001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4;  
018  
019import java.util.AbstractList;  
020import java.util.ArrayList;  
021import java.util.Collection;  
022import java.util.Collections;  
023import java.util.HashSet;  
024import java.util.Iterator;  
025import java.util.List;  
026  
027import org.apache.commons.collections4.bag.HashBag;  
028import org.apache.commons.collections4.functors.DefaultEquator;  
029import org.apache.commons.collections4.list.FixedSizeList;  
030import org.apache.commons.collections4.list.LazyList;  
031import org.apache.commons.collections4.list.PredicatedList;  
032import org.apache.commons.collections4.list.TransformedList;  
033import org.apache.commons.collections4.list.UnmodifiableList;  
034import org.apache.commons.collections4.sequence.CommandVisitor;  
035import org.apache.commons.collections4.sequence.EditScript;  
036import org.apache.commons.collections4.sequence.SequencesComparator;  
037  
038/\*\*  
039 \* Provides utility methods and decorators for {@link List} instances.  
040 \*  
041 \* @since 1.0  
042 \*/  
043public class ListUtils {  
044  
045 /\*\*  
046 \* <code>ListUtils</code> should not normally be instantiated.  
047 \*/  
048 private ListUtils() {}  
049  
050 //-----------------------------------------------------------------------  
051  
052 /\*\*  
053 \* Returns an immutable empty list if the argument is <code>null</code>,  
054 \* or the argument itself otherwise.  
055 \*  
056 \* @param <T> the element type  
057 \* @param list the list, possibly <code>null</code>  
058 \* @return an empty list if the argument is <code>null</code>  
059 \*/  
060 public static <T> List<T> emptyIfNull(final List<T> list) {  
061 return list == null ? Collections.<T>emptyList() : list;  
062 }  
063  
064 /\*\*  
065 \* Returns either the passed in list, or if the list is {@code null},  
066 \* the value of {@code defaultList}.  
067 \*  
068 \* @param <T> the element type  
069 \* @param list the list, possibly {@code null}  
070 \* @param defaultList the returned values if list is {@code null}  
071 \* @return an empty list if the argument is <code>null</code>  
072 \* @since 4.0  
073 \*/  
074 public static <T> List<T> defaultIfNull(final List<T> list, final List<T> defaultList) {  
075 return list == null ? defaultList : list;  
076 }  
077  
078 /\*\*  
079 \* Returns a new list containing all elements that are contained in  
080 \* both given lists.  
081 \*  
082 \* @param <E> the element type  
083 \* @param list1 the first list  
084 \* @param list2 the second list  
085 \* @return the intersection of those two lists  
086 \* @throws NullPointerException if either list is null  
087 \*/  
088 public static <E> List<E> intersection(final List<? extends E> list1, final List<? extends E> list2) {  
089 final List<E> result = new ArrayList<>();  
090  
091 List<? extends E> smaller = list1;  
092 List<? extends E> larger = list2;  
093 if (list1.size() > list2.size()) {  
094 smaller = list2;  
095 larger = list1;  
096 }  
097  
098 final HashSet<E> hashSet = new HashSet<>(smaller);  
099  
100 for (final E e : larger) {  
101 if (hashSet.contains(e)) {  
102 result.add(e);  
103 hashSet.remove(e);  
104 }  
105 }  
106 return result;  
107 }  
108  
109 /\*\*  
110 \* Subtracts all elements in the second list from the first list,  
111 \* placing the results in a new list.  
112 \* <p>  
113 \* This differs from {@link List#removeAll(Collection)} in that  
114 \* cardinality is respected; if <Code>list1</Code> contains two  
115 \* occurrences of <Code>null</Code> and <Code>list2</Code> only  
116 \* contains one occurrence, then the returned list will still contain  
117 \* one occurrence.  
118 \*  
119 \* @param <E> the element type  
120 \* @param list1 the list to subtract from  
121 \* @param list2 the list to subtract  
122 \* @return a new list containing the results  
123 \* @throws NullPointerException if either list is null  
124 \*/  
125 public static <E> List<E> subtract(final List<E> list1, final List<? extends E> list2) {  
126 final ArrayList<E> result = new ArrayList<>();  
127 final HashBag<E> bag = new HashBag<>(list2);  
128 for (final E e : list1) {  
129 if (!bag.remove(e, 1)) {  
130 result.add(e);  
131 }  
132 }  
133 return result;  
134 }  
135  
136 /\*\*  
137 \* Returns the sum of the given lists. This is their intersection  
138 \* subtracted from their union.  
139 \*  
140 \* @param <E> the element type  
141 \* @param list1 the first list  
142 \* @param list2 the second list  
143 \* @return a new list containing the sum of those lists  
144 \* @throws NullPointerException if either list is null  
145 \*/  
146 public static <E> List<E> sum(final List<? extends E> list1, final List<? extends E> list2) {  
147 return subtract(union(list1, list2), intersection(list1, list2));  
148 }  
149  
150 /\*\*  
151 \* Returns a new list containing the second list appended to the  
152 \* first list. The {@link List#addAll(Collection)} operation is  
153 \* used to append the two given lists into a new list.  
154 \*  
155 \* @param <E> the element type  
156 \* @param list1 the first list  
157 \* @param list2 the second list  
158 \* @return a new list containing the union of those lists  
159 \* @throws NullPointerException if either list is null  
160 \*/  
161 public static <E> List<E> union(final List<? extends E> list1, final List<? extends E> list2) {  
162 final ArrayList<E> result = new ArrayList<>(list1.size() + list2.size());  
163 result.addAll(list1);  
164 result.addAll(list2);  
165 return result;  
166 }  
167  
168 /\*\*  
169 \* Selects all elements from input collection which match the given  
170 \* predicate into an output list.  
171 \* <p>  
172 \* A <code>null</code> predicate matches no elements.  
173 \*  
174 \* @param <E> the element type  
175 \* @param inputCollection the collection to get the input from, may not be null  
176 \* @param predicate the predicate to use, may be null  
177 \* @return the elements matching the predicate (new list)  
178 \* @throws NullPointerException if the input list is null  
179 \*  
180 \* @since 4.0  
181 \* @see CollectionUtils#select(Iterable, Predicate)  
182 \*/  
183 public static <E> List<E> select(final Collection<? extends E> inputCollection,  
184 final Predicate<? super E> predicate) {  
185 return CollectionUtils.select(inputCollection, predicate, new ArrayList<E>(inputCollection.size()));  
186 }  
187  
188 /\*\*  
189 \* Selects all elements from inputCollection which don't match the given  
190 \* predicate into an output collection.  
191 \* <p>  
192 \* If the input predicate is <code>null</code>, the result is an empty list.  
193 \*  
194 \* @param <E> the element type  
195 \* @param inputCollection the collection to get the input from, may not be null  
196 \* @param predicate the predicate to use, may be null  
197 \* @return the elements <b>not</b> matching the predicate (new list)  
198 \* @throws NullPointerException if the input collection is null  
199 \*  
200 \* @since 4.0  
201 \* @see CollectionUtils#selectRejected(Iterable, Predicate)  
202 \*/  
203 public static <E> List<E> selectRejected(final Collection<? extends E> inputCollection,  
204 final Predicate<? super E> predicate) {  
205 return CollectionUtils.selectRejected(inputCollection, predicate, new ArrayList<E>(inputCollection.size()));  
206 }  
207  
208 /\*\*  
209 \* Tests two lists for value-equality as per the equality contract in  
210 \* {@link java.util.List#equals(java.lang.Object)}.  
211 \* <p>  
212 \* This method is useful for implementing <code>List</code> when you cannot  
213 \* extend AbstractList. The method takes Collection instances to enable other  
214 \* collection types to use the List implementation algorithm.  
215 \* <p>  
216 \* The relevant text (slightly paraphrased as this is a static method) is:  
217 \* <blockquote>  
218 \* Compares the two list objects for equality. Returns  
219 \* {@code true} if and only if both  
220 \* lists have the same size, and all corresponding pairs of elements in  
221 \* the two lists are <i>equal</i>. (Two elements {@code e1} and  
222 \* {@code e2} are <i>equal</i> if <code>(e1==null ? e2==null :  
223 \* e1.equals(e2))</code>.) In other words, two lists are defined to be  
224 \* equal if they contain the same elements in the same order. This  
225 \* definition ensures that the equals method works properly across  
226 \* different implementations of the {@code List} interface.  
227 \* </blockquote>  
228 \*  
229 \* <b>Note:</b> The behaviour of this method is undefined if the lists are  
230 \* modified during the equals comparison.  
231 \*  
232 \* @see java.util.List  
233 \* @param list1 the first list, may be null  
234 \* @param list2 the second list, may be null  
235 \* @return whether the lists are equal by value comparison  
236 \*/  
237 public static boolean isEqualList(final Collection<?> list1, final Collection<?> list2) {  
238 if (list1 == list2) {  
239 return true;  
240 }  
241 if (list1 == null || list2 == null || list1.size() != list2.size()) {  
242 return false;  
243 }  
244  
245 final Iterator<?> it1 = list1.iterator();  
246 final Iterator<?> it2 = list2.iterator();  
247 Object obj1 = null;  
248 Object obj2 = null;  
249  
250 while (it1.hasNext() && it2.hasNext()) {  
251 obj1 = it1.next();  
252 obj2 = it2.next();  
253  
254 if (!(obj1 == null ? obj2 == null : obj1.equals(obj2))) {  
255 return false;  
256 }  
257 }  
258  
259 return !(it1.hasNext() || it2.hasNext());  
260 }  
261  
262 /\*\*  
263 \* Generates a hash code using the algorithm specified in  
264 \* {@link java.util.List#hashCode()}.  
265 \* <p>  
266 \* This method is useful for implementing <code>List</code> when you cannot  
267 \* extend AbstractList. The method takes Collection instances to enable other  
268 \* collection types to use the List implementation algorithm.  
269 \*  
270 \* @see java.util.List#hashCode()  
271 \* @param list the list to generate the hashCode for, may be null  
272 \* @return the hash code  
273 \*/  
274 public static int hashCodeForList(final Collection<?> list) {  
275 if (list == null) {  
276 return 0;  
277 }  
278 int hashCode = 1;  
279 final Iterator<?> it = list.iterator();  
280  
281 while (it.hasNext()) {  
282 final Object obj = it.next();  
283 hashCode = 31 \* hashCode + (obj == null ? 0 : obj.hashCode());  
284 }  
285 return hashCode;  
286 }  
287  
288 //-----------------------------------------------------------------------  
289 /\*\*  
290 \* Returns a List containing all the elements in <code>collection</code>  
291 \* that are also in <code>retain</code>. The cardinality of an element <code>e</code>  
292 \* in the returned list is the same as the cardinality of <code>e</code>  
293 \* in <code>collection</code> unless <code>retain</code> does not contain <code>e</code>, in which  
294 \* case the cardinality is zero. This method is useful if you do not wish to modify  
295 \* the collection <code>c</code> and thus cannot call <code>collection.retainAll(retain);</code>.  
296 \* <p>  
297 \* This implementation iterates over <code>collection</code>, checking each element in  
298 \* turn to see if it's contained in <code>retain</code>. If it's contained, it's added  
299 \* to the returned list. As a consequence, it is advised to use a collection type for  
300 \* <code>retain</code> that provides a fast (e.g. O(1)) implementation of  
301 \* {@link Collection#contains(Object)}.  
302 \*  
303 \* @param <E> the element type  
304 \* @param collection the collection whose contents are the target of the #retailAll operation  
305 \* @param retain the collection containing the elements to be retained in the returned collection  
306 \* @return a <code>List</code> containing all the elements of <code>c</code>  
307 \* that occur at least once in <code>retain</code>.  
308 \* @throws NullPointerException if either parameter is null  
309 \* @since 3.2  
310 \*/  
311 public static <E> List<E> retainAll(final Collection<E> collection, final Collection<?> retain) {  
312 final List<E> list = new ArrayList<>(Math.min(collection.size(), retain.size()));  
313  
314 for (final E obj : collection) {  
315 if (retain.contains(obj)) {  
316 list.add(obj);  
317 }  
318 }  
319 return list;  
320 }  
321  
322 /\*\*  
323 \* Removes the elements in <code>remove</code> from <code>collection</code>. That is, this  
324 \* method returns a list containing all the elements in <code>collection</code>  
325 \* that are not in <code>remove</code>. The cardinality of an element <code>e</code>  
326 \* in the returned collection is the same as the cardinality of <code>e</code>  
327 \* in <code>collection</code> unless <code>remove</code> contains <code>e</code>, in which  
328 \* case the cardinality is zero. This method is useful if you do not wish to modify  
329 \* <code>collection</code> and thus cannot call <code>collection.removeAll(remove);</code>.  
330 \* <p>  
331 \* This implementation iterates over <code>collection</code>, checking each element in  
332 \* turn to see if it's contained in <code>remove</code>. If it's not contained, it's added  
333 \* to the returned list. As a consequence, it is advised to use a collection type for  
334 \* <code>remove</code> that provides a fast (e.g. O(1)) implementation of  
335 \* {@link Collection#contains(Object)}.  
336 \*  
337 \* @param <E> the element type  
338 \* @param collection the collection from which items are removed (in the returned collection)  
339 \* @param remove the items to be removed from the returned <code>collection</code>  
340 \* @return a <code>List</code> containing all the elements of <code>c</code> except  
341 \* any elements that also occur in <code>remove</code>.  
342 \* @throws NullPointerException if either parameter is null  
343 \* @since 3.2  
344 \*/  
345 public static <E> List<E> removeAll(final Collection<E> collection, final Collection<?> remove) {  
346 final List<E> list = new ArrayList<>();  
347 for (final E obj : collection) {  
348 if (!remove.contains(obj)) {  
349 list.add(obj);  
350 }  
351 }  
352 return list;  
353 }  
354  
355 //-----------------------------------------------------------------------  
356 /\*\*  
357 \* Returns a synchronized list backed by the given list.  
358 \* <p>  
359 \* You must manually synchronize on the returned list's iterator to  
360 \* avoid non-deterministic behavior:  
361 \*  
362 \* <pre>  
363 \* List list = ListUtils.synchronizedList(myList);  
364 \* synchronized (list) {  
365 \* Iterator i = list.iterator();  
366 \* while (i.hasNext()) {  
367 \* process (i.next());  
368 \* }  
369 \* }  
370 \* </pre>  
371 \*  
372 \* This method is just a wrapper for {@link Collections#synchronizedList(List)}.  
373 \*  
374 \* @param <E> the element type  
375 \* @param list the list to synchronize, must not be null  
376 \* @return a synchronized list backed by the given list  
377 \* @throws NullPointerException if the list is null  
378 \*/  
379 public static <E> List<E> synchronizedList(final List<E> list) {  
380 return Collections.synchronizedList(list);  
381 }  
382  
383 /\*\*  
384 \* Returns an unmodifiable list backed by the given list.  
385 \* <p>  
386 \* This method uses the implementation in the decorators subpackage.  
387 \*  
388 \* @param <E> the element type  
389 \* @param list the list to make unmodifiable, must not be null  
390 \* @return an unmodifiable list backed by the given list  
391 \* @throws NullPointerException if the list is null  
392 \*/  
393 public static <E> List<E> unmodifiableList(final List<? extends E> list) {  
394 return UnmodifiableList.unmodifiableList(list);  
395 }  
396  
397 /\*\*  
398 \* Returns a predicated (validating) list backed by the given list.  
399 \* <p>  
400 \* Only objects that pass the test in the given predicate can be added to the list.  
401 \* Trying to add an invalid object results in an IllegalArgumentException.  
402 \* It is important not to use the original list after invoking this method,  
403 \* as it is a backdoor for adding invalid objects.  
404 \*  
405 \* @param <E> the element type  
406 \* @param list the list to predicate, must not be null  
407 \* @param predicate the predicate for the list, must not be null  
408 \* @return a predicated list backed by the given list  
409 \* @throws NullPointerException if the List or Predicate is null  
410 \*/  
411 public static <E> List<E> predicatedList(final List<E> list, final Predicate<E> predicate) {  
412 return PredicatedList.predicatedList(list, predicate);  
413 }  
414  
415 /\*\*  
416 \* Returns a transformed list backed by the given list.  
417 \* <p>  
418 \* This method returns a new list (decorating the specified list) that  
419 \* will transform any new entries added to it.  
420 \* Existing entries in the specified list will not be transformed.  
421 \* <p>  
422 \* Each object is passed through the transformer as it is added to the  
423 \* List. It is important not to use the original list after invoking this  
424 \* method, as it is a backdoor for adding untransformed objects.  
425 \* <p>  
426 \* Existing entries in the specified list will not be transformed.  
427 \* If you want that behaviour, see {@link TransformedList#transformedList}.  
428 \*  
429 \* @param <E> the element type  
430 \* @param list the list to predicate, must not be null  
431 \* @param transformer the transformer for the list, must not be null  
432 \* @return a transformed list backed by the given list  
433 \* @throws NullPointerException if the List or Transformer is null  
434 \*/  
435 public static <E> List<E> transformedList(final List<E> list,  
436 final Transformer<? super E, ? extends E> transformer) {  
437 return TransformedList.transformingList(list, transformer);  
438 }  
439  
440 /\*\*  
441 \* Returns a "lazy" list whose elements will be created on demand.  
442 \* <p>  
443 \* When the index passed to the returned list's {@link List#get(int) get}  
444 \* method is greater than the list's size, then the factory will be used  
445 \* to create a new object and that object will be inserted at that index.  
446 \* <p>  
447 \* For instance:  
448 \*  
449 \* <pre>  
450 \* Factory<Date> factory = new Factory<Date>() {  
451 \* public Date create() {  
452 \* return new Date();  
453 \* }  
454 \* }  
455 \* List<Date> lazy = ListUtils.lazyList(new ArrayList<Date>(), factory);  
456 \* Date date = lazy.get(3);  
457 \* </pre>  
458 \*  
459 \* After the above code is executed, <code>date</code> will refer to  
460 \* a new <code>Date</code> instance. Furthermore, that <code>Date</code>  
461 \* instance is the fourth element in the list. The first, second,  
462 \* and third element are all set to <code>null</code>.  
463 \*  
464 \* @param <E> the element type  
465 \* @param list the list to make lazy, must not be null  
466 \* @param factory the factory for creating new objects, must not be null  
467 \* @return a lazy list backed by the given list  
468 \* @throws NullPointerException if the List or Factory is null  
469 \*/  
470 public static <E> List<E> lazyList(final List<E> list, final Factory<? extends E> factory) {  
471 return LazyList.lazyList(list, factory);  
472 }  
473  
474 /\*\*  
475 \* Returns a "lazy" list whose elements will be created on demand.  
476 \* <p>  
477 \* When the index passed to the returned list's {@link List#get(int) get}  
478 \* method is greater than the list's size, then the transformer will be used  
479 \* to create a new object and that object will be inserted at that index.  
480 \* <p>  
481 \* For instance:  
482 \*  
483 \* <pre>  
484 \* List<Integer> hours = Arrays.asList(7, 5, 8, 2);  
485 \* Transformer<Integer,Date> transformer = input -> LocalDateTime.now().withHour(hours.get(input));  
486 \* List<LocalDateTime> lazy = ListUtils.lazyList(new ArrayList<LocalDateTime>(), transformer);  
487 \* Date date = lazy.get(3);  
488 \* </pre>  
489 \*  
490 \* After the above code is executed, <code>date</code> will refer to  
491 \* a new <code>Date</code> instance. Furthermore, that <code>Date</code>  
492 \* instance is the fourth element in the list. The first, second,  
493 \* and third element are all set to <code>null</code>.  
494 \*  
495 \* @param <E> the element type  
496 \* @param list the list to make lazy, must not be null  
497 \* @param transformer the transformer for creating new objects, must not be null  
498 \* @return a lazy list backed by the given list  
499 \* @throws NullPointerException if the List or Transformer is null  
500 \*/  
501 public static <E> List<E> lazyList(final List<E> list, final Transformer<Integer, ? extends E> transformer) {  
502 return LazyList.lazyList(list, transformer);  
503 }  
504  
505 /\*\*  
506 \* Returns a fixed-sized list backed by the given list.  
507 \* Elements may not be added or removed from the returned list, but  
508 \* existing elements can be changed (for instance, via the  
509 \* {@link List#set(int, Object)} method).  
510 \*  
511 \* @param <E> the element type  
512 \* @param list the list whose size to fix, must not be null  
513 \* @return a fixed-size list backed by that list  
514 \* @throws NullPointerException if the List is null  
515 \*/  
516 public static <E> List<E> fixedSizeList(final List<E> list) {  
517 return FixedSizeList.fixedSizeList(list);  
518 }  
519  
520 //-----------------------------------------------------------------------  
521 /\*\*  
522 \* Finds the first index in the given List which matches the given predicate.  
523 \* <p>  
524 \* If the input List or predicate is null, or no element of the List  
525 \* matches the predicate, -1 is returned.  
526 \*  
527 \* @param <E> the element type  
528 \* @param list the List to search, may be null  
529 \* @param predicate the predicate to use, may be null  
530 \* @return the first index of an Object in the List which matches the predicate or -1 if none could be found  
531 \*/  
532 public static <E> int indexOf(final List<E> list, final Predicate<E> predicate) {  
533 if (list != null && predicate != null) {  
534 for (int i = 0; i < list.size(); i++) {  
535 final E item = list.get(i);  
536 if (predicate.evaluate(item)) {  
537 return i;  
538 }  
539 }  
540 }  
541 return -1;  
542 }  
543  
544 //-----------------------------------------------------------------------  
545 /\*\*  
546 \* Returns the longest common subsequence (LCS) of two sequences (lists).  
547 \*  
548 \* @param <E> the element type  
549 \* @param a the first list  
550 \* @param b the second list  
551 \* @return the longest common subsequence  
552 \* @throws NullPointerException if either list is {@code null}  
553 \* @since 4.0  
554 \*/  
555 public static <E> List<E> longestCommonSubsequence(final List<E> a, final List<E> b) {  
556 return longestCommonSubsequence( a, b, DefaultEquator.defaultEquator() );  
557 }  
558  
559 /\*\*  
560 \* Returns the longest common subsequence (LCS) of two sequences (lists).  
561 \*  
562 \* @param <E> the element type  
563 \* @param a the first list  
564 \* @param b the second list  
565 \* @param equator the equator used to test object equality  
566 \* @return the longest common subsequence  
567 \* @throws NullPointerException if either list or the equator is {@code null}  
568 \* @since 4.0  
569 \*/  
570 public static <E> List<E> longestCommonSubsequence(final List<E> a, final List<E> b,  
571 final Equator<? super E> equator) {  
572 if (a == null || b == null) {  
573 throw new NullPointerException("List must not be null");  
574 }  
575 if (equator == null) {  
576 throw new NullPointerException("Equator must not be null");  
577 }  
578  
579 final SequencesComparator<E> comparator = new SequencesComparator<>(a, b, equator);  
580 final EditScript<E> script = comparator.getScript();  
581 final LcsVisitor<E> visitor = new LcsVisitor<>();  
582 script.visit(visitor);  
583 return visitor.getSubSequence();  
584 }  
585  
586 /\*\*  
587 \* Returns the longest common subsequence (LCS) of two {@link CharSequence} objects.  
588 \* <p>  
589 \* This is a convenience method for using {@link #longestCommonSubsequence(List, List)}  
590 \* with {@link CharSequence} instances.  
591 \*  
592 \* @param a the first sequence  
593 \* @param b the second sequence  
594 \* @return the longest common subsequence as {@link String}  
595 \* @throws NullPointerException if either sequence is {@code null}  
596 \* @since 4.0  
597 \*/  
598 public static String longestCommonSubsequence(final CharSequence a, final CharSequence b) {  
599 if (a == null || b == null) {  
600 throw new NullPointerException("CharSequence must not be null");  
601 }  
602 final List<Character> lcs = longestCommonSubsequence(new CharSequenceAsList( a ), new CharSequenceAsList( b ));  
603 final StringBuilder sb = new StringBuilder();  
604 for ( final Character ch : lcs ) {  
605 sb.append(ch);  
606 }  
607 return sb.toString();  
608 }  
609  
610 /\*\*  
611 \* A helper class used to construct the longest common subsequence.  
612 \*/  
613 private static final class LcsVisitor<E> implements CommandVisitor<E> {  
614 private final ArrayList<E> sequence;  
615  
616 public LcsVisitor() {  
617 sequence = new ArrayList<>();  
618 }  
619  
620 @Override  
621 public void visitInsertCommand(final E object) {}  
622  
623 @Override  
624 public void visitDeleteCommand(final E object) {}  
625  
626 @Override  
627 public void visitKeepCommand(final E object) {  
628 sequence.add(object);  
629 }  
630  
631 public List<E> getSubSequence() {  
632 return sequence;  
633 }  
634 }  
635  
636 /\*\*  
637 \* A simple wrapper to use a CharSequence as List.  
638 \*/  
639 private static final class CharSequenceAsList extends AbstractList<Character> {  
640  
641 private final CharSequence sequence;  
642  
643 public CharSequenceAsList(final CharSequence sequence) {  
644 this.sequence = sequence;  
645 }  
646  
647 @Override  
648 public Character get( final int index ) {  
649 return Character.valueOf(sequence.charAt( index ));  
650 }  
651  
652 @Override  
653 public int size() {  
654 return sequence.length();  
655 }  
656  
657 }  
658  
659 //-----------------------------------------------------------------------  
660 /\*\*  
661 \* Returns consecutive {@link List#subList(int, int) sublists} of a  
662 \* list, each of the same size (the final list may be smaller). For example,  
663 \* partitioning a list containing {@code [a, b, c, d, e]} with a partition  
664 \* size of 3 yields {@code [[a, b, c], [d, e]]} -- an outer list containing  
665 \* two inner lists of three and two elements, all in the original order.  
666 \* <p>  
667 \* The outer list is unmodifiable, but reflects the latest state of the  
668 \* source list. The inner lists are sublist views of the original list,  
669 \* produced on demand using {@link List#subList(int, int)}, and are subject  
670 \* to all the usual caveats about modification as explained in that API.  
671 \* <p>  
672 \* Adapted from http://code.google.com/p/guava-libraries/  
673 \*  
674 \* @param <T> the element type  
675 \* @param list the list to return consecutive sublists of  
676 \* @param size the desired size of each sublist (the last may be smaller)  
677 \* @return a list of consecutive sublists  
678 \* @throws NullPointerException if list is null  
679 \* @throws IllegalArgumentException if size is not strictly positive  
680 \* @since 4.0  
681 \*/  
682 public static <T> List<List<T>> partition(final List<T> list, final int size) {  
683 if (list == null) {  
684 throw new NullPointerException("List must not be null");  
685 }  
686 if (size <= 0) {  
687 throw new IllegalArgumentException("Size must be greater than 0");  
688 }  
689 return new Partition<>(list, size);  
690 }  
691  
692 /\*\*  
693 \* Provides a partition view on a {@link List}.  
694 \* @since 4.0  
695 \*/  
696 private static class Partition<T> extends AbstractList<List<T>> {  
697 private final List<T> list;  
698 private final int size;  
699  
700 private Partition(final List<T> list, final int size) {  
701 this.list = list;  
702 this.size = size;  
703 }  
704  
705 @Override  
706 public List<T> get(final int index) {  
707 final int listSize = size();  
708 if (index < 0) {  
709 throw new IndexOutOfBoundsException("Index " + index + " must not be negative");  
710 }  
711 if (index >= listSize) {  
712 throw new IndexOutOfBoundsException("Index " + index + " must be less than size " +  
713 listSize);  
714 }  
715 final int start = index \* size;  
716 final int end = Math.min(start + size, list.size());  
717 return list.subList(start, end);  
718 }  
719  
720 @Override  
721 public int size() {  
722 return (int) Math.ceil((double) list.size() / (double) size);  
723 }  
724  
725 @Override  
726 public boolean isEmpty() {  
727 return list.isEmpty();  
728 }  
729 }  
730}