
- *Strength.* This refers to the number of levels that are to be considered in comparison, and is another important feature. Most of the time a three-level strength is needed for comparison of strings. In some cases, a larger number of levels will be needed, while in others—especially in searching—fewer levels will be desired.
- *Case Ordering.* Some dictionaries and authors collate uppercase before lowercase while others use the reverse, so that preference needs to be customizable. Sometimes the case ordering is mandated by the government, as in Denmark. Often it is simply a customization or user preference.
- *Punctuation.* Another common option is whether to treat punctuation (including spaces) as base characters or treat such characters as only making a level 4 difference.
- *User-Defined Rules.* Such rules provide specified results for given combinations of letters. For example, in an index, an author may wish to have symbols sorted as if they were spelled out; thus "?" may sort as if it were the string "question mark".
- *Merged Tailorings.* An option may allow the merging of sets of rules for different languages. For example, someone may want Latin characters sorted as in French, and Arabic characters sorted as in Persian. In such an approach, generally one of the tailorings is designated the “master” in cases of conflicting weights for a given character.
- *Script Order.* A user may wish to specify which scripts come first. For example, in a book index an author may want index entries in the predominant script that the book itself is written in to come ahead of entries for any other script. For example:

  b < ב < β < б [Latin < Hebrew < Greek < Cyrillic] *versus*
  β < b < б < ב [Greek < Latin < Cyrillic < Hebrew]

  Attempting to achieve this effect by introducing an extra strength level before the first (primary) level would give incorrect ordering results for strings which mix characters of more than one script.

- *Numbers.* A customization may be desired to allow sorting numbers in numeric order. If strings including numbers are merely sorted alphabetically, the string “A-10” comes before the string “A-2”, which is often not desired. This behavior can be customized, but it is complicated by ambiguities in recognizing numbers within strings (because they may be formatted according to different language conventions). Once each number is recognized, it can be preprocessed to convert it into a format that allows for correct numeric sorting, such as a textual version of the IEEE numeric format.

Phonetic sorting of Han characters requires use of either a lookup dictionary of words or, more typically, special construction of programs or databases to maintain an associated phonetic spelling for the words in the text.

1.5 Other Applications of Collation

The same principles about collation behavior apply to realms beyond sorting. In particular, searching should behave consistently with sorting. For example, if two letters are treated as identical base letters for sorting, then those letters should also be treated as identical for searching. The ability to set the maximal strength level is very important for searching.

Selection is the process of using the comparisons between the endpoints of a range, as when using a SELECT command in a database query. It is crucial that the range returned be correct according to the user's expectations. For example, if a German businessman making a database selection to sum up revenue in each of the cities from *O...* to *P...* for planning purposes does not realize that all cities starting with *Ö* were excluded because the query selection was using a Swedish collation, he will be one very unhappy customer.

A sequence of characters considered a unit in collation, such as *ch* in Slovak, represents a *collation grapheme cluster*. For applications of this concept, see Unicode Technical Standard #18, "Unicode

Regular Expressions" [UTS18]. For more information on grapheme clusters, see Unicode Standard Annex #29, "Unicode Text Segmentation" [UAX29].

1.6 Merging Sort Keys

Sort keys may need to be merged. For example, the simplest way to sort a database according to two fields is to sort field by field, sequentially. This gives the results in column one in *Table 6*. (The examples in this table are ordered using the **Shifted** option for handling variable collation elements such as the space character; see *Section 4 Variable Weighting* for details.) All the levels in Field 1 are compared first, and then all the levels in Field 2. The problem with this approach is that high-level differences in the second field are swamped by minute differences in the first field, which results in unexpected ordering for the first names.

Table 6. Merged Fields

Sequential	Weak First	Merged
$F1_{L1}$, $F1_{L2}$, $F1_{L3}$, $F2_{L1}$, $F2_{L2}$, $F2_{L3}$	$F1_{L1}$, $F2_{L1}$, $F2_{L2}$, $F2_{L3}$	$F1_{L1}$, $F2_{L1}$, $F1_{L2}$, $F2_{L2}$, $F1_{L3}$, $F2_{L3}$
di Silva Fred	disílva Fred	di Silva Fred
di Silva John	diSilva Fred	diSilva Fred
diSilva Fred	di Silva Fred	disílva Fred
diSilva John	di Silva John	di Silva John
disílva Fred	diSilva John	diSilva John
disílva John	disílva John	disílva John

A second way to do the sorting is to ignore all but base-level differences in the sorting of the first field. This gives the results in the second column. The first names are all in the right order, but the problem is now that the first field is not correctly ordered except by the base character level.

The correct way to sort two fields is to merge the fields, as shown in the "Merged" column. Using this technique, all differences in the fields are taken into account, and the levels are considered uniformly. Accents in all fields are ignored if there are any base character differences in any of the field, and case in all fields is ignored if there are accent or base character differences in any of the fields.

Since UCA 18.0, the effect of merging sort keys can be achieved by concatenating the strings with **U+FFFE** separating them. See Reserved Weights for details.

1.7 Performance

Collation is one of the most performance-critical features in a system. Consider the number of comparison operations that are involved in sorting or searching large databases, for example. Most production implementations will use a number of optimizations to speed up string comparison.

Strings are often preprocessed into sort keys, so that multiple comparisons operations are much faster. With this mechanism, a collation engine generates a *sort key* from any given string. The binary comparison of two sort keys yields the same result (less, equal, or greater) as the collation engine would return for a comparison of the original strings. Thus, for a given collation C and any two strings A and B:

$$A \leq B \text{ according to } C \text{ if and only if } \text{sortkey}(C, A) \leq \text{sortkey}(C, B)$$

However, simple string comparison is faster for any individual comparison, because the generation of a sort key requires processing an entire string, while differences in most string comparisons are found before all the characters are processed. Typically, there is a considerable difference in performance,

with simple string comparison being about 5 to 10 times faster than generating sort keys and then using a binary comparison.

Sort keys, on the other hand, can be much faster for multiple comparisons. Because binary comparison is much faster than string comparison, it is faster to use sort keys whenever there will be more than about 10 comparisons per string, if the system can afford the storage.

1.8 What Collation is Not

There are a number of common expectations about and misperceptions of collation. This section points out many things that collation is not and cannot be.

Collation is not aligned with character sets or repertoires of characters.

Swedish and German share most of the same characters, for example, but have very different sorting orders.

Collation is not code point (binary) order.

A simple example of this is the fact that capital Z comes before lowercase a in the code charts. As noted earlier, beginners may complain that a particular Unicode character is “not in the right place in the code chart.” That is a misunderstanding of the role of the character encoding in collation. While the Unicode Standard does not gratuitously place characters such that the binary ordering is odd, the only way to get the linguistically-correct order is to use a language-sensitive collation, not a binary ordering.

Collation is not a property of strings.

In a list of cities, with each city correctly tagged with its language, a German user will expect to see all of the cities sorted according to German order, and will not expect to see a word with *ö* appear after *z*, simply because the city has a Swedish name. As in the earlier example, it is crucially important that if a German businessman makes a database selection, such as to sum up revenue in each of of the cities from *O...* to *P...* for planning purposes, cities starting with *Ö* *not* be excluded.

Collation order is not preserved under concatenation or substring operations, in general.

For example, the fact that x is less than y does not mean that x + z is less than y + z, because characters may form contractions across the substring or concatenation boundaries. In summary:

x < y does not imply that xz < yz
x < y does not imply that zx < zy
xz < yz does not imply that x < y
zx < zy does not imply that x < y

Collation order is not preserved when comparing sort keys generated from different collation sequences.

Remember that sort keys are a preprocessing of strings according to a given set of collation features. Different features result in different binary sequences. For example, if there are two collations, F and G, where F is a French collation, and G is a German phonebook ordering, then:

- A ≤ B according to F if and only if sortkey(F, A) ≤ sortkey(F, B), *and*
- A ≤ B according to G if and only if sortkey(G, A) ≤ sortkey(G, B)
- The relation between sortkey(F, A) and sortkey(G, B) says nothing about whether A ≤ B according to F, or whether A ≤ B according to G.

Collation order is not a stable sort.

Stability is a property of a sort algorithm, not of a collation sequence.

Stable Sort

A *stable sort* is one where two records with a field that compares as equal will retain their order if sorted according to that field. This is a property of the sorting algorithm, *not* of the comparison mechanism. For example, a bubble sort is stable, while a Quicksort is not. This is a useful property, but cannot be accomplished by modifications to the comparison mechanism or tailorings. See also *Appendix A, Deterministic Sorting*.

Deterministic Comparison

A *deterministic comparison* is different. It is a comparison in which strings that are not canonical equivalents will not be judged to be equal. This is a property of the comparison, not of the sorting algorithm. This is not a particularly useful property—its implementation also requires extra processing in string comparison or an extra level in sort keys, and thus may degrade performance to little purpose. However, if a deterministic comparison is required, the specified mechanism is to append the NFD form of the original string after the sort key, in *Section 7.3, Form Sort Keys*. See also *Appendix A, Deterministic Sorting*.

A deterministic comparison is also sometimes referred to as a *stable (or semi-stable) comparison*. Those terms are not to be preferred, because they tend to be confused with *stable sort*.

Collation order is not fixed.

Over time, collation order will vary: there may be fixes needed as more information becomes available about languages; there may be new government or industry standards for the language that require changes; and finally, new characters added to the Unicode Standard will interleave with the previously-defined ones. This means that collations must be carefully versioned.

1.9 The Unicode Collation Algorithm

The Unicode Collation Algorithm (UCA) details how to compare two Unicode strings while remaining conformant to the requirements of the Unicode Standard. This standard includes the Default Unicode Collation Element Table (DUCET), which is data specifying the default collation order for all Unicode characters, and the CLDR root collation element table that is based on the DUCET. This table is designed so that it can be *tailored* to meet the requirements of different languages and customizations.

Briefly stated, the Unicode Collation Algorithm takes an input Unicode string and a Collation Element Table, containing mapping data for characters. It produces a sort key, which is an array of unsigned 16-bit integers. Two or more sort keys so produced can then be binary-compared to give the correct comparison between the strings for which they were generated.

The Unicode Collation Algorithm assumes multiple-level key weighting, along the lines widely implemented in IBM technology, and as described in the Canadian sorting standard [CanStd] and the International String Ordering standard [ISO14651].

By default, the algorithm makes use of three fully-customizable levels. For the Latin script, these levels correspond roughly to:

1. alphabetic ordering
2. diacritic ordering
3. case ordering.

A final level may be used for tie-breaking between strings not otherwise distinguished.

This design allows implementations to produce culturally acceptable collation, with a minimal burden on memory requirements and performance. In particular, it is possible to construct Collation Element Tables that use 32 bits of collation data for most characters.

Implementations of the Unicode Collation Algorithm are not limited to supporting only three levels. They are free to support a fully customizable 4th level (or more levels), as long as they can produce the same results as the basic algorithm, given the right Collation Element Tables. For example, an application which uses the algorithm, but which must treat some collection of special characters as ignorable at the first three levels *and* must have those specials collate in non-Unicode order (for example to emulate an existing EBCDIC-based collation), may choose to have a fully customizable 4th level. The downside of this choice is that such an application will require more storage, both for the Collation Element Table and in constructed sort keys.

The Collation Element Table may be tailored to produce particular culturally required orderings for different languages or locales. As in the algorithm itself, the tailoring can provide full customization for three (or more) levels.

1.9.1 Goals

The algorithm is designed to satisfy the following goals:

1. A complete, unambiguous, specified ordering for all characters in Unicode.
2. A complete resolution of the handling of canonical and compatibility equivalences as relates to the default ordering.
3. A complete specification of the meaning and assignment of collation levels, including whether a character is ignorable by default in collation.
4. A complete specification of the rules for using the level weights to determine the default collation order of strings of arbitrary length.
5. Allowance for override mechanisms (*tailoring*) to create language-specific orderings. Tailoring can be provided by any well-defined syntax that takes the default ordering and produces another well-formed ordering.
6. An algorithm that can be efficiently implemented, in terms of both performance and memory requirements.

Given the standard ordering and the tailoring for any particular language, any two companies or individuals—with their own proprietary implementations—can take any arbitrary Unicode input and produce exactly the same ordering of two strings. In addition, when given an appropriate tailoring this algorithm can pass the Canadian and ISO 14651 benchmarks ([CanStd], [ISO14651]).

> **Note:** The Default Unicode Collation Element Table does not explicitly list weights for all assigned Unicode characters. However, the algorithm is well defined over *all* Unicode code points. See *Section 10.1.2, Unassigned and Other Code Points*.

1.9.2 Non-Goals

The Default Unicode Collation Element Table (DUCET) explicitly does not provide for the following features:

1. *Reversibility:* from a Collation Element one is not guaranteed to be able to recover the original character.
2. *Numeric formatting:* numbers composed of a string of digits or other numerics will not necessarily sort in *numerical order.*
3. *API:* no particular API is specified or required for the algorithm.
4. *Title sorting:* removing articles such as *a* and *the* during bibliographic sorting is not provided.
5. *Stability of binary sort key values between versions:* weights in the DUCET may change between versions. For more information, see "Collation order is not a stable sort" in *Section 1.8, What Collation is Not*.

6. *Linguistic applicability:* to meet most user expectations, a linguistic tailoring is needed. For more information, see *Section 8, Tailoring*.

The feature of linguistic applicability deserves further discussion. DUCET does not and cannot actually provide linguistically correct sorting for every language without further tailoring. That would be impossible, due to conflicting requirements for ordering different languages that share the same script. It is not even possible in the specialized cases where a script may be predominantly used by a single language, because of the limitations of the DUCET table design and because of the requirement to minimize implementation overhead for all users of DUCET.

Instead, the goal of DUCET is to provide a reasonable default ordering for all scripts that are *not* tailored. Any characters used in the language of primary interest for collation are expected to be tailored to meet all the appropriate linguistic requirements for that language. For example, for a user interested primarily in the Malayalam language, DUCET would be tailored to get all details correct for the expected Malayalam collation order, while leaving other characters (Greek, Cyrillic, Han, and so forth) in the default order, because the order of those other characters is not of primary concern. Conversely, a user interested primarily in the Greek language would use a Greek-specific tailoring, while leaving the Malayalam (and other) characters in their default order in the table.

2 Conformance

The Unicode Collation Algorithm does not restrict the many different ways in which implementations can compare strings. However, any Unicode-conformant implementation that purports to implement the Unicode Collation Algorithm must do so as described in this document.

A conformance test for the UCA is available. See *Section 12.2, Conformance Tests*.

The Unicode Collation Algorithm is a *logical* specification. Implementations are free to change any part of the algorithm as long as any two strings compared by the implementation are ordered the same as they would be by the algorithm as specified. Implementations may also use a different format for the data in the Default Unicode Collation Element Table. The sort key is a logical intermediate object: if an implementation produces the same results in comparison of strings, the sort keys can differ in format from what is specified in this document. (See *Section 9, Implementation Notes*.)

2.1 Basic Conformance Requirements

The conformance requirements of the Unicode Collation Algorithm are as follows:

UTS10-C1. *For a given Unicode Collation Element Table, a conformant implementation shall replicate the same comparisons of strings as those produced by Section 7, Main Algorithm.*

In particular, a conformant implementation must be able to compare any two canonical-equivalent strings as being equal, for all Unicode characters supported by that implementation.

UTS10-C2. *A conformant implementation shall support at least three levels of collation.*

A conformant implementation is only required to implement three levels. However, it may implement four (or more) levels if desired.

UTS10-C3. *A conformant implementation that supports any of the following features: backward levels, variable weighting, and semi-stability (S3.10), shall do so in accordance with this specification.*

A conformant implementation is not required to support these features; however, if it does, it must interpret them properly. If an implementation intends to support the Canadian standard [CanStd] then it should implement a backwards secondary level.

UTS10-C4. *An implementation that claims to conform to the UCA must specify the UCA version it conforms to.*

The version number of this document is synchronized with the version of the Unicode Standard which specifies the repertoire covered.

UTS10-C5. *An implementation claiming conformance to Matching and Searching according to UTS #10 shall meet the requirements described in Section 11, Searching and Matching.*

2.2 Additional Conformance Requirements

If a conformant implementation compares strings in a legacy character set, it must provide the same results as if those strings had been transcoded to Unicode. The implementation should specify the conversion table and transcoding mechanism.

A claim of conformance to C6 (UCA parametric tailoring) from earlier versions of the Unicode Collation Algorithm is to be interpreted as a claim of conformance to LDML parametric tailoring. See *Setting Options* in [UTS35Collation].

An implementation that supports a parametric reordering which is not based on CLDR should specify the reordering groups.

3 Definitions and Notation

The Unicode Collation Algorithm depends on the concept of mapping characters in Unicode strings to sequences of collation weights called sort keys. Those sort keys, in turn, can be directly compared to determine the relative order of the strings. This section provides precise definitions of the special terminology used in the algorithm and its intermediate steps, along with explanation of the notation used in examples and in the discussion of the algorithm.

3.1 Collation Weights, Elements, and Levels

The basic concepts of collation weights, collation elements, and collation levels are defined here first, as all other aspects of the Unicode Collation Algorithm depend fundamentally on those concepts.

UTS10-D1. *Collation Weight:* A non-negative integer used in the UCA to establish a means for systematic comparison of constructed sort keys.

UTS10-D2. *Collation Element:* An ordered list of collation weights.

The term collation element is abbreviated as *CE*.

UTS10-D3. *Collation Level:* The position of a collation weight in a collation element.

In other words, the collation level refers to the first position, second position, and so forth, in a collation element. The *collation level* can also be used to refer collectively to all the weights at the same relative position in a sequence of collation elements.

Unless otherwise noted, all weights used in the example collation elements in this specification are displayed in hexadecimal format. Collation elements are shown in square brackets, with the collation weights for each level separated by dots for clarity. For example:

```
[.06D9.0020.0002]
```

UTS10-D4. *Primary Weight:* The first collation weight in a collation element.

A primary weight is also called the *Level 1* weight. Level 1 is also abbreviated as *L1*.

UTS10-D5. *Secondary Weight:* The second collation weight in a collation element.

A secondary weight is also called the *Level 2* weight. Level 2 is also abbreviated as *L2*.

UTS10-D6. *Tertiary Weight:* The third collation weight in a collation element.

A tertiary weight is also called the *Level 3* weight. Level 3 is also abbreviated as *L3*.

UTS10-D7. *Quaternary Weight:* The fourth collation weight in a collation element.

A quaternary weight is also called the *Level 4* weight. Level 4 is also abbreviated as *L4*.

In principle, collation levels can extend past Level 4 to add additional levels, but the specification of the Unicode Collation Algorithm does not require defining more levels. In some special cases, such as support of Japanese collation, an implementation may need to define additional levels.

For convenience, this specification uses subscripted numbers after the symbol referring to a particular collation element to refer to the collation weights of that collation element at designated levels. Thus, for a collation element X, X_1 refers to the primary weight, X_2 refers to the secondary weight, X_3 refers to the tertiary weight, and X_4 refers to the quaternary weight.

3.2 Ignorables

UTS10-D8. *Ignorable Weight:* A collation weight whose value is zero.

In the 4-digit hexadecimal format used in this specification, ignorable weights are expressed as "0000".

Ignorable weights are passed over by the rules that construct sort keys from sequences of collation elements. Thus, their presence in collation elements does not impact the comparison of strings using the resulting sort keys. The judicious assignment of ignorable weights in collation elements is an important concept for the UCA.

The following definitions specify collation elements which have particular patterns of ignorable weights in them.

UTS10-D9. *Primary Collation Element:* A collation element whose Level 1 weight is not an ignorable weight.

UTS10-D10. *Secondary Collation Element:* A collation element whose Level 1 weight is an ignorable weight but whose Level 2 weight is not an ignorable weight.

UTS10-D11. *Tertiary Collation Element:* A collation element whose Level 1 and Level 2 weights are ignorable weights but whose Level 3 weight is not an ignorable weight.

UTS10-D12. *Quaternary Collation Element:* A collation element whose Level 1, Level 2, and Level 3 weights are ignorable weights but whose Level 4 weight is not an ignorable weight.

UTS10-D13. *Completely Ignorable Collation Element:* A collation element which has ignorable weights at all levels.

UTS10-D14. *Ignorable Collation Element:* A collation element which is not a primary collation element.

The term *ignorable collation element* is a convenient cover term for any type of collation element which has a zero primary weight. It includes secondary, tertiary, quaternary, and completely ignorable collation elements. In contrast, a primary collation element, which by definition does not have a zero primary weight, can also be referred to as a ***non**-ignorable collation element*.

UTS10-D15. *Level N Ignorable:* A collation element which has an ignorable weight at level N, but not at level N+1.

This concept is useful for parameterized expressions with weight level as a parameter. For example "Level 1 ignorable" is a synonym for a secondary collation element. This alternate terminology is generally avoided in this specification, however, because of the potential for confusion.

UTS10-D16. *Variable Collation Element:* A primary collation element with a low (but non-zero) value for its primary weight.

> Low primary weights are generally reserved for punctuation and symbols, to enable special handling of those kinds of characters. Variable collation elements are subject to special rules when constructing sort keys. See *Section 4, Variable Weighting*. In the Default Unicode Collation Element Table [Allkeys] the primary weights of all variable collation elements are prefixed with an asterisk instead of a dot, so that they can be clearly identified.

> Since UCA 18.0, the lowest primary weight **0200** is reserved for the primary weight of the **U+FFFE** collation element. This primary weight is never variable.

The relationship between these terms for patterns of ignorable weights in collation elements, together with schematic examples of the corresponding collation elements, is shown in the following table, constructed on the assumption that collation elements have four collation levels. Note that quaternary collation elements have the same schematic pattern of weights as variable collation elements which have been *shifted*.

Schematic Example	Main Term	General Type	Level Notation
`[.nnnn.nnnn.nnnn.nnnn]`	Primary Collation Element	Non-ignorable	Level 0 Ignorable
`[*nnnn.nnnn.nnnn.nnnn]`	Variable Collation Element (not shifted)		
`[.0000.nnnn.nnnn.nnnn]`	Secondary Collation Element	Ignorable	Level 1 Ignorable
`[.0000.0000.nnnn.nnnn]`	Tertiary Collation Element		Level 2 Ignorable
`[.0000.0000.0000.nnnn]`	Quaternary Collation Element		Level 3 Ignorable
	Variable Collation Element (shifted)		
`[.0000.0000.0000.0000]`	Completely Ignorable Collation Element		Level 4 Ignorable

For collation element tables, such as DUCET, which only use three levels of collation weights, the terminology would adjust as follows, and similar adjustments can be made to express collation element tables using more than four weight levels.

Schematic Example	Main Term	General Type	Level Notation
`[.nnnn.nnnn.nnnn]`	Primary Collation Element	Non-ignorable	Level 0 Ignorable
`[*nnnn.nnnn.nnnn]`	Variable Collation Element (not shifted)		
`[.0000.nnnn.nnnn]`	Secondary Collation Element	Ignorable	Level 1 Ignorable
`[.0000.0000.nnnn]`	Tertiary Collation Element		Level 2 Ignorable
`[.0000.0000.0000]`	Completely Ignorable Collation Element		Level 3 Ignorable

3.3 Mappings

An important feature of the Unicode Collation Algorithm is the systematic mapping of Unicode characters (in Unicode strings) to sequences of collation elements, for the purpose of comparing those strings. The sequence of collation elements is then converted into a sort key, suitable for direct comparison. This section defines the various types of collation element mappings discussed in the specification of the algorithm.

UTS10-D17. *Collation Element Mapping:* A mapping from one (or more) Unicode characters to one (or more) collation elements.

Effectively, a given collation element table defines a mathematical function. It is instantiated as a list of collation element mappings. Collectively, the input for those mappings, which generally consists of all Unicode code points plus some well-defined set of short sequences of Unicode characters, constitutes

the *domain* of the function. Collectively, the output for those mappings consists of a defined list of collation element sequences; the set of which constitutes the *codomain* of the function. And the collation element table itself constitutes the *graph* of the function.

> **Note:** For formal completeness, a collation element mapping is usually defined to include in its domain *all* Unicode code points, including noncharacters and unassigned, reserved code points. The Unicode Collation Algorithm specifies the mapping for unassigned code points, as well as some ranges of assigned characters, by means of *implicit weights*. See *Section 10, Weight Derivation* for details. However, because specific collation element tables such as the Default Unicode Collation Element Table (DUCET) generally only contain a list of collation element mappings for assigned characters, and maps those assigned characters to collation elements with *explicit* weights, the definitions in this section are simplified by referring to the *input* values just in terms of "Unicode characters".

Collation element mappings are divided into subtypes, based on a distinction between whether the *input* of the mapping constitutes a single Unicode character or a sequence of Unicode characters, and a separate distinction between whether the *output* of the mapping constitutes a single collation element or a sequence of collation elements.

UTS10-D18. *Simple Mapping:* A collation element mapping from one Unicode character to one collation element.

UTS10-D19. *Expansion:* A collation element mapping from one Unicode character to a sequence of more than one collation element.

UTS10-D20. *Many-to-One Mapping:* A collation element mapping from more than one Unicode character to one collation element.

UTS10-D21. *Many-to-Many Mapping:* A collation element mapping from more than one Unicode character to a sequence of more than one collation element.

UTS10-D22. *Contraction:* Either a many-to-one mapping or a many-to-many mapping.

Both many-to-many mappings and many-to-one mappings are referred to as *contractions* in the discussion of the Unicode Collation Algorithm, even though many-to-many mappings often do not actually shorten anything. The main issue for implementations is that for both many-to-one mappings and many-to-many mappings, the weighting algorithm must first identify a sequence of characters in the input string and "contract" them together as a unit for weight lookup in the table. The identified unit may then be mapped to any number of collation elements. Contractions pose particular issues for implementations, because all eligible contraction targets must be identified first, before the application of simple mappings, so that processing for simple mappings does not bleed away the context needed to correctly identify the contractions.

3.3.1 Simple Mappings

Most of the mappings in a collation element table are simple mappings.

The following table shows several instances of simple mappings. These simple mappings are used in the examples illustrating the algorithm.

Character	Collation Element	Name
0300 "`"	[.0000.0021.0002]	COMBINING GRAVE ACCENT
0061 "a"	[.1C47.0020.0002]	LATIN SMALL LETTER A
0062 "b"	[.1C60.0020.0002]	LATIN SMALL LETTER B
0063 "c"	[.1C7A.0020.0002]	LATIN SMALL LETTER C
0043 "C"	[.1C7A.0020.0008]	LATIN CAPITAL LETTER C
0064 "d"	[.1C8F.0020.0002]	LATIN SMALL LETTER D

0065 "e"	[.1CAA.0020.0002]	LATIN SMALL LETTER E

3.3.2 Expansions

The following table shows an example of an expansion. The first two rows show simple mappings for "a" and "e". The third row shows the single Unicode character *æ* is mapped to a sequence of collation elements, rather than a single collation element.

Character	Collation Elements	Name
0061 "a"	[.1C47.0020.0002]	LATIN SMALL LETTER A
0065 "e"	[.1CAA.0020.0002]	LATIN SMALL LETTER E
00E6 "æ"	[.1C47.0020.0004][.0000.0110.0004][.1CAA.0020.0004]	LATIN SMALL LETTER AE

In the row with the expansion for "æ", the two underlined primary weights have the same values as the primary weights for the simple mappings for "a" and "e", respectively. This is the basis for establishing a primary equivalence between "æ" and the sequence "ae".

3.3.3 Contractions

Similarly, where the sequence *ch* is treated as a single digraph letter, as for instance in Slovak, it is represented as a mapping from two characters to a single collation element, as shown in the following example:

Character	Collation Element	Name
0063 0068 "ch"	[.1D19.0020.0002]	LATIN SMALL LETTER C, LATIN SMALL LETTER H

In this example, the collation element [.1D19.0020.0002] has a primary weight one greater than the primary weight for the letter *h* by itself, so that the sequence *ch* will collate after *h* and before *i*. This example shows the result of a tailoring of collation element mappings to weight sequences of letters as a single unit.

3.3.4 Many-to-Many Mappings

In some cases a sequence of two or more characters is mapped to a sequence of two or more collation elements. For example, this technique is used in the Default Unicode Collation Element Table [Allkeys] to handle weighting of rearranged sequences of Thai or Lao left-side-vowel + consonant. See *Section 6.1.1, Rearrangement and Contractions*.

Certain characters may both expand and contract. See *Section 1.3, Contextual Sensitivity.*

3.4 Collation Element Tables

UTS10-D23. *Collation Element Table:* A table of collation element mappings.

The basic idea of a collation element table is that it contains the collation weight information necessary to construct sort keys for Unicode strings.

UTS10-D24. *Explicit Weight Mapping:* A mapping to one (or more) collation elements which is explicitly listed in a collation element table.

UTS10-D25. *Implicit Weight Mapping:* A mapping to one (or more) collation elements which is *not* explicitly listed in a collation element table, but which is instead derived by rule.

The convention used by the Unicode Collation Algorithm is that the mapping for any character which is not listed explicitly in a given collation element table is instead determined by the implicit weight derivation rules. This convention extends to all unassigned code points, so that all Unicode strings can have determinant sort keys constructed for them. See *Section 10, Weight Derivation* for the rules governing the assignment of implicit weights.

Implementations can produce the same result using various representations of weights. In particular, while the Default Unicode Collation Element Table [Allkeys] stores weights of all levels using 16-bit integers, and such weights are shown in examples in this document, other implementations may choose to store weights in larger or smaller integer units, and may store weights of different levels in integer units of different sizes. See *Section 9, Implementation Notes*.

The specific collation weight values shown in examples are illustrative only; they may not match the weights in the latest Default Unicode Collation Element Table [Allkeys].

UTS10-D26. *Minimum Weight at a Level:* The least weight in any collation element in a given collation element table, at a specified level.

The minimum weight at level n is abbreviated with the notation: MIN_n.

UTS10-D27. *Maximum Weight at a Level:* The greatest weight in any collation element in a given collation element table, at a specified level.

The maximum weight at level n is abbreviated with the notation: MAX_n.

3.5 Input Matching

The Unicode Collation Algorithm involves a step where the content of an input string is matched up, piece by piece, against the mappings in the collation element table, to produce an array of collation elements, which in turn is processed further into a sort key. This section provides definitions for terms relevant to that input matching process.

UTS10-D28. *Input Match:* An association between a sequence of one or more Unicode characters in the input string to a collation element mapping in the collation element table, where the sequence from the input string exactly matches the Unicode characters specified for that mapping.

The exact rules for processing an input string systematically are specified in *Section 7.2, Produce Collation Element Arrays*. Those rules determine how to walk down an input string, to pull out the candidate character (or sequence of characters) for input matching, in order to find each successive relevant collation element.

UTS10-D29. *Single Character Match:* An input match for a single character.

UTS10-D30. *Contraction Match:* An input match for a sequence of characters.

UTS10-D31. *Contiguous Match:* A contraction match for a contiguous sequence of characters.

UTS10-D32. *Discontiguous Match:* A contraction match for a discontiguous sequence of characters.

The concept of a discontiguous match is necessary for the UCA, because the algorithm requires identical outcomes for canonically equivalent input sequences. UCA specifies that an input string shall first be normalized into NFD. (Normalize Each String) However, normalization of a combining character sequence can, in principle, separate a combining mark from a base character for which a contraction mapping has been defined, as shown in the following example:

<***a**, combining dot below,* ***combining ring above***>
≡
<***a**,* ***combining ring above****, combining dot below*>.

In this example, the first line, with the *dot below* occurring ahead of the *ring above*, is normalized to NFD; however, the sequence shown in the second line is canonically equivalent. Now, if a contraction has been defined in a collation element table for the sequence <**a**, **combining ring above**>, as might be the case for a Danish string ordering, for which "å" sorts after "z", then that sequence must be found and a weight matching be done, *even in the case where the match has become discontiguous* as a result of the string normalization. See *Section 7.2, Produce Collation Element Array*, for details about how the UCA specifies that discontiguous matching be done.

UTS10-D33. *Non-Starter:* An assigned character with Canonical_Combining_Class ≠ 0.

This definition of non-starter is based on the definition of *starter*. See D107 in [Unicode]. By the definition of Canonical_Combining_Class, a non-starter must be a combining mark; however, not all combining marks are non-starters, because many combining marks have Canonical_Combining_Class = 0. A non-starter cannot be an unassigned code point: all unassigned code points are starters, because their Canonical_Combining_Class value defaults to 0.

UTS10-D34. *Blocking Context:* The presence of a character B between two characters C_1 and C_2, where ccc(B) = 0 *or* ccc(B) ≥ ccc(C_2).

The notation ccc(B) is an abbreviation for "the Canonical_Combining_Class value of B".

UTS10-D35. *Unblocked Non-Starter:* A non-starter C_2 which is not in a blocking context with respect to a preceding character C_1 in a string.

In the context <C_1 ... B ... C_2>, if there is no intervening character B which meets the criterion for being a *blocking context*, and if C_2 is a non-starter, then it is also an *unblocked non-starter*.

The concept of an unblocked non-starter is pertinent to the determination of sequences which constitute a discontiguous match. The process to find a longest match can continue skipping over subsequent unblocked non-starters in an attempt to find the longest discontiguous match. The search for such a match will terminate at the first non-starter in a blocked context, or alternatively, at the first starter encountered during the lookahead. The details of this discontiguous matching are spelled out in *Section 7.2, Produce Collation Element Array*.

A sequence of characters which otherwise would result in a contraction match can be made to sort as separate characters by inserting, someplace within the sequence, a starter that maps to a completely ignorable collation element. By definition this creates a blocking context, even though the completely ignorable collation element would not otherwise affect the assigned collation weights. There are two characters, U+00AD SOFT HYPHEN and U+034F COMBINING GRAPHEME JOINER, that are particularly useful for this purpose. These can be used to separate sequences of characters that would normally be weighted as units, such as Slovak "ch" or Danish "aa". See *Section 8.3, Use of Combining Grapheme Joiner*.

3.6 Sort Keys

This section provides definitions for *sort key* and for other concepts used in the assembly of sort keys. See also *Section 9.7, Incremental Comparison* for a discussion of implementation tradeoffs which impact performance of string comparisons.

UTS10-D36. *Sort Key:* An array of non-negative integers, associated with an input string, systematically constructed by extraction of collation weights from a collation element array, and suitable for binary comparison.

The Unicode Collation Algorithm maps input strings, which are very complicated to compare accurately, down to these simple arrays of integers called *sort keys*, which can then be compared very reliably and quickly with a low-level memory comparison operation, such as memcmp().

The discussion and examples in this specification use the same 16-bit hexadecimal notation for the integers in a sort key as is used for the collation weights. There is, however, no requirement that integers in a sort key be stored with the same integral width as collation weights stored in a collation element table. The only practical requirement is that all of the integers in a given sort key (and any sort key to be compared with it) be stored with the *same* integral width. That means that they constitute simple arrays, amenable to efficient, low-level array processing.

UTS10-D37. *Level Separator:* A low integer weight used in the construction of sort keys to separate collation weights extracted from different levels in the collation element array.

The UCA uses the value zero (0000) for the level separator, to guarantee that the level separator has a lower value than any of the actual collation weights appended to the sort key from the collation element array. Implementations can, however, use a non-zero value, as long as that value is lower than the minimum weight at that level.

3.7 Comparison

This section provides the definitions and notational conventions used in describing the comparison of collation elements and of strings.

3.7.1 Equality

UTS10-D38. Two collation elements are *primary equal* if and only if the primary weight of each collation element is equal.

UTS10-D39. Two collation elements are *secondary equal* if and only if the secondary weight of each collation element is equal *and* the collation elements are primary equal.

UTS10-D40. Two collation elements are *tertiary equal* if and only if the tertiary weight of each collation element is equal *and* the collation elements are secondary equal.

UTS10-D41. Two collation elements are *quaternary equal* if and only if the quaternary weight of each collation element is equal *and* the collation elements are tertiary equal.

3.7.2 Inequality

UTS10-D42. A collation element X is *primary less than* collation element Y if and only if the primary weight of X is less than the primary weight of Y.

UTS10-D43. A collation element X is *secondary less than* collation element Y if and only if

- X is primary less than Y *or*
- X is primary equal to Y *and* the secondary weight of X is less than the secondary weight of Y.

UTS10-D44. A collation element X is *tertiary less than* collation element Y if and only if

- X is secondary less than Y *or*
- X is secondary equal to Y *and* the tertiary weight of X is less than the tertiary weight of Y.

UTS10-D45. A collation element X is *quaternary less than* collation element Y if and only if

- X is tertiary less than Y *or*
- X is tertiary equal to Y *and* the quaternary weight of X is less than the quaternary weight of Y.

Other inequality comparison relations can be defined by corollary, based on the definitions of the equality and/or less than relations, on a level-by-level basis. For example, a collation element X is *primary greater than* collation element Y if and only if Y is primary less than X. A collation element X is *primary less than or equal to* collation element Y if and only if (X is primary less than Y) or (X is primary equal to Y). And so forth for other levels.

3.7.3 Notation for Collation Element Comparison

This specification uses a notation with subscripted numbers following the equals sign to represent the level-specific equality relations, as shown in *Table 7*.

Table 7. Equals Notation

Notation	Reading	Meaning
$X =_1 Y$	*X is primary equal to Y*	$X_1 = Y_1$

$X =_2 Y$	*X is secondary equal to Y*	$X_2 = Y_2$ and $X =_1 Y$
$X =_3 Y$	*X is tertiary equal to Y*	$X_3 = Y_3$ and $X =_2 Y$
$X =_4 Y$	*X is quaternary equal to Y*	$X_4 = Y_4$ and $X =_3 Y$

Similarly, subscripted numbers following the less than sign are used to indicate the level-specific less than comparison relations, as shown in *Table 8*.

Table 8. Less Than Notation

Notation	Reading	Meaning
$X <_1 Y$	*X is primary less than Y*	$X_1 < Y_1$
$X <_2 Y$	*X is secondary less than Y*	$X <_1 Y$ or ($X =_1 Y$ and $X_2 < Y_2$)
$X <_3 Y$	*X is tertiary less than Y*	$X <_2 Y$ or ($X =_2 Y$ and $X_3 < Y_3$)
$X <_4 Y$	*X is quaternary less than Y*	$X <_3 Y$ or ($X =_3 Y$ and $X_4 < Y_4$)

Other symbols for inequality relations are given their customary definitions in terms of the notational conventions just described:

- $X \leq_n Y$ if and only if $X <_n Y$ or $X =_n Y$
- $X >_n Y$ if and only if $Y <_n X$
- $X \geq_n Y$ if and only if $Y \leq_n X$

3.7.4 Notation for String Comparison

This notation for collation elements is also adapted to refer to ordering between strings, as shown in *Table 9*, where A and B refer to two strings.

Table 9. Notation for String Ordering

Notation	Meaning
$A <_2 B$	A is less than B, and there is a primary or secondary difference between them
$A <_2 B$ and $A =_1 B$	A is less than B, but there is *only* a secondary difference between them
$A \equiv B$	A and B are equivalent (equal at all levels) according to a given Collation Element Table
$A = B$	A and B are bit-for-bit identical

Where only plain text ASCII characters are available the fallback notation in *Table 10* may be used.

Table 10. Fallback Notation

Notation	Fallback
$X <_n Y$	X <[n] Y
X_n	X[n]
$X \leq_n Y$	X <=[n] Y
$A \equiv B$	A =[a] B

3.8 Parametric Settings

An implementation of the Unicode Collation Algorithm may be tailored with parametric settings. Two important parametric settings relate to the specific handling of variable collation elements and the treatment of "backward accents". For the details about variable collation elements, see Section 4, Variable Weighting.

3.8.1 Backward Accents

In some French dictionary ordering traditions, accents are sorted from the back of the string to the front of the string. This behavior is not marked in the Default Unicode Collation Element Table, but may be specified with a parametric setting. In such a case, the collation elements for the accents would be *backward* at Level 2.

UTS10-D46. *Backward at a Level:* A setting which specifies that once an array of collation elements has been created for a given input string, the collation weights at that particular level be scanned in *reversed* order when constructing the sort key for that input string. Since UCA 18.0, text *segments* separated by the lowest primary weight **0200**, the primary weight of the U+FFFE collation element, are processed in forward segment order, and *within* each segment the secondary weights are compared backwards. For details see *Section 10.1.5, Reserved Weights.*

In principle, the specification of the UCA is consistent with a setting for backward at any particular level, or even at multiple levels, but the only practical use for this parametric setting is to be backward at the *second* level.

This parametric setting defaults to be forward (i.e., not "backward") at all levels.

4 Variable Weighting

Variable collation elements, which typically include punctuation characters and which may or may not include a subset of symbol characters, require special handling in the Unicode Collation Algorithm.

Based on the variable-weighting setting, variable collation elements can be either treated as quaternary collation elements or not. When they are treated as quaternary collation elements, any sequence of ignorable collation elements that immediately follows the variable collation element is also affected.

There are four possible options for variable weighted characters:

1. **Non-ignorable:** Variable collation elements are not reset to be quaternary collation elements. All mappings defined in the table are unchanged.
2. **Blanked:** Variable collation elements and any subsequent ignorable collation elements are reset so that all weights (except for the identical level) are zero. ~~It is the same as the Shifted Option, except that there is no fourth level.~~ This is equivalent to the Shifted option used with strength=tertiary.
3. **Shifted:** Variable collation elements are reset to zero at levels one through three. In addition, a new fourth-level weight is appended, whose value depends on the type, as shown in *Table 11*. Any subsequent ignorable collation elements following a variable collation element are reset so that their weights at levels one through four are zero.
   - A combining grave accent after a space would have the value `[.0000.0000.0000.0000]`.
   - A combining grave accent after a *Capital A* would be unchanged.
4. ~~**Shift-Trimmed:** This option is the same as **Shifted**, except that all trailing FFFFs are trimmed from the sort key. This could be used to emulate POSIX behavior, but is otherwise not recommended.~~

Note: The L4 weight used for non-variable collation elements for the Shifted ~~and Shift-Trimmed~~ option can be any value which is greater than the primary weight of any variable collation element. In this document, it is simply set to FFFF, which is the maximum possible primary weight in the DUCET.

Review Note: The Properties and Algorithms Working Group recommends the removal of the Shift-Trimmed option. This has not yet been approved by the UTC.

In UCA versions 17.0 and earlier another option, Shift-Trimmed, was defined. This option was the same as Shifted, except that all trailing FFFFs were trimmed from the sort key. This option could be used to emulate POSIX behavior, but was otherwise not recommended.

In UCA versions 6.1 and 6.2 another option, IgnoreSP, was defined. That was a variant of Shifted that reduced the set of variable collation elements to include only spaces and punctuation, as in CLDR.

Table 11. L4 Weights for Shifted Variables

Type	L4	Examples
L1, L2, L3 = 0	0000	*null* `[.0000.0000.0000.0000]`
L1=0, L3 ≠ 0, following a Variable	0000	*combining grave* `[.0000.0000.0000.0000]`
L1 ≠ 0, Variable	old L1	*space* `[.0000.0000.0000.0209]`
L1 = 0, L3 ≠ 0, *not* following a Variable	FFFF	*combining grave* `[.0000.0035.0002.FFFF]`
L1 ≠ 0, *not* Variable	FFFF	*Capital A* `[.06D9.0020.0008.FFFF]`

The ~~variants of the~~ **Shifted** option provides for an improved ordering when the variable collation elements are ignorable, while still only requiring three fields to be stored in memory for each collation element. This option results in somewhat longer sort keys, although they can be compressed (see *Section 9.1, Reducing Sort Key Lengths* and *Section 9.3, Reducing Table Sizes)*.

4.1 Examples of Variable Weighting

Table 12 shows the differences between orderings using the different options for variable collation elements. In this example, sample strings differ by the third character: a letter, *space,* '-' *hyphen-minus (002D)*, or '-' *hyphen (2010);* followed by an uppercase/lowercase distinction.

Table 12. Comparison of Variable Ordering

Non-ignorable	Blanked	Shifted	Shifted (CLDR)	~~Shift-Trimmed~~
de luge de Luge de-luge de-Luge de-luge de-Luge death deluge deLuge demark	death de luge de-luge deluge de-luge de Luge de-Luge deLuge de-Luge demark	death de luge de-luge de-luge deluge de Luge de-Luge de-Luge deLuge demark	death de luge de-luge de-luge deluge de Luge de-Luge de-Luge deLuge demark	~~death~~ ~~deluge~~ ~~de luge~~ ~~de-luge~~ ~~de-luge~~ ~~deLuge~~ ~~de Luge~~ ~~de-Luge~~ ~~de-Luge~~ ~~demark~~
🙰happy 🙰sad ♡happy ♡sad	🙰happy ♡happy 🙰sad ♡sad	🙰happy ♡happy 🙰sad ♡sad	🙰happy 🙰sad ♡happy ♡sad	~~🙰happy~~ ~~♡happy~~ ~~🙰sad~~ ~~♡sad~~

The following points out some salient features of each of the columns in Table 12.

1. **Non-ignorable.** The words with *hyphen-minus* or *hyphen* are grouped together, but before all letters in the third position. This is because they are not ignorable, and have primary values that differ from the letters. The symbols ☠ and ♡ have primary differences.
2. **Blanked.** The words with *hyphen-minus* or *hyphen* are separated by "deluge", because the letter "l" comes between them in Unicode code order. The symbols ☠ and ♡ are *ignored* on levels 1-3.
3. **Shifted.** This is illustrated with two subtypes, which differ in their handling of symbols:
   a. **Shifted (DUCET).** The *hyphen-minus* and *hyphen* are grouped together, and their differences are less significant than the casing differences in the letter "l". This grouping results from the fact that they are ignorable, but their fourth level differences are according to the original primary order, which is more intuitive than Unicode order. The symbols ☠ and ♡ are *ignored* on levels 1-3.
   b. **Shifted (CLDR).** The same as Shifted (DUCET), except that the symbols ☠ and ♡ have primary differences. This change results from the CLDR base tailoring of DUCET, which lowers the boundary between variable collation elements and other primary collation elements, to exclude symbols from variable weighting.
4. ~~**Shift-Trimmed.** Note how "deLuge" comes between the cased versions with spaces and hyphens. The symbols ☠ and ♡ are *ignored* on levels 1-3.~~

4.2 Interleaving

Primary weights for variable collation elements are not *interleaved* with other primary weights. This allows for more compact storage of tables of collation weights in memory. Rather than using one bit per collation element to determine whether the collation element is variable, the implementation only needs to store the maximum primary value for all the variable collation elements. All collation elements with primary weights from ~~1~~ **0201** to that maximum are variables; all other collation elements are not.

Note: Since UCA 18.0, the lowest primary weight **0200** is reserved for the primary weight of the **U+FFFE** collation element. This primary weight is never variable.

5 Well-Formedness of Collation Element Tables

To be well-formed, a collation element table must meet certain criteria. For example:

- Collation elements have to consist of collation weights (non-negative integers).
- Each collation element must have the same number of levels (not 4 for one and 7 for the next, and so on).
- There must not be ambiguous entries implying different collation elements mapped to the same character.

In addition, a well-formed Collation Element Table for the Unicode Collation Algorithm meets the following, less obvious, well-formedness conditions:

WF1. Except in special cases detailed in *Section 9.2, Large Weight Values*, no collation element can have a zero weight at Level N and a non-zero weight at Level N-1.

- For example, the secondary weight can only be ignorable if the primary weight is ignorable.
- For a detailed example of what happens if this condition is not met, see *Section 7.5 Rationale for Well-Formed Collation Element Tables*.

WF2. Secondary weights of secondary collation elements must be strictly greater than secondary weights of all primary collation elements. Tertiary weights of tertiary collation elements must be strictly greater than tertiary weights of all primary and secondary collation elements.

- Given collation elements [A, B, C], [0, D, E], [0, 0, F], where the letters are non-zero weights, the following must be true:
  - D > B
  - F > C

  - F > E
- For a detailed example of what happens if this condition is not met, see *Section 7.5 Rationale for Well-Formed Collation Element Tables*.

WF3. No variable collation element has an ignorable primary weight.

WF4. For all variable collation elements U, V, if there is a collation element W such that $U_1 \leq W_1$ and $W_1 \leq V_1$, then W is also variable.

- This provision prevents interleaving.

WF5. If a table contains a contraction consisting of a sequence of N code points, with N > 2 and the last code point being a non-starter, then the table must also contain a contraction consisting of the sequence of the first N-1 code points.

- For example, if "ae<umlaut>" is a contraction, then "ae" must be a contraction as well.
- ~~For a principled exception to this well-formedness condition in DUCET, see *Section 6.7 Tibetan and Well-Formedness of DUCET*.~~

6 Default Unicode Collation Element Table

The Default Unicode Collation Element Table is provided in [Allkeys]. This table provides a mapping from characters to collation elements for all the explicitly weighted characters. The mapping lists characters in the order that they are weighted. Any code points that are not explicitly mentioned in this table are given a derived collation element, as described in *Section 7, Weight Derivation*.

The Default Unicode Collation Element Table does not aim to provide precisely correct ordering for each language and script; tailoring is required for correct language handling in almost all cases. The goal is instead to have all the *other* characters, those that are not tailored, show up in a reasonable order. This is particularly true for contractions, because contractions can result in larger tables and significant performance degradation. Contractions are required in tailorings, but their use is kept to a minimum in the Default Unicode Collation Element Table to enhance performance.

6.1 Contractions in DUCET

In the Default Unicode Collation Element Table, contractions are necessary where a canonical decomposable character requires a distinct primary weight in the table, so that the canonical-equivalent character sequences are given the same weights. For example, Indic two-part vowels have primary weights as units, and their canonical-equivalent sequence of vowel parts must be given the same primary weight by means of a contraction entry in the table. The same applies to a number of precomposed Cyrillic characters with diacritic marks and to a small number of Arabic letters with *madda* or *hamza* marks.

6.1.1 Rearrangement and Contractions

Certain characters, such as the Thai vowels เ through ไ (and related vowels in the Lao, New Tai Lue, and Tai Viet scripts of Southeast Asia), are not represented in strings in phonetic order. The exact list of such characters is given by the Logical_Order_Exception property in the Unicode Character Database [UAX44]. For collation, they are conceptually rearranged by swapping them with the following character before further processing, because doing so places them in a syllabic position where they can be compared with other vowels that follow a consonant. This is currently done for the UCA by providing these sequences as many-to-many mappings (contractions) in the Default Unicode Collation Element Table.

Contractions are entered in the table for Thai, Lao, New Tai Lue, and Tai Viet vowels with the Logical_Order_Exception property. Because each of these scripts has four or five vowels that are represented in strings in visual order, those vowels cannot simply be weighted by their representation order in strings. One option is to preprocess relevant strings to identify and reorder all vowels with the Logical_Order_Exception property around the following consonant. That approach was used in Version 4.0 and earlier of the UCA. Starting with Version 4.1 of the UCA, contractions for the relevant

combinations of vowel+consonant have been entered in the Default Unicode Collation Element Table instead.

Contractions in DUCET for sequences involving characters that have the Logical_Order_Exception property in scripts using the visual order model simplify the statement of the UCA and certain aspects of its implementation. However, such an approach may not be optimal for *search* and *string matching* with the UCA. One approach for searching and matching in Thai and similar languages is to simply create a tailoring that undoes the contraction entries in DUCET for scripts using the visual order model. Removal of the contractions narrows the match boundaries and avoids the need for contraction lookup, and thereby may improve performance for searching and matching.

6.1.2 Omission of Generic Contractions

Generic contractions of the sort needed to handle digraphs such as "ch" in Spanish or Czech sorting should be dealt with in tailorings to the default table—because they often vary in ordering from language to language, and because every contraction entered into the default table has a significant implementation cost for all applications of the default table, even those which may not be particularly concerned with the affected script. See the Unicode Common Locale Data Repository [CLDR] for extensive tailorings of the DUCET for various languages, including those requiring contractions.

6.2 Weighting Considerations in DUCET

The Default Unicode Collation Element Table is constructed to be consistent with the Unicode Normalization algorithm, and to respect the Unicode character properties. It is not, however, merely algorithmically derivable based on considerations of canonical equivalence and an inspection of character properties, because the assignment of levels also takes into account characteristics of particular scripts. For example, the combining marks generally have *secondary collation elements*; however, the Indic combining vowels are given non-zero Level 1 weights, because they are as significant in sorting as the consonants.

Any character may have variant forms or applied accents which affect collation. Thus, for FULL STOP there are three compatibility variants: a fullwidth form, a compatibility form, and a small form. These get different tertiary weights accordingly. For more information on how the table was constructed, see *Section 7.2, Tertiary Weight Table*.

6.3 Overall Order of DUCET

Table 13 summarizes the overall ordering of the collation elements in the Default Unicode Collation Element Table. The collation elements are ordered by primary, secondary, and tertiary weights, with primary, secondary, and tertiary weights for variables blanked (replaced by "0000"). Entries in the table which contain a sequence of collation elements have a multi-level ordering applied: comparing the primary weights first, then the secondary weights, and so on. This construction of the table makes it easy to see the order in which characters would be collated.

The weightings in the table are grouped by major categories. For example, whitespace characters come before punctuation, and symbols come before numbers. These groupings allow for programmatic reordering of scripts and other characters of interest, without table modification. For example, numbers can be reordered to be after letters instead of before. For more information, see the *Unicode Common Locale Data Repository* [CLDR].

The trailing and reserved primary weights must be the highest primaries, or else they would not function as intended. Therefore, they must not be subject to parametric reordering.

Unassigned-implicit primaries sort just before trailing weights. This is to facilitate CLDR Collation Reordering where the codes **Zzzz** and **other** (which are both used for “all other groups and scripts”) include the unassigned-implicit range. This range is reorderable.

Table 13. DUCET Ordering

Values	Type	Examples of Characters

$X_1, X_2, X_3 = 0$	completely ignorable and quaternary collation elements	Control codes and format characters Hebrew points Arabic tatweel ...
$X_1, X_2 = 0$; $X_3 \neq 0$	tertiary collation elements	*None in DUCET; could be in tailorings*
$X_1 = 0$; $X_2, X_3 \neq 0$	secondary collation elements	Most nonspacing marks Some letters and other combining marks
$X_1, X_2, X_3 \neq 0$	primary collation elements	
	lowest primary weight **0200**	*Special collation element* U+FFFE
	variable	Whitespace Punctuation General symbols but not Currency signs
	regular	Some general symbols Currency signs Numbers Letters of Latin, Greek, and other scripts...
	implicit (ideographs)	CJK Unified and similar Ideographs given implicit weights
	implicit (unassigned)	Unassigned and others given implicit weights
	trailing	*None in DUCET; could be in tailorings*
	reserved	*Special collation elements* U+FFFD U+FFFF

Note: The position of the boundary between variable and regular collation elements can be tailored.

6.4 Exceptional Grouping in DUCET

There are a number of exceptions in the grouping of characters in DUCET, where for various reasons characters are grouped in different categories. Examples are provided below for each type of exception.

1. If the NFKD decomposition of a character starts with certain punctuation characters, it is grouped with punctuation.
   - U+2474 ⑴ PARENTHESIZED DIGIT ONE
2. If the NFKD decomposition of a character starts with a character having General_Category=Number, then it is grouped with numbers.
   - U+3358 0㍘ IDEOGRAPHIC TELEGRAPH SYMBOL FOR HOUR ZERO
3. Many non-decimal numbers are grouped with general symbols.
   - U+2180 ↀ ROMAN NUMERAL ONE THOUSAND C D
4. Some numbers are grouped with the letters for particular scripts.
   - U+3280 ㊀ CIRCLED IDEOGRAPH ONE
5. Some letter modifiers are grouped with general symbols, others with their script.
   - U+3005 々 IDEOGRAPHIC ITERATION MARK
6. There are a few other exceptions, such as currency signs grouped with letters because of their decompositions.
   - U+20A8 ₨ RUPEE SIGN

6.5 Tailoring of DUCET

Note that the [CLDR] root collation tailors the DUCET. For details see *Root Collation* in [UTS35Collation].

For most languages, some degree of tailoring is required to match user expectations. For more information, see *Section 8, Tailoring*.

6.6 Default Values in DUCET

In the Default Unicode Collation Element Table and in typical tailorings, most unaccented letters differ in the primary weights, but have secondary weights (such as a_1) equal to MIN_2. The secondary collation elements will have secondary weights greater than MIN_2. Characters that are compatibility or case variants will have equal primary and secondary weights (for example, $a_1 = A_1$ and $a_2 = A_2$), but have different tertiary weights (for example, $a_3 < A_3$). The unmarked characters will have a_3 equal to MIN_3.

This use of secondary and tertiary weights does not guarantee that the meaning of a secondary or tertiary weight is uniform across tables. For example, in a tailoring a *capital A* and *katakana ta* could both have a tertiary weight of 3.

6.7 Tibetan and Well-Formedness of DUCET

Versions of the DUCET ~~is~~ prior to UCA 18.0 were *not entirely well-formed*. ~~It does~~ They did not include two Tibetan contraction mappings required for well-formedness condition 5:

```
0FB2 0F71 ; CE(0FB2) CE(0F71)
0FB3 0F71 ; CE(0FB3) CE(0F71)
```

In this formulation of contractions, the convention "CE(XXXX)" specifies the collation element in DUCET assigned to the code point "XXXX". "CE(XXXX XXXX)" specifies the collation element in DUCET assigned to the code point sequence "XXXX XXXX".

~~However,~~ Adding just these two contractions would disturb the default sort order for Tibetan. In order to also preserve the sort order for Tibetan, the following eight contractions ~~would have to be added as well~~ have also been added:

```
0FB2 0F71 0F72 ; CE(0FB2) CE(0F71 0F72)
0FB2 0F73      ; CE(0FB2) CE(0F71 0F72)
0FB2 0F71 0F74 ; CE(0FB2) CE(0F71 0F74)
0FB2 0F75      ; CE(0FB2) CE(0F71 0F74)

0FB3 0F71 0F72 ; CE(0FB3) CE(0F71 0F72)
0FB3 0F73      ; CE(0FB3) CE(0F71 0F72)
0FB3 0F71 0F74 ; CE(0FB3) CE(0F71 0F74)
0FB3 0F75      ; CE(0FB3) CE(0F71 0F74)
```

The requirement for these contractions to maintain well-formedness of the DUCET is the result of the complications imposed by the canonical decompositions of the two deprecated Tibetan characters, U+0F77 TIBETAN VOWEL SIGN VOCALIC RR and U+0F79 TIBETAN VOWEL SIGN VOCALIC LL. Most Tibetan text does not contain either of these deprecated characters.

~~The [CLDR] root collation adds all ten of these contractions.~~

6.8 Stability of DUCET

The contents of the DUCET will remain unchanged in any particular version of the UCA. However, the contents may change between successive versions of the UCA as new characters are added, or more information is obtained about existing characters.

Implementers should be aware that using different versions of the UCA or different versions of the Unicode Standard could result in different collation results of their data. There are numerous ways

collation data could vary across versions, for example:

1. Code points that were unassigned in a previous version of the Unicode Standard are now assigned in the current version, and will have a sorting semantic appropriate to the repertoire to which they belong. For example, the code points U+103D0..U+103DF were undefined in Unicode 3.1. Because they were assigned characters in Unicode 3.2, their sorting semantics and respective sorting weights changed as of that version.
2. Certain semantics of the Unicode standard could change between versions, such that code points are treated in a manner different than previous versions of the standard.
3. More information is gathered about a particular script, and the weight of a code point may need to be adjusted to provide a more linguistically accurate sort.

Any of these reasons could necessitate a change between versions with regards to collation weights for code points. It is therefore important that the implementers specify the version of the UCA, as well as the version of the Unicode Standard under which their data is sorted.

The policies which the UTC uses to guide decisions about the collation weight assignments made for newly assigned characters are enumerated in the UCA Default Table Criteria for New Characters. In addition, there are policies which constrain the timing and type of changes which are allowed for the DUCET table between versions of the UCA. Those policies are enumerated in Change Management for the Unicode Collation Algorithm.

7 Main Algorithm

The main algorithm has four steps:

1. Normalize each input string.
2. Produce an array of collation elements for each string.
3. Produce a sort key for each string from the arrays of collation elements.
4. Compare the two sort keys with a binary comparison operation.

The result of the binary comparison of the two sort keys is the ordering for the two original strings.

7.1 Normalize Each String

Step 1. Produce a normalized form of each input string, applying S1.1.

S1.1 Convert the string into Normalization Form D (see [UAX15]).

- Conformant implementations may skip this step in certain circumstances, *as long as they get the same results*. For techniques that may be useful in such an approach, see *Section 9.5, Avoiding Normalization*.

7.2 Produce Collation Element Arrays

Step 2. Construct a collation element array for each input string by sequencing through the (NFD) normalized string.

Figure 1 gives an example of the application of this step to one input string.

Figure 1. String to Collation Element Array

Normalized String	Collation Element Array
ca◌́b	[.0706.0020.0002], [.06D9.0020.0002], [.0000.0021.0002], [.06EE.0020.0002]

The construction of the collation element array is done by initializing an empty collation element array and then starting at the beginning of the normalized string, applying various steps and substeps as specified below, iterating until the end of the string is reached. Each loop through the steps and

substeps first seeks the longest contiguous match, and then any discontiguous match, and appends to the collation element array based upon the mapping for that match.

S2.1 Find the longest initial substring S at each point that has a match in the collation element table.

S2.1.1 If there are any non-starters following S, process each non-starter C.

S2.1.2 If C is an unblocked non-starter with respect to S, find if S + C has a match in the collation element table.

> **Note:** This condition is specific to non-starters, and is not precisely the same as the concept of blocking in normalization, since it is dealing with look ahead for a discontiguous match, rather than with normalization forms. Hangul jamos and other starters are only supported with contiguous matches .

S2.1.3 If there is a match, replace S by S + C, and remove C.

S2.2 Fetch the corresponding collation element(s) from the table if there is a match. If there is no match, synthesize a collation element as described in *Section 10.1, Derived Collation Elements*.

S2.3 Process collation elements according to the variable-weight setting, as described in *Section 4, Variable Weighting*.

S2.4 Append the collation element(s) to the collation element array.

S2.5 Proceed to the next point in the string (past S).

Steps S2.1 through S2.5 are iterated until the end of the string is reached.

> **Note:** The extra non-starter C needs to be considered in Step 2.1.1 because otherwise irrelevant characters could interfere with matches in the table. For example, suppose that the contraction <*a, combining_ring*> (= *å*) is ordered after *z*. If a string consists of the three characters <*a, combining_ring, combining_cedilla*>, then the normalized form is <*a, combining_cedilla, combining_ring*>, which separates the *a* from the *combining_ring*. Without considering the extra non-starter, this string would compare incorrectly as after *a* and not after *z*.
>
> If the desired ordering treats <*a, combining_cedilla*> as a contraction which should take precedence over <*a, combining_ring*>, then an additional mapping for the combination <*a, combining_ring, combining_cedilla*> can be introduced to produce this effect.
>
> For conformance to Unicode canonical equivalence, only unblocked non-starters are matched in Step 2.1.2. For example, <*a, combining_macron, combining_ring*> would compare as after *a-macron*, and not after *z*. Additional mappings can be added to customize behavior.
>
> Also note that the Algorithm employs two distinct contraction matching methods:
>
> - Step 2.1 “Find the longest initial substring S” is a contiguous, longest-match method. In particular, it must support matching of a contraction ABC even if there is not also a contraction AB. Thus, an implementation that incrementally matches a lengthening initial substring must be able to handle partial matches like for AB.
> - Steps 2.1.1 “process each non-starter C” and 2.1.2 “find if S + C has a match in the table”, where one or more intermediate non-starters may be skipped (making it discontiguous), extends a contraction match by one code point at a time to find the next match. In particular, if C is a non-starter and if the table had a mapping for ABC but not one for AB, then a discontiguous-contraction match on text ABMC (with M being a skippable non-starter) would never be found. Well-formedness condition 5 requires the presence of the prefix contraction AB.
> - In either case, the prefix contraction AB cannot be added to the table automatically because it would yield the wrong order for text ABD if there is a contraction BD.

7.3 Form Sort Keys

Step 3. Construct a sort key for each collation element array by successively appending all non-zero weights from the collation element array. Figure 2 gives an example of the application of this step to one collation element array.

Figure 2. Collation Element Array to Sort Key

Collation Element Array	Sort Key
`[.0706.0020.0002], [.06D9.0020.0002], [.0000.0021.0002], [.06EE.0020.0002]`	`0706 06D9 06EE 0000 0020 0020 0021 0020 0000 0002 0002 0002 0002`

Weights are appended from each level of the collation element array in turn, starting from level 1 and proceeding to level 3 (or to the last level, if more than 3 levels are supported). If an ordering is specified to be backward at a level, then for that level, the weights are appended in reverse order for that level.

An implementation may allow the maximum level to be set to a smaller level than the available levels in the collation element array. For example, if the maximum level is set to 2, then level 3 and higher weights are not appended to the sort key. Thus any differences at levels 3 and higher will be ignored, effectively ignoring any such differences in determination of the final result for the string comparison.

Here is a more detailed statement of the algorithm:

S3.1 For each weight level L in the collation element array from 1 to the maximum level,

S3.2 If L is not 1, append a *level separator*

Note: The level separator is zero (0000), which is guaranteed to be lower than any weight in the resulting sort key. This guarantees that when two strings of unequal length are compared, where the shorter string is a prefix of the longer string, the longer string is always sorted after the shorter—in the absence of special features like contractions. For example: "abc" < "abcX" where "X" can be any character(s).

S3.3 If the collation element table is forwards at level L,

S3.4 For each collation element CE in the array

S3.5 Append CE_L to the sort key if CE_L is non-zero.

S3.6 Else the collation table is backwards at level L, so

S3.7 Form a list of all the non-zero CE_L values.

S3.8 Reverse that list

S3.9 Append the CE_L values from that list to the sort key.

S3.10 If a semi-stable sort is required, then after all the level weights have been added, append a copy of the NFD version of the original string.
Since UCA 18.0, a minimal, unique weight must be emitted for U+FFFE. (For details see *Section 10.1.5, Reserved Weights*.)

This strength level is called the *identical level*, and this feature is called *semi-stability*. (See also *Appendix A, Deterministic Sorting*.)

7.4 Compare Sort Keys

Step 4. Compare the sort keys for each of the input strings, using a binary comparison. This means that:

- Level 3 differences are ignored if there are any Level 1 or 2 differences.
- Level 2 differences are ignored if there are any Level 1 differences.
- Level 1 differences are never ignored.

Figure 3. Comparison of Sort Keys

	String	Sort Key
1	cab	0706 06D9 06EE 0000 0020 0020 **0020** 0000 **0002** 0002 0002
2	Cab	0706 06D9 06EE 0000 0020 0020 **0020** 0000 **0008** 0002 0002
3	cáb	0706 06D9 06EE 0000 0020 0020 **0021** 0020 0000 0002 0002 0002 0002
4	dab	0712 06D9 06EE 0000 0020 0020 0020 0000 0002 0002 0002

In *Figure 3*, "cab" $<_3$ "Cab" $<_2$ "cáb" $<_1$ "dab". The differences that produce the ordering are shown by the **bold underlined** items:

- For strings 1 and 2, the first difference is in 0002 versus 0008 (Level 3).
- For strings 2 and 3, the first difference is in 0020 versus 0021 (Level 2).
- For strings 3 and 4, the first difference is in 0706 versus 0712 (Level 1).

7.5 Rationale for Well-Formed Collation Element Tables

While forming sort keys, zero weights are omitted. If collation elements were not well-formed according to conditions 1 and 2, the ordering of collation elements could be incorrectly reflected in the sort key. The following examples illustrate this.

Suppose well-formedness condition 1 were broken, and secondary weights of the Latin characters were zero (ignorable) and that (as normal) the primary weights of case-variants are equal: that is, $a_1 = A_1$. Then the following incorrect keys would be generated:

Order	String	Normalized	Sort Key
1	"áe"	a, acute, e	a_1 e_1 0000 $acute_2$ 0000 $\mathbf{a_3}$ $acute_3$ e_3...
2	"Aé"	A, e, acute	a_1 e_1 0000 $acute_2$ 0000 $\mathbf{A_3}$ $acute_3$ e_3...

Because the secondary weights for *a, A,* and *e* are lost in forming the sort key, the relative order of the acute is also lost, resulting in an incorrect ordering based solely on the case of *A* versus *a*. With well-formed weights, this does not happen, and the following correct ordering is obtained:

Order	String	Normalized	Sort Key
1	"Aé"	A, e, acute	a_1 e_1 0000 a_2 $\mathbf{e_2}$ $acute_2$ 0000 a_3 $acute_3$ e_3...
2	"áe"	a, acute, e	a_1 e_1 0000 a_2 $\mathbf{acute_2}$ e_2 0000 A_3 $acute_3$ e_3...

However, there are circumstances—typically in expansions—where higher-level weights in collation elements can be zeroed (resulting in ill-formed collation elements) without consequence (see *Section 9.2, Large Weight Values*). Implementations are free to do this as long as they produce the same result as with well-formed tables.

Suppose on the other hand, well-formedness condition 2 were broken. Let there be a tailoring of 'b' as a secondary difference from 'a' resulting in the following collation elements where the one for 'b' is ill-formed.

```
0300  ; [.0000.0035.0002] # (DUCET) COMBINING GRAVE ACCENT
0061  ; [.15EF.0020.0002] # (DUCET) LATIN SMALL LETTER A
0062  ; [.15EF.0040.0002] # (tailored) LATIN SMALL LETTER B
```

Then the following incorrect ordering would result: "aa" < "àa" < "ab" — The secondary difference on the *second* character (b) trumps the accent on the *first* character (à).

A correct tailoring would give 'b' a secondary weight lower than that of any secondary collation element, for example: (assuming the DUCET did not use secondary weight 0021 for any secondary collation element)

```
0300  ; [.0000.0035.0002] # (DUCET) COMBINING GRAVE ACCENT
0061  ; [.15EF.0020.0002] # (DUCET) LATIN SMALL LETTER A
0062  ; [.15EF.0021.0002] # (tailored) LATIN SMALL LETTER B
```

Then the following correct ordering would result: "aa" < "ab" < "àa"

8 Tailoring

Tailoring consists of any well-defined change in the Collation Element Table and/or any well-defined change in the behavior of the algorithm. Typically, a tailoring is expressed by means of a formal syntax which allows detailed manipulation of values in a Collation Element Table, with or without an additional collection of parametric settings which modify specific aspects of the behavior of the algorithm. A tailoring can be used to provide linguistically-accurate collation, if desired. Tailorings usually specify one or more of the following kinds of changes:

1. Reordering any character (or contraction) with respect to others in the default ordering. The reordering can represent a Level 1 difference, Level 2 difference, Level 3 difference, or identity (in levels 1 to 3). Because such reordering includes sequences, arbitrary multiple mappings can be specified.
2. Removing contractions, such as the Cyrillic contractions which are not necessary for the Russian language, and the Thai/Lao reordering contractions which are not necessary for string *search*.
3. Setting the secondary level to be backwards (for some French dictionary ordering traditions) or forwards (normal).
4. Set variable weighting options.
5. Customizing the exact list of variable collation elements.
6. Allow normalization to be turned off where input is already normalized.

For best interoperability, it is recommended that tailorings for particular locales (or languages) make use of the tables provided in the latest version of the Unicode Common Locale Data Repository [CLDR]. The CLDR collation tailorings support vetted orderings for many natural languages. They also include a meaningful ordering for emoji characters, as developed by the Unicode Emoji Subcommittee of the UTC.

For an example of a tailoring syntax, see *Section 8.2, Tailoring Example*.

8.1 Parametric Tailoring

Parametric tailoring, if supported, is specified using a set of attribute-value pairs that specify a particular kind of behavior relative to the UCA. The standard parameter names (attributes) and their possible values are listed in the table *Collation Settings* in [UTS35Collation].

The default values for collation parameters specified by the UCA algorithm may differ from the LDML defaults given in the LDML table *Collation Settings*. The table indicates both default values. For example, the UCA default for alternate handling is **shifted**, while the general default in LDML is **non-ignorable**. Also, defaults in CLDR data may vary by locale. For example, **normalization** is turned off in most CLDR locales (those that don't normally use multiple accents). The default for strength in UCA is **tertiary**; it can be changed for different locales in CLDR.

When a locale or language identifier is specified for tailoring of the UCA, the identifier uses the syntax from [UTS35], *Section 3, Unicode Language and Locale Identifiers*. Unless otherwise specified,

tailoring by locale uses the tables from the Unicode Common Locale Data Repository [CLDR].

8.2 Tailoring Example

Unicode [CLDR] provides a powerful tailoring syntax in [UTS35Collation], as well as tailoring data for many locales. The CLDR tailorings are based on the CLDR root collation, which itself is a tailored version of the DUCET table (see *Root Collation* in [UTS35Collation]). The CLDR collation tailoring syntax is a subset of the ICU syntax. Some of the most common syntax elements are shown in *Table 14*. A simpler version of this syntax is also used in Java, although at the time of this writing, Java does not implement the UCA.

Table 14. ICU Tailoring Syntax

Syntax	Description
& y < x	Make x primary-greater than y
& y << x	Make x secondary-greater than y
& y <<< x	Make x tertiary-greater than y
& y = x	Make x equal to y

Either *x* or *y* in this syntax can represent more than one character, to handle contractions and expansions.

Entries for tailoring can be abbreviated in a number of ways:

- They do not need to be separated by newlines.
- Characters can be specified directly, instead of using their hexadecimal Unicode values.
- In rules of the form "x < y & y < z", "& y" can be omitted, leaving just "x < y < z".

These abbreviations can be applied successively, so the examples shown in *Table 15* are equivalent in ordering.

Table 15. Equivalent Tailorings

ICU Syntax	DUCET Syntax

a <<< A << à <<< À < b <<< B	`0061 ; [.0001.0001.0001] % a`
	`0040 ; [.0001.0001.0002] % A`
	`00E0 ; [.0001.0002.0001] % à`
	`00C0 ; [.0001.0002.0002] % À`
	`0042 ; [.0002.0001.0001] % b`
	`0062 ; [.0002.0001.0002] % B`

The syntax has many other capabilities: for more information, see [UTS35Collation] and [ICUCollator].

8.3 Use of Combining Grapheme Joiner

The Unicode Collation Algorithm involves the normalization of Unicode text strings before collation weighting. U+034F COMBINING GRAPHEME JOINER (CGJ) is ordinarily ignored in collation key weighting in the UCA, but it can be used to block the reordering of combining marks in a string as described in [Unicode]. In that case, its effect can be to invert the order of secondary key weights associated with those combining marks. Because of this, the two strings would have distinct keys, making it possible to treat them distinctly in searching and sorting without having to further tailor either the combining grapheme joiner or the combining marks.

The CGJ can also be used to prevent the formation of contractions in the Unicode Collation Algorithm. Thus, for example, while *ch* is sorted as a single unit in a tailored Slovak collation, the sequence <*c*, CGJ, *h*> will sort as a *c* followed by an *h*. This can also be used in German, for example, to force *ü* to

be sorted as *u* + *umlaut* (thus *u* $<_2$ *ü*), even where a dictionary sort is being used (which would sort *ue* $<_3$ *ü)*. This happens without having to further tailor either the combining grapheme joiner or the sequence.

> **Note:** As in a few other cases in the Unicode Standard, the name of the CGJ can be misleading —the usage above is in some sense the inverse of "joining".

Sequences of characters which include the combining grapheme joiner or other completely ignorable characters may also be given tailored weights. Thus the sequence <c, CGJ, h> could be weighted completely differently from either the contraction "ch" or the sequence "c" followed by "h" without the contraction. However, this application of CGJ is not recommended, because it would produce effects much different than the normal usage above, which is to simply interrupt contractions.

8.4 Preprocessing

In addition to tailoring, some implementations may choose to preprocess the text for special purposes. Once such preprocessing is done, the standard algorithm can be applied.

Examples include:

- mapping "McBeth" to "MacBeth"
- mapping "St." to "Street" or "Saint", depending on the context
- dropping articles, such as "a" or "the"
- using extra information, such as pronunciation data for Han characters

Such preprocessing is outside of the scope of this document.

9 Implementation Notes

As noted above for efficiency, implementations may vary from this logical algorithm as long as they produce the same result. The following items discuss various techniques that can be used for reducing sort key length, reducing table sizes, customizing for additional environments, searching, and other topics.

9.1 Reducing Sort Key Lengths

The following discuss methods of reducing sort key lengths. If these methods are applied to all of the sort keys produced by an implementation, they can result in significantly shorter and more efficient sort keys while retaining the same ordering.

9.1.1 Eliminating Level Separators

Level separators are not needed between two levels in the sort key, if the weights are properly chosen. For example, if all L3 weights are less than all L2 weights, then no level separator is needed between them. If there is a fourth level, then the separator before it needs to be retained.

The following example shows a sort key with these level separators removed.

String	Technique(s) Applied	Sort Key
càb	none	`0706 06D9 06EE 0000 0020 0020 0021 0020 0000 0002 0002 0002 0002`
càb	1	`0706 06D9 06EE 0020 0020 0021 0020 0002 0002 0002 0002`

While this technique is relatively easy to implement, it can interfere with other compression methods.

9.1.2 L2/L3 in 8 Bits

The L2 and L3 weights commonly are small values. Where that condition occurs for all possible values, they can then be represented as single 8-bit quantities.

The following example modifies the first example with both these changes (and grouping by bytes). Note that the separator has to remain after the primary weight when combining these techniques. If any separators are retained (such as before the fourth level), they need to have the same width as the previous level.

String	Technique(s) Applied	Sort Key
càb	none	07 06 06 D9 06 EE 00 00 00 20 00 20 00 21 00 20 00 00 00 02 00 02 00 02 00 02
càb	1, 2	07 06 06 D9 06 EE 00 00 20 20 21 20 02 02 02 02

9.1.3 Machine Words

The sort key can be represented as an array of different quantities depending on the machine architecture. For example, comparisons as arrays of unsigned 32-bit quantities may be much faster on some machines. When using arrays of unsigned 32-bit quantities, the original sort key is to be padded with trailing (not leading) zeros as necessary.

String	Technique(s) Applied	Sort Key
càb	1, 2	07 06 06 D9 06 EE 00 00 20 20 21 20 02 02 02 02
càb	1, 2, 3	070606D9 06EE0000 20202120 02020202

9.1.4 Run-Length Compression

Generally sort keys do not differ much in the secondary or tertiary weights, which tends to result in keys with a lot of repetition. This also occurs with quaternary weights generated with the shifted parameter. By the structure of the collation element tables, there are also many weights that are never assigned at a given level in the sort key. One can take advantage of these regularities in these sequences to compact the length—while retaining the same sort sequence—by using the following technique. (There are other techniques that can also be used.)

This is a logical statement of the process; the actual implementation can be much faster and performed as the sort key is being generated.

- For each level ***n,*** find the most common value COMMON produced at that level by the collation element table for typical strings. For example, for the Default Unicode Collation Element Table, this is:
  - 0020 for the secondaries (corresponding to unaccented characters)
  - 0002 for tertiaries (corresponding to lowercase or unmarked letters)
  - FFFF for quaternaries (corresponding to non-ignorables with the shifted parameter)
- Reassign the weights in the collation element table at level ***n*** to create a gap of size GAP above COMMON. Typically for secondaries or tertiaries this is done after the values have been reduced to a byte range by the above methods. Here is a mapping that moves weights up or down to create a gap in a byte range.
  ```
  w → w + 01 - MIN, for MIN <= w < COMMON
  w → w + FF - MAX, for COMMON < w <= MAX
  ```
- At this point, weights go from 1 to MINTOP, and from MAXBOTTOM to MAX. These new unassigned values are used to run-length encode sequences of COMMON weights.
- When generating a sort key, look for maximal sequences of **m** COMMON values in a row. Let W be the weight right after the sequence.
  - If W < COMMON (or there is no W), replace the sequence by a synthetic low weight equal to (MINTOP + m).
  - If W > COMMON, replace the sequence by a synthetic high weight equal to (MAXBOTTOM - m).

In the example shown in *Figure 4*, the low weights are 01, 02; the high weights are FE, FF; and the common weight is 77.

Figure 4. Run-Length Compression

Original Weights	Compressed Weights
01	01
02	02
77 01	03 01
77 02	03 02
77 77 01	04 01
77 77 02	04 02
77 77 77 01	05 01
77 77 77 02	05 02
...	...
77 77 77 FE	FB FE
77 77 77 FF	FB FF
77 77 FE	FC FE
77 77 FF	FC FF
77 FE	FD FE
77 FF	FD FF
FE	FE
FF	FF

- The last step is a bit too simple, because the synthetic weights must not collide with other values having long strings of COMMON weights. This is done by using a sequence of synthetic weights, absorbing as much length into each one as possible. A value BOUND is defined between MINTOP and MAXBOTTOM. The exact value for BOUND can be chosen based on the expected frequency of synthetic low weights versus high weights for the particular collation element table.
  - If a synthetic low weight would not be less than BOUND, use a sequence of low weights of the form (BOUND-1)..(BOUND-1)(MINTOP + remainder) to express the length of the sequence.
  - Similarly, if a synthetic high weight would be less than BOUND, use a sequence of high weights of the form (BOUND)..(BOUND)(MAXBOTTOM - remainder).

This process results in keys that are never longer than the original, are generally much shorter, and result in the same comparisons.

9.2 Large Weight Values

If an implementation uses short integers (for example, bytes or 16-bit words) to store weights, then some weights require sequences of those short integers. The lengths of the sequences can vary, using short sequences for the weights of common characters and longer sequences for the weights of rare characters.

For example, suppose that 50,000 supplementary private-use characters are used in an implementation which uses 16-bit words for primary weights, and that these are to be sorted after a character whose primary weight is `x`. In such cases, the second ~~CE ("continuation")~~ collation element (CE), also referred to as the "continuation", does not have to be well formed.

Simply assign ~~them~~ all of these characters dual collation elements of the following form:

```
[.(X+1).zzzz.wwww], [.yyyy.0000.0000]
```

If there is an element with the primary weight `(X+1)`, then it also needs to be converted into a dual collation element.

The private-use characters will then sort properly with respect to each other and the rest of the characters. The second collation element of this dual collation element pair is one of the instances in which ill-formed collation elements are allowed. The first collation element of each of these pairs is well-formed, and the first element only occurs in combination with them. (It is not permissible for any

weight's sequence of units to be an initial sub-sequence of another weight's sequence of units.) In this way, ordering is preserved with respect to other, non-paired collation elements.

The continuation technique appears in the DUCET, for all implicit primary weights:

```
2F00 ; [.FB40.0020.0004][.CE00.0000.0000] # KANGXI RADICAL ONE
```

As an example for level 2, suppose that 2,000 L2 weights are to be stored using byte values. Most of the weights require at least two bytes. One possibility would be to use 8 lead byte values for them, storing pairs of CEs of the form [.yyyy.zz.ww][.0000.nn.00]. This would leave 248 byte values (minus byte value zero, and some number of byte values for level separators and run-length compression) available as single-byte L2 weights of as many high-frequency characters, storing single CEs of the form [.yyyy.zz.ww].

Note that appending and comparing weights in a backwards level needs to handle the most significant bits of a weight first, even if the bits of that weight are spread out in the data structure over multiple collation elements.

9.3 Reducing Table Sizes

The data tables required for collation of the entire Unicode repertoire can be quite sizable. This section discusses ways to significantly reduce the table size in memory. These recommendations have very important implications for implementations.

9.3.1 Contiguous Weight Ranges

Whenever collation elements have different primary weights, the ordering of their secondary weights is immaterial. Thus all of the secondaries that share a single primary can be renumbered to a contiguous range without affecting the resulting order. The same technique can be applied to tertiary weights.

9.3.2 Leveraging Unicode Tables

Because all canonically decomposable characters are decomposed in Step 1.1, no collation elements need to be supplied for them. The DUCET has over 2,000 of these, but they can all be dropped with no change to the ordering (it does omit the 11,172 Hangul syllables).

The collation elements for the Han characters (unless tailored) are algorithmically derived; no collation elements need to be stored for them either.

This means that only a small fraction of the total number of Unicode characters need to have an explicit collation element. This can cut down the memory storage considerably.

In addition, most characters with compatibility decompositions can have collation elements computed at runtime to save space, duplicating the work that was done to compute the Default Unicode Collation Element Table. This can provide important savings in memory space. The process works as follows.

1. Derive the compatibility decomposition. For example,

```
2475 PARENTHESIZED DIGIT TWO => 0028, 0032, 0029
```

2. Look up the collation, discarding completely ignorables. For example,

```
0028 [*023D.0020.0002] % LEFT PARENTHESIS
0032 [.06C8.0020.0002] % DIGIT TWO
0029 [*023E.0020.0002] % RIGHT PARENTHESIS
```

3. Set the L3 values according to the table in *Section 10.2, Tertiary Weight Table*. For example,

```
0028 [*023D.0020.0004] % LEFT PARENTHESIS
0032 [.06C8.0020.0004] % DIGIT TWO
0029 [*023E.0020.0004] % RIGHT PARENTHESIS
```

4. Concatenate the result to produce the sequence of collation elements that the character maps to. For example,

```
2475 [*023D.0020.0004] [.06C8.0020.0004] [*023E.0020.0004]
```

Some characters cannot be computed in this way. They must be filtered out of the default table and given specific values. For example, the *long s* has a secondary difference, not a tertiary.

```
0073 [.17D9.0020.0002] # LATIN SMALL LETTER S
017F [.17D9.0020.0004][.0000.013A.0004] # LATIN SMALL LETTER LONG S
```

9.3.3 Reducing the Repertoire

If characters are not fully supported by an implementation, then their code points can be treated as if they were unassigned. This allows them to be algorithmically constructed from code point values instead of including them in a table. This can significantly reduce the size of the required tables. See *Section 10.1, Derived Collation Elements* for more information.

9.3.4 Memory Table Size

Applying the above techniques, an implementation can thus safely pack all of the data for a collation element into a single 32-bit quantity: 16 for the primary, 8 for the secondary and 8 for the tertiary. Then applying techniques such as the Two-Stage table approach described in *"Multistage Tables"* in *Section 5.1, Transcoding to Other Standards* of [Unicode], the mapping table from characters to collation elements can be both fast and small.

9.4 Avoiding Zero Bytes

If the resulting sort key is to be a C-string, then zero bytes must be avoided. This can be done by:

- using the value 0101_{16} for the level separator instead of 0000
- preprocessing the weight values to avoid zero bytes, for example by remapping 16-bit weights as follows (and larger weight values in analogous ways):

$$x \rightarrow 0101_{16} + (x / 255)*256 + (x \% 255)$$

Where the values are limited to 8-bit quantities (as discussed above), zero bytes are even more easily avoided by just using 01 as the level separator (where one is necessary), and mapping weights by:

$$x \rightarrow 01 + x$$

9.5 Avoiding Normalization

Conformant implementations must get the same results as the Unicode Collation Algorithm, but such implementations may use different techniques to get those results, usually with the goal of achieving better performance. For example, an implementation may be able to avoid normalizing most, if not all, of an input string in Step 1 of the algorithm.

In a straightforward implementation of the algorithm, canonically decomposable characters do not require mappings to collation elements because S1.1 decomposes them, so they do not occur in any of the following algorithm steps and thus are irrelevant for the collation elements lookup. For example, there need not be a mapping for “ü” because it is always decomposed to the sequence “u + ◌̈”.

In an optimized implementation, a canonically decomposable character like “ü” may map directly to the sequence of collation elements for the decomposition (“ü” → CE(u)CE(◌̈), unless there is a contraction defined for that sequence). In other words, using the convention “CE(c)” to represent the collation element assigned to the character “c”:

“ü” → CE(u) CE(◌̈)

For most input strings, these mappings can be used directly for correct results, rather than first having to normalize the text.

While such an approach can lead to significantly improved performance, there are various issues that need to be handled, including but not limited to the following:

1. Typically, the easiest way to manage the data is to add mappings for each of the canonically equivalent strings, the so-called “canonical closure”. Thus, each of {ǭ, ǫ +¯, ō +˛, o +¯+˛, o +˛+ ¯} can map to the same collation elements.
2. These collation elements must be in the same order as if the characters were decomposed using Normalization Form D.
3. The easiest approach is to detected sequences that are in the format known as “Fast C or D form” (FCD: see [UTN5]), and to directly look up collation elements for characters in such FCD sequences, without normalizing them.
4. In any difficult cases, such as if a sequence is not in FCD form, or when there are contractions that cross sequence boundaries, the algorithm can fall back to doing a full NFD normalization.

9.6 Case Comparisons

In some languages, it is common to sort lowercase before uppercase; in other languages this is reversed. Often this is more dependent on the individual concerned, and is not standard across a single language. It is strongly recommended that implementations provide parameterization that allows uppercase to be sorted before lowercase, and provides information as to the standard (if any) for particular countries. For more information, see *Case Parameters* in [UTS35Collation].

9.7 Incremental Comparison

For one-off comparison of strings, actual implementations of the UCA typically do not construct complete sort keys for strings. Instead, an efficient implementation simply processes collation weights until the first point at which the outcome of the comparison is determined. This technique is called incremental comparison.

For example, to compare the strings "azzzzzz" and "byyyyyy" there is generally no need to build sort key values for the six "z" characters in the first string and the six "y" characters in the second string, because the result of the comparison is already apparent after the first character has been weighted.

Incremental comparison is tricky to implement, however, as care needs to be taken to handle all potential expansion and contraction mappings correctly. The *conformance* requirement for the UCA is simply that the correct comparison be calculated, *as if* the full sort keys had been constructed and compared. Collation elements can be incrementally generated as needed from two strings, and compared with an algorithm that produces the same results as comparison of the two sort keys would have. The choice of which algorithm to use depends on the number of comparisons between the same strings.

- Generally, incremental comparison is *more* efficient than producing full sort keys if strings are only to be compared once and if they are typically dissimilar, because differences are caught in the first few characters without having to process both strings to the end.
- Generally, incremental comparison is *less* efficient than producing full sort keys if items are to be compared multiple times.

It is very tricky to produce an incremental comparison that produces correct results. Some attempted implementations of incremental comparison for the UCA have not even been transitive! Be sure to thoroughly test any code for incremental comparison.

9.8 Catching Mismatches

Sort keys from two different tailored collations cannot be compared, because the weights may end up being rearranged arbitrarily. To catch this case, implementations can produce a hash value from the collation data, and prepend it to the sort key. Except in extremely rare circumstances, this will distinguish the sort keys. The implementation then has the opportunity to signal an error.

9.9 Handling Collation Graphemes

A collation ordering determines a *collation grapheme cluster* (also known as a collation grapheme or collation character), which is a sequence of characters that is treated as a primary unit by the ordering. For example, *ch* is a collation grapheme for a Slovak ordering. These are generally contractions, but may include additional ignorable characters.

Roughly speaking, a collation grapheme cluster is the longest substring whose corresponding collation elements start with a non-zero primary weight, and contain as few other collation elements with non-zero primary weights as possible. In some cases, collation grapheme clusters may be *degenerate*: they may have collation elements that do not contain a non-zero weight, or they may have no non-zero weights at all.

For example, consider a collation for language in which "ch" is treated as a contraction, and "à" as an expansion. The expansion for à contains collation weights corresponding to *combining-grave* + "a" (but in an unusual order). In that case, the string <`ab`ch`à> would have the following clusters:

- *combining-grave* (a degenerate case),
- "a"
- "b`"
- "ch`"
- "à" (also a degenerate case, starting with a zero primary weight).

To find the collation grapheme cluster boundaries in a string, the following algorithm can be used:

1. Set **position** to be equal to 0, and set a boundary there.
2. If **position** is at the end of the string, set a boundary there, and return.
3. Set **startPosition** = **position**.
4. Fetch the next collation element(s) mapped to by the character(s) at **position**, setting **position** to the end of the character(s) mapped.
    1. This fetch must collect collation elements, including discontiguous contractions, until no characters are skipped.
    2. It cannot rewrite the input string for S2.1.3 (that would invalidate the indexes).
5. If the collation element(s) contain a collation element with a non-zero primary weight, set a boundary at **startPosition**.
6. Loop to step 2.

For information on the use of collation graphemes, see [UTS18].

9.10 Sorting Plain Text Data Files

When reading data from plain text files for sorting and other processing, characters that serve as data field separators are stripped before comparing strings. For example, line separators (carriage return and/or line feed) are removed from lines of text, and commas (and often leading and trailing white space) are removed from fields in CSV (comma separated values) files.

This preprocessing is equivalent to reading records from a spreadsheet or database. When importing such a text data file into a spreadsheet or database, the line separators are always removed.

As a result, the collation element mappings for separator characters are unused and are immaterial for sorting structured data.

10 Weight Derivation

This section describes the generation of the Default Unicode Collation Element Table (DUCET), and the assignment of weights to code points that are not explicitly mentioned in that table. The assignment of weights uses information derived from the Unicode Character Database [UAX44].

10.1 Derived Collation Elements

Siniform ideographs — most notably modern CJK (Han) ideographs — and Hangul syllables are not explicitly mentioned in the default table. Ideographs are mapped to collation elements that are derived from their Unicode code point value as described in *Section 10.1.3, Implicit Weights*. For a discussion of derived collation elements for Hangul syllables and other issues related to the collation of Korean, see *Section 10.1.6, Hangul Collation*.

10.1.1 Handling Ill-Formed Code Unit Sequences

Unicode strings sometimes contain ill-formed code unit sequences. Such ill-formed sequences must not be interpreted as valid Unicode characters. See *Section 3.2, Conformance Requirements* in [Unicode]. For example, expressed in UTF-32, a Unicode string might contain a 32-bit value corresponding to a surrogate code point (General_Category Cs) or an out-of-range value (< 0 or > 10FFFF), or a UTF-8 string might contain misconverted byte values that cannot be interpreted. Implementations of the Unicode Collation Algorithm may choose to treat such ill-formed code unit sequences as error conditions and respond appropriately, such as by throwing an exception.

An implementation of the Unicode Collation Algorithm may also choose not to treat ill-formed sequences as an error condition, but instead to give them explicit weights. This strategy provides for determinant comparison results for Unicode strings, even when they contain ill-formed sequences. However, to avoid security issues when using this strategy, ill-formed code sequences should not be given an ignorable or variable primary weight.

There are two recommended approaches, based on how these ill-formed sequences are typically handled by character set converters.

- The first approach is to weight each maximal ill-formed subsequence as if it were U+FFFD REPLACEMENT CHARACTER. (For more information about maximal ill-formed subsequences, see *Section 3.9, Unicode Encoding Forms* in [Unicode].)
- A second approach is to generate an implicit weight for any surrogate code point as if it were an unassigned code point, using the method of *Section 10.1.3, Implicit Weights*.

10.1.2 Unassigned and Other Code Points

Each unassigned code point and each other code point that is not explicitly mentioned in the table is mapped to a sequence of two collation elements as described in *Section 10.1.3, Implicit Weights*.

10.1.3 Implicit Weights

Code points that do not have explicit mappings in the DUCET are mapped to collation elements with implicit primary weights that sort between regular explicit weights and trailing weights. Within each set represented by a row of the following table, the code points are sorted in code point order.

> **Note:** The following method yields implicit weights in the form of pairs of 16-bit words, appropriate for UCA+DUCET. As described in *Section 9.2, Large Weight Values*, an implementation may use longer or shorter integers. Such an implementation would need to modify the generation of implicit weights appropriately while yielding the same relative order. Similarly, an implementation might use very different actual weights than the DUCET, and the “base” weights would have to be adjusted as well.

For each code point CP that does not have an explicit collation element in the DUCET, find the matching row in the following table and compute the two 16-bit primary weight units AAAA and BBBB. BBBB will always have bit 15 set, to ensure that BBBB is never zero. CP maps to a pair of collation elements of this form:

[.AAAA.0020.0002][.BBBB.0000.0000]

The **allkeys.txt** file specifies the relevant parameters for siniform ideographic scripts (but not for Han ideographs) in @implicitweights lines, see *Section 12.1, Allkeys File Format*.

If a fourth or higher weights are used, then the same pattern is followed for those weights. They are set to a non-zero value in the first collation element and zero in the second. (Because all distinct code points have a different **AAAA/BBBB** combination, the exact non-zero value does not matter.)

Decomposable characters are excluded because they are otherwise handled in the UCA.

Table 16. Computing Implicit Weights

Type	Subtype	Code Points (CP)	AAAA	BBBB
Siniform ideographic scripts	Tangut	Assigned code points in Block=Tangut OR Block=Tangut_Supplement	0xFB00	(CP - 0x17000) | 0x8000
		Assigned code points in Block=Tangut_Components OR Block=Tangut_Components_Supplement	0xFB01	(CP - 0x18800) | 0x8000
	Nushu	Assigned code points in Block=Nushu	0xFB02	(CP - 0x1B170) | 0x8000
	Khitan Small Script	Assigned code points in Block=Khitan_Small_Script	0xFB03	(CP - 0x18B00) | 0x8000
	Jurchen	Assigned code points in Block=Jurchen OR Block=Jurchen_Radicals	0xFB04	(CP - 0x18E00) | 0x8000
	Seal	Assigned code points in Block=Seal	0xFB05	(CP - 0x3D000) | 0x8000
Han	Core Han Unified Ideographs	Unified_Ideograph=True **AND** ((Block=CJK_Unified_Ideograph) OR (Block=CJK_Compatibility_Ideographs)) In regex notation: [\p{unified_ideograph}&[\p{Block=CJK_Unified_Ideographs} \p{Block=CJK_Compatibility_Ideographs}]]	0xFB40 + (CP >> 15) (0xFB40..0xFB41)	(CP & 0x7FFF) | 0x8000
	All other Han Unified Ideographs	Unified_Ideograph=True **AND NOT** ((Block=CJK_Unified_Ideograph) OR (Block=CJK_Compatibility_Ideographs)) In regex notation: [\p{unified ideograph}-[\p{Block=CJK_Unified_Ideographs} \p{Block=CJK_Compatibility_Ideographs}]]	0xFB80 + (CP >> 15) (0xFB80, 0xFB84..0xFB85)	
Unassigned		Any other code point	0xFBC0 + (CP >> 15) (0xFBC0..0xFBE1)	

Note: The Common Template Table (CTT) defined in [ISO14651] makes use of the weight derivation defined in Table 16 for implicit weights. However, instead of using precise 16-bit word values, as specified above for DUCET, the CTT uses *symbolic* values. When calculating implicit weights for the CTT, one takes the hexadecimal value in the "AAAA" column of Table 16, prefixed with the letter "R", and the hexadecimal value in the "BBBB" column of Table 16, prefixed with the letter "T", to create a pair of weight symbols. Thus, for example, using Table 16 to weight the CJK Unified ideograph U+4E00, for UCA one calculates a pair of collation elements as follows:

```
[.FB40.0020.0002][.CE00.0000.0000]
```

In contrast, the corresponding entry that would be added to a tailoring of the CTT would be:

```
<U4E00> "<RFB40><TCE00>";<BASE>;<MIN>;<SFFFF> % CJK UNIFIED IDEOGRAPH-4E00
```

10.1.4 Trailing Weights

In the DUCET, the primary weights from FC00 to FFFC (near the top of the range of primary weights) are available for use as trailing weights.

In many writing systems, the convention for collation is to order by syllables (or other units similar to syllables). In most cases a good approximation to syllabic ordering can be obtained in the UCA by weighting initial elements of syllables in the appropriate primary order, followed by medial elements (such as vowels), followed by final elements, if any. The default weights for the UCA in the DUCET are assigned according to this general principle for many scripts. This approach handles syllables within a given script fairly well, but unexpected results can occur when syllables of different lengths are adjacent to characters with higher primary weights, as illustrated in the following example:

Case 1

1	{G}{A}
2	{G}{A}{K}

Case 2

2	{G}{A}{K}事
1	{G}{A}事

In this example, the symbols {G}, {A}, and {K} represent letters in a script where syllables (or other sequences of characters) are sorted as units. By proper choice of weights for the individual letters, the syllables can be ordered correctly. However, the weights of the following characters may cause syllables of different lengths to change order. Thus {G}{A}{K} comes after {G}{A} in Case 1, but in Case 2, it comes *before*. That is, the order of these two syllables would be reversed when each is followed by a CJK ideograph, with a high primary weight: in this case, U+4E8B (事).

This unexpected behavior can be avoided by using trailing weights to tailor the non-initial letters in such syllables. The trailing weights, by design, have higher values than the primary weights for characters in all scripts, including the implicit weights used for CJK ideographs. Thus in the example, if {K} is tailored with a trailing weight, it would have a higher weight than any CJK ideograph, and as a result, the relative order of the two syllables {G}{A}{K} and {G}{A} would not be affected by the presence of a CJK ideograph following either syllable.

10.1.5 Reserved Weights

In the DUCET, the primary weights from FFFD to FFFF (at the very top of the range of primary weights) and the lowest primary weight 0200 are reserved for special collation elements.

~~For example, i~~In DUCET, **U+FFFD REPLACEMENT CHARACTER** maps to a collation element with the fixed primary weight of **FFFD**, thus ensuring that it is not a variable collation element. This means that implementations using U+FFFD as a replacement for ill-formed code unit sequences will not have those replacement characters ignored in collation.

The **noncharacter U+FFFE** maps to a collation element with the fixed lowest primary weight of **0200**.

- Its primary weight is not “variable”: U+FFFE must not become ignorable in alternate handling / variable weighting.
- On the identical level, a minimal, unique weight must be emitted for U+FFFE as well. This allows for Merging Sort Keys within code point space.
- For example, when sorting names in a database, a sortable string can be formed with *last_name* + '\uFFFE' + *first_name*. These strings would sort properly, without ever comparing the last part of a last name with the first part of another first name.
- For backwards secondary level sorting, text *segments* separated by U+FFFE are processed in forward segment order, and *within* each segment the secondary weights are compared backwards. This is so that such combined strings are processed consistently with merging their sort keys (for example, by concatenating them level by level with a low separator).
- **Note:** With unique, low weights on all levels it is possible to achieve `sortkey(str1 + "\uFFFE" + str2) == mergeSortkeys(sortkey(str1), sortkey(str2))`. When that is not necessary, then code can be a little simpler (no special handling for U+FFFE except for backwards-secondary), sort keys can be a little shorter (when using compressible common non-primary weights for U+FFFE), and another low weight can be used in tailorings.

The **noncharacter U+FFFF** maps to a collation element with the fixed highest primary weight of **FFFF**. The assignment of highest primary weight for this noncharacter makes possible the reliable specification of a range of strings, for example “Sch” ≤ X ≤ “Sch\uFFFF”, to include all strings starting with “Sch” or equivalent.

10.1.6 Hangul Collation

The Hangul script for Korean is in a rather unique position, because of its large number of precomposed syllable characters, and because those precomposed characters are the normal (NFC) form of interchanged text. For Hangul syllables to sort correctly, either the DUCET table must be tailored or both the UCA algorithm and the table must be tailored. The essential problem results from the fact that Hangul syllables can also be represented with a sequence of conjoining jamo characters and because syllables represented that way may be of different lengths, with or without a trailing consonant jamo. That introduces the trailing weights problem, as discussed in *Section 10.1.4, Trailing Weights*. This section describes several approaches which implementations may take for tailoring to deal with the trailing weights problem for Hangul.

> **Note:** The Unicode Technical Committee recognizes that it would be preferable if a single "best" approach could be standardized and incorporated as part of the specification of the UCA algorithm and the DUCET table. However, picking a solution requires working out a common approach to the problem with ISO SC2, which takes considerable time. In the meantime, implementations can choose among the various approaches discussed here, when faced with the need to order Korean data correctly.

The following discussion makes use of definitions and abbreviations from *Section 3.12, Conjoining Jamo Behavior* in [Unicode]. In addition, a special symbol (Ⓣ) is introduced to indicate a terminator weight. For convenience in reference, these conventions are summarized here:

Description	Abbr	Weight
Leading consonant	L	W_L
Vowel	V	W_V
Trailing consonant	T	W_T
Terminator weight	-	Ⓣ

Simple Method

The specification of the Unicode Collation Algorithm requires that Hangul syllables be decomposed. However, if the weight table is tailored so that the primary weights for Hangul jamo are adjusted, then

the Hangul syllables can be left as single code points and be treated in much the same way as CJK ideographs. The adjustment is specified as follows:

1. Tailor each L to have a primary weight corresponding to the first Hangul syllable starting with that jamo.
2. Tailor all Vs and Ts to be ignorable at the primary level.

The net effect of such a tailoring is to provide a Hangul collation which is approximately equivalent to one of the more complex methods specified below. This may be sufficient in environments where individual jamo are not generally expected.

Three more complex and complete methods are spelled out below. First the nature of the tailoring is described. Then each method is exemplified, showing the implications for the relative weighting of jamo and illustrating how each method produces correct results.

Each of these three methods can correctly represent the ordering of all Hangul syllables, both for modern Korean and for Old Korean. However, there are implementation trade-offs between them. These trade-offs can have a significant impact on the acceptability of a particular implementation. For example, substantially longer sort keys will cause serious performance degradations and database index bloat. Some of the pros and cons of each method are mentioned in the discussion of each example. Note that if the repertoire of supported Hangul syllables is limited to those required for modern Korean (those of the form LV or LVT), then each of these methods becomes simpler to implement.

Data Method

1. Tailor the Vs and Ts to be Trailing Weights, with the ordering T < V
2. Tailor each sequence of multiple L's that occurs in the repertoire as a contraction, with an independent primary weight after any prefix's weight.

For example, if L_1 has a primary weight of 555, and L_2 has a primary weight of 559, then the sequence L_1L_2 would be treated as a contraction and be given a primary weight chosen from the range 556 to 558.

Terminator Method

1. Add an internal terminator primary weight (Ⓣ).
2. Tailor all jamo so that Ⓣ < T < V < L
3. Algorithmically add the terminator primary weight (Ⓣ) to the end of every standard Korean syllable block.

The details of the algorithm for parsing Hangul data into standard Korean syllable blocks can be found in *Section 8, Hangul Syllable Boundary Determination* of [UAX29]

Interleaving Method

The interleaving method requires tailoring both the DUCET table and the way the algorithm handles Korean text.

Generate a tailored weight table by assigned an explicit primary weight to each precomposed Hangul syllable character, with a 1-weight gap between each one. (See *Section 10.2, Large Weight Values*.)

Separately define a small, internal table of jamo weights. This internal table of jamo weights is separate from the tailored weight table, and is only used when processing standard Korean syllable blocks. Define this table as follows:

1. Give each jamo a 1-byte weight.
2. Add an internal terminator 1-byte weight (Ⓣ).
3. Assign these values so that: Ⓣ < T < V < L.

When processing a string to assign collation weights, whenever a substring of jamo and/or precomposed Hangul syllables in encountered, break it into standard Korean syllable blocks. For each syllable identified, assign a weight as follows:

1. If a syllable is canonically equivalent to one of the precomposed Hangul syllable characters, then assign the weight based on the tailored weight table.
2. If a syllable is not canonically equivalent to one of the precomposed Hangul syllable characters, then assign a weight sequence by the following steps:
   a. Find the greatest precomposed Hangul syllable that the parsed standard Korean syllable block is greater than. Call that the "base syllable".
   b. Take the weight of the base syllable from the tailored weight table and increment by one. This will correspond to the gap weight in the table.
   c. Concatenate a weight sequence consisting of the gap weight, followed by a byte weight for each of the jamo in the decomposed representation of the standard Korean syllable block, followed by the byte for the terminator weight.

Data Method Example

The data method provides for the following order of weights, where the X_b are all the scripts sorted before Hangul, and the X_a are all those sorted after.

X_b	L	X_a	T	V

This ordering gives the right results among the following:

Chars	Weights			Comments
$L_1V_1X_a$	W_{L1}	W_{V1}	W_{Xa}	
L_1V_1L ...	W_{L1}	W_{V1}	W_{Ln} ...	
$L_1V_1X_b$	W_{L1}	W_{V1}	W_{Xb}	
$L_1V_1T_1$	W_{L1}	W_{V1}	W_{T1}	Works because W_T > all W_X and W_L
$L_1V_1V_2$	W_{L1}	W_{V1}	W_{V2}	Works because W_V > all W_T
$L_1L_2V_1$	W_{L1L2}	W_{V1}		Works if L_1L_2 is a contraction

The disadvantages of the data method are that the weights for T and V are separated from those of L, which can cause problems for sort key compression, and that a combination of LL that is outside the contraction table will not sort properly.

Terminator Method Example

The terminator method would assign the following weights:

Ⓣ	X_b	T	V	L	X_a

This ordering gives the right results among the following:

Chars	Weights				Comments
$L_1V_1X_a$	W_{L1}	W_{V1}	Ⓣ	W_{Xa}	
$L_1V_1L_n$ ...	W_{L1}	W_{V1}	Ⓣ	W_{Ln} ...	
$L_1V_1X_b$	W_{L1}	W_{V1}	Ⓣ	W_{Xb}	

$L_1V_1T_1$	W_{L1}	W_{V1}	W_{T1}	Ⓣ	Works because W_T > all W_X and Ⓣ
$L_1V_1V_2$	W_{L1}	W_{V1}	W_{V2}	Ⓣ	Works because W_V > all W_T
$L_1L_2V_1$	W_{L1}	W_{L2}	W_{V1}	Ⓣ	Works because W_L > all W_V

The disadvantages of the terminator method are that an extra weight is added to all Hangul syllables, increasing the length of sort keys by roughly 40%, and the fact that the terminator weight is non-contiguous can disable sort key compression.

Interleaving Method Example

The interleaving method provides for the following assignment of weights. W_n represents the weight of a Hangul syllable, and $W_{n'}$ is the weight of the gap right after it. The L, V, T weights will only occur after a W, and thus can be considered part of an entire weight.

X_b	W	X_a

Byte weights:

Ⓣ	T	V	L

This ordering gives the right results among the following:

Chars	Weights		Comments
$L_1V_1X_a$	W_n	X_a	
$L_1V_1L_n$ ...	W_n	W_k ...	The L_n will start another syllable
$L_1V_1X_b$	W_n	X_b	
$L_1V_1T_1$	W_m		Works because $W_m > W_n$
$L_1V_1V_2$	$W_{m'L1V1V2}$Ⓣ		Works because $W_{m'} > W_m$
$L_1L_2V_1$	$W_{m'L1L2V1}$Ⓣ		Works because the byte weight for $_{L2}$ > all $_V$

The interleaving method is somewhat more complex than the others, but produces the shortest sort keys for all of the precomposed Hangul syllables, so for normal text it will have the shortest sort keys. If there were a large percentage of ancient Hangul syllables, the sort keys would be longer than other methods.

10.2 Tertiary Weight Table

In the DUCET, characters are given tertiary weights according to *Table 17*. The Decomposition Type is from the Unicode Character Database [UAX44]. The Case or Kana Subtype entry refers either to a case distinction or to a specific list of characters. The weights are from MIN = 2 to MAX = $1F_{16}$, excluding 7, which is not used for historical reasons. The MAX value 1F was used for some trailing collation elements. This usage began with UCA version 9 (Unicode 3.1.1) and continued until UCA version 6.2. It is no longer used in the DUCET.

The Samples show some minimal values that are distinguished by the different weights. All values are distinguished. The samples have empty cells when there are no (visible) values showing a distinction.

Table 17. Tertiary Weight Assignments

Decomposition Type	Case or Kana Subtype	Weight	Samples					
NONE		0x0002	i	ب	)	mw	1/2	***X***

<wide>		0x0003	ｉ					
<compat>		0x0004	i, ⁱ					
<font>		0x0005	𝐢					
<circle>		0x0006	ⓘ					
!unused!		0x0007						
NONE	Uppercase	0x0008	I			MW		
<wide>	Uppercase	0x0009	Ｉ		)			
<compat>	Uppercase	0x000A	I					
<font>	Uppercase	0x000B	ℑ					
<circle>	Uppercase	0x000C	Ⓘ					
<small>	small hiragana (3041, 3043, ...)	0x000D						ぁ
NONE	normal hiragana (3042, 3044, ...)	0x000E						あ
<small>	small katakana (30A1, 30A3, ...)	0x000F			)			ァ
<narrow>	small narrow katakana (FF67..FF6F)	0x0010						ｧ
NONE	normal katakana (30A2, 30A4, ...)	0x0011						ア
<narrow>	narrow katakana (FF71..FF9D), narrow hangul (FFA0..FFDF)	0x0012						ｱ
<circle>	circled katakana (32D0..32FE)	0x0013						㋐
<super>		0x0014			)			
<sub>		0x0015			)			
<vertical>		0x0016			︶			
<initial>		0x0017		ﺑ				
<medial>		0x0018		ﺑ				
<final>		0x0019		ﺐ				
<isolated>		0x001A		ﺐ				
<noBreak>		0x001B						
<square>		0x001C				㎿		
<square>, <super>, <sub>	Uppercase	0x001D				㏿		
<fraction>		0x001E					½	
n/a	(MAX value)	0x001F						

The <compat> weight 0x0004 is given to characters that do not have more specific decomposition types. It includes superscripted and subscripted combining letters, for example U+0365 COMBINING LATIN SMALL LETTER I and U+1DCA COMBINING LATIN SMALL LETTER R BELOW. These combining letters occur in abbreviations in Medieval manuscript traditions.

11 Searching and Matching

Language-sensitive searching and matching are closely related to collation. Strings that compare as equal at some strength level should be matched when doing language-sensitive matching. For example, at a primary strength, "ß" would match against "ss" according to the UCA, and "aa" would match "å" in a Danish tailoring of the UCA. The main difference from the collation comparison operation is that the ordering is not important. Thus for matching it does not matter that "å" would sort after "z" in a Danish tailoring—the only relevant information is that they do not match.

The basic operation is matching: determining whether string X matches string Y. Other operations are built on this:

- Y contains X when there is some substring of Y that matches X
- A search for a string X in a string Y succeeds if Y contains X.
- Y starts with X when some initial substring of Y matches X
- Y ends with X when some final substring of Y matches X

The collation settings determine the results of the matching operation (see *Section 8.1, Parametric Tailoring*). Thus users of searching and matching need to be able to modify parameters such as locale or comparison strength. For example, setting the strength to exclude differences at Level 3 has the effect of ignoring case and compatibility format distinctions between letters when matching. Excluding differences at Level 2 has the effect of also ignoring accentual distinctions when matching.

Conceptually, a string matches some target where a substring of the target has the same sort key, but there are a number of complications:

1. The lengths of matching strings may differ: "aa" and "å" would match in Danish.
2. Because of ignorables (at different levels), there are different possible positions where a string matches, depending on the attribute settings of the collation. For example, if hyphens are ignorable for a certain collation, then "abc" will match "abc", "ab-c", "abc-", "-abc-", and so on.
3. Suppose that the collator has contractions, and that a contraction spans the boundary of the match. Whether it is considered a match may depend on user settings, just as users are given a "Whole Words" option in searching. So in a language where "ch" is a contraction with a different primary from "c", "bac" would not match in "bach" (given the proper user setting).
4. Similarly, combining character sequences may need to be taken into account. Users may not want a search for "abc" to match in "...abç..." (with a cedilla on the c). However, this may also depend on language and user customization. In particular, a useful technique is discussed in *Section 11.2, Asymmetric Search*.
5. The above two conditions can be considered part of a general condition: "Whole Characters Only"; very similar to the common "Whole Words Only" checkbox that is included in most search dialog boxes. (For more information on grapheme clusters and searching, see [UAX29] and [UTS18].)
6. If the matching does not check for "Whole Characters Only," then some other complications may occur. For example, suppose that P is "x^", and Q is "x ^¸". Because the cedilla and circumflex can be written in arbitrary order and still be equivalent, in most cases one would expect to find a match for P in Q. A canonically-equivalent matching process requires special processing at the boundaries to check for situations like this. (It does not require such special processing within the P or the substring of Q because collation is defined to observe canonical equivalence.)

The following are used to provide a clear definition of searching and matching that deal with the above complications:

DS1. Define *S[start,end]* to be the substring of S that includes the character after the offset *start* up to the character before offset *end*. For example, if S is "abcd", then S[1,3] is "bc". Thus S = S[0,length(S)].

DS1a. A boundary condition is a test imposed on an offset within a string. An example includes Whole Word Search, as defined in [UAX29].

The tailoring parameter *match-boundaries* specifies constraints on matching (see *Section 8.1, Parametric Tailoring*). The parameter *match-boundaries=whole-character* requires that the start and

end of a match each be on a grapheme boundary. The value *match-boundaries=whole-word* further requires that the start and end of a match each be on a word boundary as well. For more information on the specification of these boundaries, see [UAX29].

By using grapheme-complete conditions, contractions and combining sequences are not interrupted except in edge cases. This also avoids the need to present visually discontiguous selections to the user (except for BIDI text).

Suppose there is a collation C, a pattern string P and a target string Q, and a boundary condition B. C has some particular set of attributes, such as a strength setting, and choice of variable weighting.

DS2. The pattern string P *has a match at Q[s,e] according to collation C* if C generates the same sort key for P as for Q[s,e], and the offsets *s* and *e* meet the boundary condition B. One can also say P has a match in Q according to C.

DS3. The pattern string P has a *canonical* match at Q[s,e] according to collation C if there is some Q' that is canonically equivalent to Q[*s,e*], and P has a match in Q'.

> For example, suppose that P is "Å", and Q is "...A◌̊◌̊...". There would not be a match for P in Q, but there would be a canonical match, because P does have a match in "A◌̊◌̊", which is canonically equivalent to "A◌̊◌̊". However, it is not commonly necessary to use canonical matches, so this definition is only supplied for completeness.

Each of the following definitions is a qualification of DS2 or DS3:

DS3a. The match is *grapheme-complete* if B requires that the offset be at a grapheme cluster boundary. Note that Whole Word Search as defined in [UAX29] is grapheme complete.

DS4. The match is *minimal* if there is no match at Q[*s+i,e-j*] for any *i* and *j such that i* ≥ *0, j* ≥ 0, and *i* + *j* > 0. In such a case, one can also say that P has a *minimal* match *at* Q[*s,e*].

DS4a. A *medial* match is determined in the following way:

1. Determine the minimal match for P at Q[s,e]
2. Determine the "minimal" pattern P[m,n], by finding:
    1. the largest m such that P[m,len(P)] matches P, then
    2. the smallest n such that P[m,n] matches P.
3. Find the smallest s' ≤ s such that Q[s',s] is canonically equivalent to P[m',m] for some m'.
4. Find the largest e' ≥ e such that Q[e',e'] is canonically equivalent to P[n', n'] for some n'.
5. The medial match is Q[s', e'].

DS4b. The match is *maximal* if there is no match at Q[*s-i,e+j*] for any *i* and *j such that i* ≥ *0, j* ≥ 0, and *i* + *j* > 0. In such a case, one can also say that P has a *maximal* match *at* Q[*s,e*].

Figure 5 illustrates the differences between these type of matches, where the collation strength is set to ignore punctuation and case, and **format** indicates the match.

Figure 5. Minimal, Medial, and Maximal Matches

	Text	Description
Pattern	*!abc!*	Notice that the *! and !* are ignored in matching.
Target Text	def$!Abc%$ghi	
Minimal Match	def$!**Abc**%$ghi	The minimal match is the tightest one, because $! and %$ are ignored in the target.
Medial Match	def$**!Abc**%$ghi	The medial one includes those characters that are binary equal.

Maximal Match	def$!Abc%$ghi	The maximal match is the loosest one, including the surrounding ignored characters.

By using minimal, maximal, or medial matches, the issue with ignorables is avoided. Medial matches tend to match user expectations the best.

When an additional condition is set on the match, the types (minimal, maximal, medial) are based on the matches *that meet that condition.* Consider the example in *Figure 6*.

Figure 6. Alternate End Points for Matches

	Value	Notes
Pattern	abc	
Strength	*primary*	thus ignoring combining marks, punctuation
Text	abc◌̧-◌̊d	two combining marks, cedilla and ring
Matches	|abc|◌̧|-|◌̊|d	four possible end points, indicated by |

If, for example, the condition is Whole Grapheme, then the matches are restricted to "abc◌̧|-◌̊|d", thus discarding match positions that would not be on a grapheme cluster boundary. In this case the minimal match would be "abc◌̧|-◌̊d"

DS6. The *first forward match* for P in Q starting at *b* is the least offset *s* greater than or equal to *b* such that for some *e*, P matches within Q[s,e].

DS7. The *first backward match* for P in Q starting at *b* is the greatest offset *s* less than or equal to *b* such that for some *e*, P matches within Q[s,e].

In DS6 and DS7, matches can be minimal, medial, or maximal; the only requirement is that the combination in use in DS6 and DS7 be specified. Of course, a possible match can also be rejected on the basis of other conditions, such as being grapheme-complete or applying Whole Word Search, as described in [UAX29]).

The choice of medial or minimal matches for the "starts with" or "ends with" operations only affects the positioning information for the end of the match or start of the match, respectively.

Special Cases. Ideally, the UCA at a secondary level would be compatible with the standard Unicode case folding and removal of compatibility differences, especially for the purpose of matching. For the vast majority of characters, it is compatible, but there are the following exceptions:

1. The UCA maintains compatibility with the DIN standard for sorting German by having the German *sharp-s* (U+00DF (ß) LATIN SMALL LETTER SHARP S) sort as a secondary difference with "SS", instead of having ß and SS match at the secondary level.
2. Compatibility normalization (NFKC) folds stand-alone accents to a combination of space + combining accent. This was not the best approach, but for backwards compatibility cannot be changed in NFKC. UCA takes a better approach to weighting stand-alone accents, but as a result does not weight them exactly the same as their compatibility decompositions.
3. Case folding maps *iota-subscript* (U+0345 (ͅ) COMBINING GREEK YPOGEGRAMMENI) to an iota, due to the special behavior of iota-subscript, while the UCA treats *iota-subscript* as a regular combining mark (secondary collation element).
4. When compared to their case and compatibility folded values, UCA compares the following as different at a secondary level, whereas other compatibility differences are at a tertiary level.
   - U+017F (ſ) LATIN SMALL LETTER LONG S (and precomposed characters containing it)
   - U+1D4C (ᵊ) MODIFIER LETTER SMALL TURNED OPEN E
   - U+2D6F (ⵯ) TIFINAGH MODIFIER LETTER LABIALIZATION MARK

In practice, most of these differences are not important for modern text, with one exception: the German ß. Implementations should consider tailoring ß to have a tertiary difference from SS, at least when collation tables are used for matching. Where full compatibility with case and compatibility folding are required, either the text can be preprocessed, or the UCA tables can be tailored to handle the outlying cases.

11.1 Collation Folding

Matching can be done by using the collation elements, directly, as discussed above. However, because matching does not use any of the ordering information, the same result can be achieved by a folding. That is, two strings would fold to the same string if and only if they would match according to the (tailored) collation. For example, a folding for a Danish collation would map both "Gård" and "gaard" to the same value. A folding for a primary-strength folding would map "Resume" and "résumé" to the same value. That folded value is typically a lowercase string, such as "resume".

A comparison between folded strings cannot be used for an ordering of strings, but it can be applied to searching and matching quite effectively. The data for the folding can be smaller, because the ordering information does not need to be included. The folded strings are typically much shorter than a sort key, and are human-readable, unlike the sort key. The processing necessary to produce the folding string can also be faster than that used to create the sort key.

The following is an example of the mappings used for such a folding using to the [CLDR] tailoring of UCA:

Parameters:

{locale=da_DK, strength=secondary, alternate=shifted}

Mapping:

...			
ᵃ	→	a	Map compatibility (tertiary) equivalents, such as full-width and superscript characters, to representative character(s)
ａ	→	a	
A	→	a	
Ａ	→	a	
...			
å	→	aa	Map contractions (a + ring above) to equivalent values
Å	→	aa	
...			

Once the table of such mappings is generated, the folding process is a simple longest-first match-and-replace: a string to be folded is first converted to NFD, then at each point in the string, the longest match from the table is replaced by the corresponding result.

However, ignorable characters need special handling. Characters that are fully ignorable at a given strength level normally map to the empty string. For example, at *strength=quaternary*, most controls and format characters map to the empty string; at *strength=primary*, most combining marks also map to the empty string. In some contexts, however, fully ignorable characters may have an effect on comparison, or characters that are not ignorable at the given strength level may be treated as ignorable.

1. Any discontiguous contractions need to be detected in the process of folding and handled according to Rule S2.1. For more information about discontiguous contractions, see *Section 3.3.3, Contractions.*

2. An ignorable character may interrupt what would otherwise be a contraction. For example, suppose that "ch" is a contraction sorting after "h", as in Slovak. In the absence of special tailoring, a CGJ or SHY between the "c" and the "h" prevents the contraction from being formed, and causes "c<CGJ>h" to not compare as equal to "ch". If the CGJ is simply folded away, they would incorrectly compare as equal. See also *Section 8.3, Use of Combining Grapheme Joiner*.
3. With the parameter values *alternate=shifted* or *alternate=blanked*, any (partially) ignorable characters after variable collation elements have their weights reset to zero at levels 1 to 3, and may thus become fully ignorable. In that context, they would also be mapped to the empty string. For more information, see *Section 4, Variable Weighting.*

11.2 Asymmetric Search

Users often find *asymmetric searching* to be a useful option. When doing an asymmetric search, a character (or grapheme cluster) in the query that is *unmarked* at the secondary and/or tertiary levels will match a character in the target that is either marked or unmarked at the same levels, but a character in the query that is *marked* at the secondary and/or tertiary levels will only match a character in the target that is marked in the same way.

At a given level, a character is unmarked if it has the lowest collation weight for that level. For the tertiary level, a plain lowercase ‘r’ would normally be treated as unmarked, while the uppercase, fullwidth, and circled characters ‘R’, ‘ ｒ ’, ‘ⓡ’ would be treated as marked. There is an exception for *kana* characters, where the "normal" form is unmarked: `0x000E` for *hiragana* and `0x0011` for *katakana*.

For the secondary level, an unaccented ‘e’ would be treated as unmarked, while the accented letters ‘é’, ‘è’ would (in English) be treated as marked. Thus in the following examples, a lowercase query character matches that character or the uppercase version of that character even if *strength* is set to tertiary, and an unaccented query character matches that character or any accented version of that character even if *strength* is set to secondary.

Asymmetric search with strength = tertiary

Query	Target Matches
resume	resume, Resume, RESUME, résumé, rèsumè, Résumé, RÉSUMÉ, …
Resume	Resume, RESUME, Résumé, RÉSUMÉ, …
résumé	résumé, Résumé, RÉSUMÉ, …
Résumé	Résumé, RÉSUMÉ, …
けんこ	けんこ, げんこ, けんご, げんご, …
げんご	げんご, …

Asymmetric search with strength = secondary

Query	Target Matches
resume	resume, Resume, RESUME, résumé, rèsumè, Résumé, RÉSUMÉ, …
Resume	resume, Resume, RESUME, résumé, rèsumè, Résumé, RÉSUMÉ, …
résumé	résumé, Résumé, RÉSUMÉ, …
Résumé	résumé, Résumé, RÉSUMÉ, …
けんこ	けんこ, ケンコ, げんこ, けんご, ゲンコ, ケンゴ, げんご, ゲンゴ, …
げんご	げんご, ゲンゴ, …

11.2.1 Returning Results

When doing an asymmetric search, there are many ways in which results might be returned:

1. Return the next single match in the text.
2. Return an unranked set of all the matches in the text, which could be used for highlighting all of the matches on a page.
3. Return a set of matches in which each match is ranked or ordered based on the closeness of the match. The closeness might be determined as follows:
   - The closest matches are those in which there is no secondary difference between the query and target; the closeness is based on the number of tertiary differences.
   - These are followed by matches in which there is a secondary difference between query and target, ranked first by number of secondary differences, and then by number of tertiary differences.

12 Data Files

The data files for each version of UCA are located in versioned subdirectories in [Data10]. The main data file with the DUCET data for each version is **allkeys.txt** [Allkeys].

Starting with Version 3.1.1 of UCA, the data directory also contains **CollationTest.zip**, a zipped file containing conformance test files. See *Section 12.2, Conformance Tests*.

Starting with Version 6.2.0 of UCA, the data directory also contains **decomps.txt**. This file lists the decompositions used when generating the DUCET. These decompositions are loosely based on the normative decomposition mappings defined in the Unicode Character Database, often mirroring the NFKD form. However, those decomposition mappings are adjusted as part of the input to the generation of DUCET, in order to produce default weights more appropriate for collation. For more details and a description of the file format, see the header of the **decomps.txt** file.

12.1 Allkeys File Format

The **allkeys.txt** file consists of a version line followed by a series of entries, all separated by newlines. A '#' or '%' and any following characters on a line are comments. Whitespace between literals is ignored. The following is an extended BNF description of the format, where "*x*+" indicates one or more *x*'s, "*x**" indicates zero or more *x*'s, "*x?*" indicates zero or one *x*, <char> is a hexadecimal Unicode code point value, and <weight> is a hexadecimal collation weight value.

```
<collationElementTable> := <version>
                           <implicitweights>*
                           <entry>+
```

The `<version>` line is of the form:

```
<version> := '@version' <major>.<minor>.<variant> <eol>
```

It is optionally followed by one or more lines that specify the parameters for computing implicit primary weights for some ranges of code points, see *Section 10.1.3, Implicit Weights* for details. An `<implicitweights>` line specifies a range of code points, from which unassigned code points are to be excluded, and the 16-bit primary-weight lead unit (AAAA in Section 10.1.3) for the implicit weights. (New in version 9.0.0.)

```
@implicitweights 17000..18AFF; FB00 # Tangut and Tangut Components
```

Each `<entry>` is a mapping from character(s) to collation element(s), and is of the following form:

```
<entry>       := <charList> ';' <collElement>+ <eol>
<charList>    := <char>+
<collElement> := "[" <alt> <weight> "." <weight> "." <weight> ("." <weight>)? "]"
<alt>         := "*" | "."
```

Collation elements marked with a "*" are *variable*.

Every collation element in the table should have the same number of fields.

Here are some selected entries taken from a particular version of the data file. (It may not match the actual values in the current data file.)

```
0020 ; [*0209.0020.0002] # SPACE
02DA ; [*0209.002B.0002] # RING ABOVE
0041 ; [.06D9.0020.0008] # LATIN CAPITAL LETTER A
3373 ; [.06D9.0020.0017] [.08C0.0020.0017] # SQUARE AU
00C5 ; [.06D9.002B.0008] # LATIN CAPITAL LETTER A WITH RING ABOVE
212B ; [.06D9.002B.0008] # ANGSTROM SIGN
0042 ; [.06EE.0020.0008] # LATIN CAPITAL LETTER B
0043 ; [.0706.0020.0008] # LATIN CAPITAL LETTER C
0106 ; [.0706.0022.0008] # LATIN CAPITAL LETTER C WITH ACUTE
0044 ; [.0712.0020.0008] # LATIN CAPITAL LETTER D
```

Implementations can also add more customizable levels, as discussed in *Section 2, Conformance*. For example, an implementation might want to handle the standard Unicode Collation, but also be capable of emulating an EBCDIC multi-level ordering (having a fourth-level EBCDIC binary order).

12.2 Conformance Tests

The following files provide conformance tests for the Unicode Collation Algorithm.

- CollationTest_SHIFTED.txt
- CollationTest_NON_IGNORABLE.txt
- CollationTest_SHIFTED_SHORT.txt
- CollationTest_NON_IGNORABLE_SHORT.txt

These files are large, and thus packaged in zip format to save download time.

The zip file is available in [Tests10].

> **Note:** These files test the sort order of an untailored DUCET table. If you are using an implementation of the CLDR Collation Algorithm with its tailored root collation data, for example ICU or a library that uses ICU for collation, then you need to test with files that reflect that sort order. The CLDR collation conformance test files have the same names (except for an added _CLDR infix) and structures as the ones here for the DUCET. You can find them in the CLDR GitHub repo in the folder “common/uca”, or in the CLDR data file download area, in the “cldr-common-*.zip” file, again in the folder “common/uca”. Select the files for the version of CLDR that is used in the implementation.

Format

There are four different files:

- The shifted vs non-ignorable files correspond to the two alternate Variable Weighting values.
- The SHORT versions omit the comments, for more compact storage.

The format is illustrated by the following example:

```
0385 0021;  # (¨) GREEK DIALYTIKA TONOS  [0316 015D | 0020 0032 0020 | 0002 0002 0002 |]
```

The part before the semicolon is the hex representation of a sequence of Unicode code points. After the hash mark is a comment. This comment is purely informational, and may change in the future. Currently it consists of the characters of the sequence in parentheses, the name of the first code point, and a representation of the sort key for the sequence.

The sort key representation is in square brackets. It uses a vertical bar for the ZERO separator. Between the bars are the primary, secondary, tertiary, and quaternary weights (if any), in hex.

Note: The sort key is purely informational. UCA does *not* require the production of any particular sort key, as long as the results of comparisons match.

Testing

The files are designed so each line in the file will order as being greater than or equal to the previous one, when using the UCA and the Default Unicode Collation Element Table. A test program can read in each line, compare it to the last line, and signal an error if order is not correct. The exact comparison that should be used is as follows:

1. Read the next line.
2. Parse each sequence up to the semicolon, and convert it into a Unicode string.
3. Compare that string with the string on the previous line, according to the UCA implementation, with strength = identical level (using S3.10).
4. If the last string is greater than the current string, then stop with an error.
5. Continue to the next line (step 1).

If there are any errors, then the UCA implementation is not compliant.

These files contain test cases that include ill-formed strings, with surrogate code points. Implementations that do not weight surrogate code points the same way as reserved code points may filter out such lines in the test cases, before testing for conformance.

Appendix A: Deterministic Sorting

There is often a good deal of confusion about what is meant by the terms "stable" or "deterministic" when applied to sorting or comparison. This confusion in terms often leads people to make mistakes in their software architecture, or make choices of language-sensitive comparison options that have significant impact on performance and memory use, and yet do not give the results that users expect.

A.1 Stable Sort

A stable sort is an algorithm where two records with equal key fields will have the same relative order that they were in before sorting, although their positions relative to other records may change. Importantly, this is a property of the sort algorithm, *not* the comparison mechanism.

Two examples of differing sort algorithms are Quicksort and Merge sort. Quicksort is not stable while Merge sort is stable. (A Bubble sort, as typically implemented, is also stable.)

- For background on the names and characteristics of different sorting methods, see [SortAlg]
- For a definition of stable sorting, see [Unstable]

Assume the following records:

Original Records

Record	Last_Name	First_Name
1	Davis	John
2	Davis	Mark
3	Curtner	Fred

The results of a Merge sort on the Last_Name field only are:

Merge Sort Results

Record	Last_Name	First_Name

3	Curtner	Fred
1	Davis	John
2	Davis	Mark

The results of a Quicksort on the Last_Name field only are:

Quicksort Results

Record	Last_Name	First_Name
3	Curtner	Fred
2	Davis	Mark
1	Davis	John

As is apparent, the Quicksort algorithm is not stable; records 1 and 2 are not in the same order they were in before sorting.

A stable sort is often desirable—for one thing, it allows records to be successively sorted according to different fields, and to retain the correct lexicographic order. Thus, with a stable sort, an application could sort all the records by First_Name, and then sort them again by Last_Name, giving the desired results: that all records would be ordered by Last_Name, and in the case where the Last_Name values are the same, be further subordered by First_Name.

A.1.1 Forcing a Stable Sort

A non-stable sort algorithm can be forced to produce stable results by comparing the *current record number* (or some other monotonically increasing value) for otherwise equal strings.

If such a modified comparison is used, for example, it forces Quicksort to get the same results as a Merge sort. In that case, ignored characters such as Zero Width Joiner (ZWJ) do not affect the outcome. The correct results occur, as illustrated below. The results below are sorted first by last name, then by first name.

Last_Name then Record Number (Forced Stable Results)

Record	Last_Name	First_Name
3	Curtner	Fred
1	Da(ZWJ)vis	John
2	Davis	Mark

If anything, this then is what users want when they say they want a deterministic comparison. See also *Section 1.6, Merging Sort Keys*.

A.2 Deterministic Sort

A *deterministic* sort is a sort algorithm that returns the same results each time. On the face of it, it would seem odd for any sort algorithm to *not* be deterministic, but there are examples of real-world sort algorithms that are not.

The key concept is that these sort algorithms *are* deterministic when two records have unequal fields, but they may return different results at different times when two records have equal fields.

For example, a classic Quicksort algorithm works recursively on ranges of records. For any given range of records, it takes the first element as the *pivot element*. However, that algorithm performs badly with input data that happens to be already sorted (or mostly sorted). A randomized Quicksort,

which picks a random element as the pivot, can on average be faster. Because of this random selection, different outputs can result from *exactly* the same input: the algorithm is not deterministic.

Enhanced Quicksort Results (Sorted by Last_Name Only)

Record	Last_Name	First_Name
3	Curtner	Fred
2	Davis	John
1	Davis	Mark

or

Record	Last_Name	First_Name
3	Curtner	Fred
1	Davis	Mark
2	Davis	John

As another example, multiprocessor sort algorithms can be non-deterministic. The work of sorting different blocks of data is farmed out to different processors and then merged back together. The ordering of records with equal fields might be different according to when different processors finish different tasks.

Note that a deterministic sort is weaker than a stable sort. A stable sort is always deterministic, but not vice versa. Typically, when people say they want a deterministic sort, they really mean that they want a stable sort.

A.3 Deterministic Comparison

A *deterministic comparison* is different than either a stable sort or a deterministic sort; it is a property of a comparison function, not a sort algorithm. This is a comparison where strings that do not have identical binary contents (optionally, after some process of normalization) will compare as unequal. A deterministic comparison is sometimes called a *stable* (or *semi-stable*) *comparison*.

There are many people who confuse a deterministic comparison with a deterministic (or stable) sort, but this ignores the fundamental difference between a comparison and a sort. A comparison is used by a sort algorithm to determine the relative ordering of two fields, such as strings. Using a deterministic comparison cannot cause a sort to be deterministic, nor to be stable. Whether a sort is deterministic or stable is a property of the sort algorithm, not the comparison function, as the prior examples show.

A.3.1 Avoid Deterministic Comparisons

A deterministic comparison is generally not good practice.

First, it has a certain performance cost in comparison, and a quite substantial impact on sort key size. (For example, ICU language-sensitive sort keys are generally about the size of the original string, so appending a copy of the original string to force a deterministic comparison generally doubles the size of the sort key.) A database using these sort keys will use more memory and disk space and thus may have reduced performance.

Second, a deterministic comparison function does not affect the order of equal fields. Even if such a function is used, the order of equal fields is not guaranteed in the Quicksort example, because the two records in question have identical Last_Name fields. It does not make a non-deterministic sort into a deterministic one, nor does it make a non-stable sort into a stable one.

Third, a deterministic comparison is often not what is wanted, when people look closely at the implications. This is especially the case when the key fields are not guaranteed to be unique according to the comparison function, as is the case for collation where some variations are ignored.

To illustrate this, look at the example again, and suppose that this time the user is sorting first by last name, then by first name.

Original Records

Record	Last_Name	First_Name

1	Davis	John
2	Davis	Mark
3	Curtner	Fred

The desired results are the following, which should result whether the sort algorithm is stable or not, because it uses both fields.

Last Name then First Name

Record	Last_Name	First_Name
3	Curtner	Fred
1	Davis	John
2	Davis	Mark

Now suppose that in record 2, the source for the data caused the last name to contain a format control character, such as a Zero Width Joiner (ZWJ, used to request ligatures on display). In this case there is no visible distinction in the forms, because the font does not have any ligatures for these sequences of Latin letters. The default UCA collation weighting causes the ZWJ to be—correctly—ignored in comparison, since it should only affect rendering. However, if that comparison is changed to be deterministic (by appending the binary values for the original string), then unexpected results will occur.

Last Name then First Name (Deterministic)

Record	Last_Name	First_Name
3	Curtner	Fred
2	Davis	Mark
1	Da(ZWJ)vis	John

Typically, when people ask for a *deterministic comparison*, they actually want a *stable sort* instead.

A.3.2 Forcing Deterministic Comparisons

One can produce a deterministic comparison function from a non-deterministic one, in the following way (in pseudo-code):

```
int new_compare (String a, String b) {
  int result = old_compare(a, b);
  if (result == 0) {
    result = binary_compare(a, b);
  }
  return result;
}
```

Programs typically also provide the facility to generate a *sort key*, which is a sequences of bytes generated from a string in alignment with a comparison function. Two sort keys will binary-compare in the same order as their original strings. The simplest means to create a deterministic sort key that aligns with the above `new_compare` is to append a copy of the original string to the sort key. This will force the comparison to be deterministic.

```
byteSequence new_sort_key (String a) {
  return old_sort_key(a) + SEPARATOR + toByteSequence(a);
}
```

Because sort keys and comparisons must be aligned, a sort key generator is deterministic if and only if a comparison is.

Some collation implementations offer the inclusion of the identical level in comparisons and in sort key generation, appending the NFD form of the input strings. Such a comparison is deterministic except that it ignores differences among canonically equivalent strings.

A.4 Stable and Portable Comparison

There are a few other terms worth mentioning, simply because they are also subject to considerable confusion. Any or all of the following terms may be easily confused with the discussion above.

A *stable comparison* is one that does not change over successive software versions. That is, as an application uses successive versions of an API, with the same "settings" (such as locale), it gets the same results.

A *stable sort key generator* is one that generates the same binary sequence over successive software versions.

> **Warning:** If the sort key generator is stable, then the associated comparison will necessarily be. However, the reverse is not guaranteed. To take a trivial example, suppose the new version of the software always adds the byte 0xFF at the start of every sort key. The results of any comparison of any two new keys would be identical to the results of the comparison of any two corresponding old keys. However, the bytes have changed, and the comparison of old and new keys would give different results. Thus there can be a stable comparison, yet an associated non-stable sort key generator.

A *portable comparison* is where corresponding APIs for comparison produce the same results across different platforms. That is, if an application uses the same "settings" (such as locale), it gets the same results.

A *portable sort key generator* is where corresponding sort key APIs produce exactly the same sequence of bytes across different platforms.

> **Warning:** As above, a comparison may be portable without the associated sort key generator being portable.

Ideally, all products would have the same string comparison and sort key generation for, say Swedish, and thus be portable. For historical reasons, this is not the case. Even if the main letters sort the same, there will be differences in the handling of other letters, or of symbols, punctuation, and other characters. There are some libraries that offer portable comparison, such as [ICUCollator], but in general the results of comparison or sort key generation may vary significantly between different platforms.

In a closed system, or in simple scenarios, portability may not matter. Where someone has a given set of data to present to a user, and just wants the output to be reasonably appropriate for Swedish, the exact order on the screen may not matter.

In other circumstances, differences can lead to data corruption. For example, suppose that two implementations do a database query for records between a pair of strings. If the collation is different in the least way, they can get different data results. Financial data might be different, for example, if a city is included in one query on one platform and excluded from the same query on another platform.

Appendix B: Synchronization with ISO/IEC 14651

The Unicode Collation Algorithm is maintained in synchronization with the International Standard, ISO/IEC 14651 [ISO14651]. Although the presentation and text of the two standards are rather distinct, the approach toward the architecture of multi-level collation weighting and string comparison is closely aligned. In particular, the synchronization between the two standards is built around the data tables which define the default (or tailorable) weights. The UCA adds many additional specifications, implementation guidelines, and test cases, over and above the synchronized weight tables. This relationship between the two standards is similar to that maintained between the Unicode Standard and ISO/IEC 10646.

For each version of the UCA, the Default Unicode Collation Element Table (DUCET) [Allkeys] is constructed based on the repertoire of the corresponding version of the Unicode Standard. The synchronized version of ISO/IEC 14651 has a Common Template Table (CTT) built for the same repertoire and ordering. The two tables are constructed with a common tool, to guarantee identical default (or tailorable) weight assignments. The CTT for ISO/IEC 14651 is constructed using only symbols, rather than explicit integral weights, and with the Shifted option for variable weighting. The detailed description of the syntax of the CTT, as well as the specification of how the symbols are interpreted and then used to weight strings for collation ordering, can be found in ISO/IEC 14651.

The detailed synchronization points between versions of UCA and published editions (or amendments) of ISO/IEC 14651 are shown in *Table 18*.

The column labeled "CTT Name" is the normative name of the CTT. Through the 6th edition of ISO/IEC 14651, that name was published in each new edition of ISO/IEC 14651 or its published amendments. The text of that CTT, and a parallel version of each table translated into French, is available from the ISO Standards Maintenance Portal. For Version 16.0 of the UCA, a corresponding version of the CTT is posted, instead, at the Unicode website at https://www.unicode.org/Public/CTT/16.0.0/. Starting with Version 17.0, the CTT can always be found at https://www.unicode.org/Public/latest/uca/. Rather than having a normative table name published in a new edition of ISO/IEC 14651, the CTT is labeled and can be referred to by the corresponding UCA version number.

Table 18. UCA and ISO/IEC 14651

UCA Version	UTS #10 Date	DUCET File Date	ISO/IEC 14651 Reference	CTT Name	CTT Permalink
18.0.0	2026-xx-xx	2026-xx-xx	14651:2025 (7th ed.)	CTT_V18_0	CTT
17.0.0	2025-08-13	2025-07-23	14651:2025 (7th ed.)	CTT_V17_0	CTT
16.0.0	2024-08-22	2024-04-25	---	CTT_V16_0	CTT
15.1.0	2023-08-14	2023-05-09	---	---	
15.0.0	2022-08-26	2022-08-09	---	---	
14.0.0	2021-08-27	2021-07-10	---	---	
13.0.0	2020-02-07	2020-01-28	14651:2020 (6th ed.)	ISO 14651_2020_TABLE1	For the 1st through the 6th edition, see links at the ISO Standards Maintenance Portal for ISO/IEC 14651
12.1.0	2019-04-26	2019-04-01	---	---	
12.0.0	2019-03-04	2019-01-25	---	---	
11.0.0	2018-05-10	2018-02-10	---	---	
10.0.0	2017-05-26	2017-04-26	14651:2018 (5th ed.)	ISO 14651_2017_TABLE1	
9.0.0	2016-05-18	2016-05-16	14651:2016 Amd 1	ISO 14651_2016_TABLE1	
8.0.0	2015-06-01	2015-02-18	14651:2016 (4th ed.)	ISO 14651_2015_TABLE1	
7.0.0	2014-05-23	2014-04-07	14651:2011 Amd 2	ISO 14651_2014_TABLE1	
6.3.0	2013-08-13	2013-05-22	---	---	
6.2.0	2012-08-30	2012-08-14	---	---	
6.1.0	2012-02-01	2011-12-06	14561:2011 Amd 1	ISO 14651_2012_TABLE1	
6.0.0	2010-10-08	2010-08-26	14561:2011 (3rd ed.)	ISO 14651_2010_TABLE1	
5.2.0	2009-10-08	2009-09-22	---	---	
5.1.0	2008-03-28	2008-03-04	14561:2007 Amd 1	ISO 14651_2008_TABLE1	

5.0.0	2006-07-10	2006-07-14	14561:2007 (2nd ed.)	ISO 14651_2006_TABLE1	
4.1.0	2005-05-05	2005-05-02	14561:2001 Amd 3	ISO 14651_2005_TABLE1	
4.0.0	2004-01-08	2003-11-01	14561:2001 Amd 2	ISO 14651_2003_TABLE1	
9.0 (= 3.1.1)	2002-07-16	2002-07-17	14561:2001 Amd 1	ISO 14651_2002_TABLE1	
8.0 (= 3.0.1)	2001-03-23	2001-03-29	14561:2001	ISO 14651_2000_TABLE1	
6.0 (= 2.1.9)	2000-08-31	2000-04-18	---	---	
5.0 (= 2.1.9)	1999-11-22	2000-04-18	---	---	

Acknowledgements

Mark Davis authored most of the original text of this document and added numerous sections in later revisions. Markus Scherer and Ken Whistler together have added to and continue to maintain the text.

Thanks to Bernard Desgraupes, Richard Gillam, Kent Karlsson, York Karsunke, Michael Kay, Marc Lodewijck, Åke Persson, Roozbeh Pournader, Javier Sola, Otto Stolz, Ienup Sung, Yoshito Umaoka, Andrea Vine, Vladimir Weinstein, Sergiusz Wolicki, and Richard Wordingham for their feedback on previous versions of this document, to Jianping Yang and Claire Ho for their contributions on matching, and to Cathy Wissink for her many contributions to the text. Julie Allen helped in copy editing of the text.

References

[Allkeys] Default Unicode Collation Element Table (DUCET)
For the latest version, see:
https://www.unicode.org/Public/latest/uca/allkeys.txt
For the 18.0.0 version, see:
https://www.unicode.org/Public/18.0.0/uca/allkeys.txt

[CanStd] CAN/CSA Z243.4.1. For availability see https://store.csagroup.org

[CLDR] Common Locale Data Repository
http://cldr.unicode.org/

[Data10] For all UCA implementation and test data
For the latest version, see:
https://www.unicode.org/Public/latest/uca/
For the 18.0.0 version, see:
https://www.unicode.org/Public/18.0.0/uca/

[FAQ] Unicode Frequently Asked Questions
https://www.unicode.org/faq/
For answers to common questions on technical issues.

[Feedback] Reporting Errors and Requesting Information Online
https://www.unicode.org/reporting.html

[Glossary] Unicode Glossary
https://www.unicode.org/glossary/
For explanations of terminology used in this and other documents.

[ICUCollator] ICU User Guide: Collation Introduction
http://userguide.icu-project.org/collation

[ISO14651] International Organization for Standardization. *Information Technology—International String ordering and comparison—Method for comparing character strings and description of the common template tailorable ordering.* (ISO/IEC 14651:2025). For availability see https://www.iso.org

[JavaCollator] http://docs.oracle.com/javase/6/docs/api/java/text/Collator.html, http://docs.oracle.com/javase/6/docs/api/java/text/RuleBasedCollator.html

[Reports] Unicode Technical Reports
https://www.unicode.org/reports/
For information on the status and development process for technical reports, and for a list of technical reports.

[SortAlg] For background on the names and characteristics of different sorting methods, see
http://en.wikipedia.org/wiki/Sorting_algorithm

[Tests10] Conformance Test Data
For the latest version, see:
https://www.unicode.org/Public/latest/uca/CollationTest.zip
For the 18.0.0 version, see:
https://www.unicode.org/Public/18.0.0/uca/CollationTest.zip

[UAX15] UAX #15: Unicode Normalization Forms
https://www.unicode.org/reports/tr15/

[UAX29] UAX #29: Unicode Text Segmentation
https://www.unicode.org/reports/tr29/

[UAX44] UAX #44: Unicode Character Database
https://www.unicode.org/reports/tr44/

[Unicode] The Unicode Consortium. The Unicode Standard, Version 18.0.0 (South San Francisco, CA: The Unicode Consortium, 2026. ISBN 978-1-936213-NN-N)
https://www.unicode.org/versions/Unicode18.0.0/

[Unstable] For a definition of stable sorting, see
http://planetmath.org/stablesortingalgorithm

[UTN5] UTN #5: Canonical Equivalence in Applications
https://www.unicode.org/notes/tn5/

[UTS18] UTS #18: Unicode Regular Expressions
https://www.unicode.org/reports/tr18/

[UTS35] UTS #35: Unicode Locale Data Markup Language (LDML)
https://www.unicode.org/reports/tr35/

[UTS35Collation] UTS #35: Unicode Locale Data Markup Language (LDML) Part 5: Collation
https://www.unicode.org/reports/tr35/tr35-collation.html

[Versions] Versions of the Unicode Standard
https://www.unicode.org/versions/
For details on the precise contents of each version of the Unicode Standard, and how to cite them.

Migration Issues

This section summarizes important migration issues which may impact implementations of the Unicode Collation Algorithm when they are updated to a new version.

UCA 18.0.0 from UCA 17.0.0 (or earlier)

- U+FFFE maps to a collation element with the lowest primary weight, which is now not variable, with a minimal, unique weight on identical level, and with special handling for backwards secondary level sorting. See *Section 10.1.5, Reserved Weights*. Implementations of the CLDR collation algorithm already do this.
- Jurchen and Seal are additional siniform ideographic scripts which are given implicit primary weights similar to Han ideographs, see *Section 10.1.3, Implicit Weights*. The parameters for the weight computation are specified in allkeys.txt, see *Section 9.1, Allkeys File Format*.
- The Shift-Trimmed option has been removed from *Section 4, Variable Weighting*. Implementers of this option may instead use the Shifted option with strength=tertiary.

UCA 13.0.0 from UCA 12.1.0 (or earlier)

- Khitan Small Script is a siniform ideographic script which is given implicit primary weights similar to Han ideographs, see *Section 10.1.3, Implicit Weights*. The parameters for the weight computation are specified in allkeys.txt, see *Section 9.1, Allkeys File Format*.
- An additional block has been added to the ranges for calculating implicit weights for the Tangut script.

UCA 10.0.0 from UCA 9.0.0 (or earlier)

- Nüshu is a siniform ideographic script which is given implicit primary weights similar to Han ideographs, see *Section 10.1.3, Implicit Weights*. The parameters for the weight computation are specified in allkeys.txt, see *Section 9.1, Allkeys File Format*.

UCA 9.0.0 from UCA 8.0.0 (or earlier)

- Tangut is a siniform ideographic script which is given implicit primary weights similar to Han ideographs, see *Section 10.1.3, Implicit Weights*. The parameters for the weight computation are specified in allkeys.txt, see *Section 9.1, Allkeys File Format*.

UCA 8.0.0 from UCA 7.0.0 (or earlier)

- Contractions for Cyrillic accented letters have been removed from the DUCET, except for Й and й (U+0419 & U+0439 Cyrillic letter short i) and their decomposition mappings. This should improve performance of Cyrillic string comparisons and simplify tailorings.
  Existing per-language tailorings need to be adjusted: Appropriate contractions need to be added, and suppressions of default contractions that are no longer present can be removed.

UCA 7.0.0 from UCA 6.3.0 (or earlier)

- There are a number of clarifications to the text that people should revisit, to make sure that their understanding is correct. These are listed in the Modifications section.

UCA 6.3.0 from UCA 6.2.0 (or earlier)

- A claim of conformance to C6 (UCA parametric tailoring) from earlier versions of the Unicode Collation Algorithm is to be interpreted as a claim of conformance to LDML parametric tailoring. See *Setting Options* in [UTS35Collation].
- The IgnoreSP option for variable weighted characters has been removed. Implementers of this option may instead refer to CLDR Shifted behavior.
- U+FFFD is mapped to a collation element with a very high primary weight. This changes the behavior of ill-formed code unit sequences, if they are weighted as if they were U+FFFD. When using the Shifted option, ill-formed code unit are no longer ignored.
- Fourth-level weights have been removed from the DUCET. Parsers of allkeys.txt may need to be modified. If an implementation relies on the fourth-level weights, then they can be computed

according to the derivation described in UCA version 6.2.

- CLDR root collation data files have been moved from the UCA data directory (where they were combined into a CollationAuxiliary.zip) to the CLDR repository. See [UTS35Collation], *Root Collation Data Files*.

UCA 6.2.0 from UCA 6.1.0 (or earlier)

- There are a number of clarifications to the text that people should revisit, to make sure that their understanding is correct. These are listed in the modifications section.
- Users of the conformance test data files need to adjust their test code. For details see the CollationTest.html documentation file.

UCA 6.1.0 from UCA 6.0.0 (or earlier)

- A new IgnoreSP option for variable weighted characters has been added. Implementations may need to be updated to support this additional option.
- Another option for parametric tailoring, reorder, has been added. Although parametric tailoring is not a required feature of UCA, it is used by [UTS35Collation], and implementers should be aware of its implications.

UCA 6.0.0 from UCA 5.2.0 (or earlier)

- Ill-formed code unit sequences are no longer required to be mapped to [.0000.0000.0000] when not treated as an error; instead, implementations are strongly encouraged not to give them ignorable primary weights, for security reasons.
- Noncharacter code points are also no longer required to be mapped to [.0000.0000.0000], but are given implicit weights instead.
- The addition of a new range of CJK unified ideographs (Extension D) means that some implementations may need to change hard-coded ranges for ideographs.

UCA 5.2.0 from UCA 5.1.0 (or earlier)

- The clarification of implicit weight BASE values in *Section 10.1.3, Implicit Weights* means that any implementation which weighted unassigned code points in a CJK unified ideograph block as if they were CJK unified ideographs will need to change.
- The addition of a new range of CJK unified ideographs (Extension C) means that some implementations may need to change hard-coded ranges for ideographs.

Modifications

The following summarizes modifications from the previous version of this document.

Revision 54

- **Proposed Update** for Unicode 18.0.0.
- Added implicit weighting for Jurchen and Seal in *Table 10, Computing Implicit Weights*.
- Updated data file references to point to new locations for Version 18.0.0.
- Added explanation in Tibetan and Well-Formedness of DUCET regarding the addition of 10 Tibetan contractions to maintain the well-formedness of DUCET.
- Separated out a subsection of Reserved Weights and added an explanation of the special handling of weights for noncharacters U+FFFE and U+FFFF.
- Added note re abbreviation of "collation element" as "CE" in UTS10-D2.
- Adjusted text in Large Weight Values and Avoiding Normalization.
- The Shift-Trimmed option has been removed from *Section 4, Variable Weighting*.

Previous revisions can be accessed with the “Previous Version” link in the header.

© 1999–2026 Unicode, Inc. This publication is protected by copyright, and permission must be obtained from Unicode, Inc. prior to any reproduction, modification, or other use not permitted by the Terms of Use. Specifically, you may make copies of this publication and may annotate and translate it solely for personal or internal business purposes and not for public distribution, provided that any such permitted copies and modifications fully reproduce all copyright and other legal notices contained in the original. You may not make copies of or modifications to this publication for public distribution, or incorporate it in whole or in part into any product or publication without the express written permission of Unicode.

Use of all Unicode Products, including this publication, is governed by the Unicode Terms of Use. The authors, contributors, and publishers have taken care in the preparation of this publication, but make no express or implied representation or warranty of any kind and assume no responsibility or liability for errors or omissions or for consequential or incidental damages that may arise therefrom. This publication is provided “AS-IS” without charge as a convenience to users.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc., in the United States and other countries.