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.sequence;  
018  
019import java.util.List;  
020  
021import org.apache.commons.collections4.Equator;  
022import org.apache.commons.collections4.functors.DefaultEquator;  
023  
024/\*\*  
025 \* This class allows to compare two objects sequences.  
026 \* <p>  
027 \* The two sequences can hold any object type, as only the <code>equals</code>  
028 \* method is used to compare the elements of the sequences. It is guaranteed  
029 \* that the comparisons will always be done as <code>o1.equals(o2)</code> where  
030 \* <code>o1</code> belongs to the first sequence and <code>o2</code> belongs to  
031 \* the second sequence. This can be important if subclassing is used for some  
032 \* elements in the first sequence and the <code>equals</code> method is  
033 \* specialized.  
034 \* </p>  
035 \* <p>  
036 \* Comparison can be seen from two points of view: either as giving the smallest  
037 \* modification allowing to transform the first sequence into the second one, or  
038 \* as giving the longest sequence which is a subsequence of both initial  
039 \* sequences. The <code>equals</code> method is used to compare objects, so any  
040 \* object can be put into sequences. Modifications include deleting, inserting  
041 \* or keeping one object, starting from the beginning of the first sequence.  
042 \* </p>  
043 \* <p>  
044 \* This class implements the comparison algorithm, which is the very efficient  
045 \* algorithm from Eugene W. Myers  
046 \* <a href="http://www.cis.upenn.edu/~bcpierce/courses/dd/papers/diff.ps">  
047 \* An O(ND) Difference Algorithm and Its Variations</a>. This algorithm produces  
048 \* the shortest possible  
049 \* {@link EditScript edit script}  
050 \* containing all the  
051 \* {@link EditCommand commands}  
052 \* needed to transform the first sequence into the second one.  
053 \* </p>  
054 \*  
055 \* @see EditScript  
056 \* @see EditCommand  
057 \* @see CommandVisitor  
058 \*  
059 \* @since 4.0  
060 \*/  
061public class SequencesComparator<T> {  
062  
063 /\*\* First sequence. \*/  
064 private final List<T> sequence1;  
065  
066 /\*\* Second sequence. \*/  
067 private final List<T> sequence2;  
068  
069 /\*\* The equator used for testing object equality. \*/  
070 private final Equator<? super T> equator;  
071  
072 /\*\* Temporary variables. \*/  
073 private final int[] vDown;  
074 private final int[] vUp;  
075  
076 /\*\*  
077 \* Simple constructor.  
078 \* <p>  
079 \* Creates a new instance of SequencesComparator using a {@link DefaultEquator}.  
080 \* <p>  
081 \* It is <em>guaranteed</em> that the comparisons will always be done as  
082 \* <code>o1.equals(o2)</code> where <code>o1</code> belongs to the first  
083 \* sequence and <code>o2</code> belongs to the second sequence. This can be  
084 \* important if subclassing is used for some elements in the first sequence  
085 \* and the <code>equals</code> method is specialized.  
086 \*  
087 \* @param sequence1 first sequence to be compared  
088 \* @param sequence2 second sequence to be compared  
089 \*/  
090 public SequencesComparator(final List<T> sequence1, final List<T> sequence2) {  
091 this(sequence1, sequence2, DefaultEquator.defaultEquator());  
092 }  
093  
094 /\*\*  
095 \* Simple constructor.  
096 \* <p>  
097 \* Creates a new instance of SequencesComparator with a custom {@link Equator}.  
098 \* <p>  
099 \* It is <em>guaranteed</em> that the comparisons will always be done as  
100 \* <code>Equator.equate(o1, o2)</code> where <code>o1</code> belongs to the first  
101 \* sequence and <code>o2</code> belongs to the second sequence.  
102 \*  
103 \* @param sequence1 first sequence to be compared  
104 \* @param sequence2 second sequence to be compared  
105 \* @param equator the equator to use for testing object equality  
106 \*/  
107 public SequencesComparator(final List<T> sequence1, final List<T> sequence2, final Equator<? super T> equator) {  
108 this.sequence1 = sequence1;  
109 this.sequence2 = sequence2;  
110 this.equator = equator;  
111  
112 final int size = sequence1.size() + sequence2.size() + 2;  
113 vDown = new int[size];  
114 vUp = new int[size];  
115 }  
116  
117 /\*\*  
118 \* Get the {@link EditScript} object.  
119 \* <p>  
120 \* It is guaranteed that the objects embedded in the {@link InsertCommand  
121 \* insert commands} come from the second sequence and that the objects  
122 \* embedded in either the {@link DeleteCommand delete commands} or  
123 \* {@link KeepCommand keep commands} come from the first sequence. This can  
124 \* be important if subclassing is used for some elements in the first  
125 \* sequence and the <code>equals</code> method is specialized.  
126 \*  
127 \* @return the edit script resulting from the comparison of the two  
128 \* sequences  
129 \*/  
130 public EditScript<T> getScript() {  
131 final EditScript<T> script = new EditScript<>();  
132 buildScript(0, sequence1.size(), 0, sequence2.size(), script);  
133 return script;  
134 }  
135  
136 /\*\*  
137 \* Build a snake.  
138 \*  
139 \* @param start the value of the start of the snake  
140 \* @param diag the value of the diagonal of the snake  
141 \* @param end1 the value of the end of the first sequence to be compared  
142 \* @param end2 the value of the end of the second sequence to be compared  
143 \* @return the snake built  
144 \*/  
145 private Snake buildSnake(final int start, final int diag, final int end1, final int end2) {  
146 int end = start;  
147 while (end - diag < end2  
148 && end < end1  
149 && equator.equate(sequence1.get(end), sequence2.get(end - diag))) {  
150 ++end;  
151 }  
152 return new Snake(start, end, diag);  
153 }  
154  
155 /\*\*  
156 \* Get the middle snake corresponding to two subsequences of the  
157 \* main sequences.  
158 \* <p>  
159 \* The snake is found using the MYERS Algorithm (this algorithms has  
160 \* also been implemented in the GNU diff program). This algorithm is  
161 \* explained in Eugene Myers article:  
162 \* <a href="http://www.cs.arizona.edu/people/gene/PAPERS/diff.ps">  
163 \* An O(ND) Difference Algorithm and Its Variations</a>.  
164 \*  
165 \* @param start1 the begin of the first sequence to be compared  
166 \* @param end1 the end of the first sequence to be compared  
167 \* @param start2 the begin of the second sequence to be compared  
168 \* @param end2 the end of the second sequence to be compared  
169 \* @return the middle snake  
170 \*/  
171 private Snake getMiddleSnake(final int start1, final int end1, final int start2, final int end2) {  
172 // Myers Algorithm  
173 // Initialisations  
174 final int m = end1 - start1;  
175 final int n = end2 - start2;  
176 if (m == 0 || n == 0) {  
177 return null;  
178 }  
179  
180 final int delta = m - n;  
181 final int sum = n + m;  
182 final int offset = (sum % 2 == 0 ? sum : sum + 1) / 2;  
183 vDown[1+offset] = start1;  
184 vUp[1+offset] = end1 + 1;  
185  
186 for (int d = 0; d <= offset ; ++d) {  
187 // Down  
188 for (int k = -d; k <= d; k += 2) {  
189 // First step  
190  
191 final int i = k + offset;  
192 if (k == -d || k != d && vDown[i-1] < vDown[i+1]) {  
193 vDown[i] = vDown[i+1];  
194 } else {  
195 vDown[i] = vDown[i-1] + 1;  
196 }  
197  
198 int x = vDown[i];  
199 int y = x - start1 + start2 - k;  
200  
201 while (x < end1 && y < end2 && equator.equate(sequence1.get(x), sequence2.get(y))) {  
202 vDown[i] = ++x;  
203 ++y;  
204 }  
205 // Second step  
206 if (delta % 2 != 0 && delta - d <= k && k <= delta + d) {  
207 if (vUp[i-delta] <= vDown[i]) { // NOPMD  
208 return buildSnake(vUp[i-delta], k + start1 - start2, end1, end2);  
209 }  
210 }  
211 }  
212  
213 // Up  
214 for (int k = delta - d; k <= delta + d; k += 2) {  
215 // First step  
216 final int i = k + offset - delta;  
217 if (k == delta - d  
218 || k != delta + d && vUp[i+1] <= vUp[i-1]) {  
219 vUp[i] = vUp[i+1] - 1;  
220 } else {  
221 vUp[i] = vUp[i-1];  
222 }  
223  
224 int x = vUp[i] - 1;  
225 int y = x - start1 + start2 - k;  
226 while (x >= start1 && y >= start2  
227 && equator.equate(sequence1.get(x), sequence2.get(y))) {  
228 vUp[i] = x--;  
229 y--;  
230 }  
231 // Second step  
232 if (delta % 2 == 0 && -d <= k && k <= d ) {  
233 if (vUp[i] <= vDown[i + delta]) { // NOPMD  
234 return buildSnake(vUp[i], k + start1 - start2, end1, end2);  
235 }  
236 }  
237 }  
238 }  
239  
240 // this should not happen  
241 throw new RuntimeException("Internal Error");  
242 }  
243  
244  
245 /\*\*  
246 \* Build an edit script.  
247 \*  
248 \* @param start1 the begin of the first sequence to be compared  
249 \* @param end1 the end of the first sequence to be compared  
250 \* @param start2 the begin of the second sequence to be compared  
251 \* @param end2 the end of the second sequence to be compared  
252 \* @param script the edited script  
253 \*/  
254 private void buildScript(final int start1, final int end1, final int start2, final int end2,  
255 final EditScript<T> script) {  
256  
257 final Snake middle = getMiddleSnake(start1, end1, start2, end2);  
258  
259 if (middle == null  
260 || middle.getStart() == end1 && middle.getDiag() == end1 - end2  
261 || middle.getEnd() == start1 && middle.getDiag() == start1 - start2) {  
262  
263 int i = start1;  
264 int j = start2;  
265 while (i < end1 || j < end2) {  
266 if (i < end1 && j < end2 && equator.equate(sequence1.get(i), sequence2.get(j))) {  
267 script.append(new KeepCommand<>(sequence1.get(i)));  
268 ++i;  
269 ++j;  
270 } else {  
271 if (end1 - start1 > end2 - start2) {  
272 script.append(new DeleteCommand<>(sequence1.get(i)));  
273 ++i;  
274 } else {  
275 script.append(new InsertCommand<>(sequence2.get(j)));  
276 ++j;  
277 }  
278 }  
279 }  
280  
281 } else {  
282  
283 buildScript(start1, middle.getStart(),  
284 start2, middle.getStart() - middle.getDiag(),  
285 script);  
286 for (int i = middle.getStart(); i < middle.getEnd(); ++i) {  
287 script.append(new KeepCommand<>(sequence1.get(i)));  
288 }  
289 buildScript(middle.getEnd(), end1,  
290 middle.getEnd() - middle.getDiag(), end2,  
291 script);  
292 }  
293 }  
294  
295 /\*\*  
296 \* This class is a simple placeholder to hold the end part of a path  
297 \* under construction in a {@link SequencesComparator SequencesComparator}.  
298 \*/  
299 private static class Snake {  
300  
301 /\*\* Start index. \*/  
302 private final int start;  
303  
304 /\*\* End index. \*/  
305 private final int end;  
306  
307 /\*\* Diagonal number. \*/  
308 private final int diag;  
309  
310 /\*\*  
311 \* Simple constructor. Creates a new instance of Snake with specified indices.  
312 \*  
313 \* @param start start index of the snake  
314 \* @param end end index of the snake  
315 \* @param diag diagonal number  
316 \*/  
317 public Snake(final int start, final int end, final int diag) {  
318 this.start = start;  
319 this.end = end;  
320 this.diag = diag;  
321 }  
322  
323 /\*\*  
324 \* Get the start index of the snake.  
325 \*  
326 \* @return start index of the snake  
327 \*/  
328 public int getStart() {  
329 return start;  
330 }  
331  
332 /\*\*  
333 \* Get the end index of the snake.  
334 \*  
335 \* @return end index of the snake  
336 \*/  
337 public int getEnd() {  
338 return end;  
339 }  
340  
341 /\*\*  
342 \* Get the diagonal number of the snake.  
343 \*  
344 \* @return diagonal number of the snake  
345 \*/  
346 public int getDiag() {  
347 return diag;  
348 }  
349 }  
350}