EXHIBIT E

```
 1 /*
 2  * Copyright (C) 2008 The Android Open Source Project
 3  *
 4  * Licensed under the Apache License, Version 2.0 (the "License");
 5  * you may not use this file except in compliance with the License.
 6  * You may obtain a copy of the License at
 7  *
 8  *      http://www.apache.org/licenses/LICENSE-2.0
 9  *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  */
16
17 package java.util;
18
19 /**
20  * This is a near duplicate of {@link TimSort}, modified for use with
21  * arrays of objects that implement {@link Comparable}, instead of using
22  * explicit comparators.
23  *
24  * <p>If you are using an optimizing VM, you may find that
    ComparableTimSort
25  * offers no performance benefit over TimSort in conjunction with a
26  * comparator that simply returns {@code
    ((Comparable)first).compareTo(Second)}.
27  * If this is the case, you are better off deleting ComparableTimSort to
28  * eliminate the code duplication.  (See Arrays.java for details.)
29  */
30 class ComparableTimSort {
31     /**
32      * This is the minimum sized sequence that will be merged.  Shorter
33      * sequences will be lengthened by calling binarySort.  If the
        entire
34      * array is less than this length, no merges will be performed.
35      *
36      * This constant should be a power of two.  It was 64 in Tim Peter's
        C
37      * implementation, but 32 was empirically determined to work better
        in
38      * this implementation.  In the unlikely event that you set this
        constant
39      * to be a number that's not a power of two, you'll need to change
        the
40      * {@link #minRunLength} computation.
41      *
42      * If you decrease this constant, you must change the stackLen
43      * computation in the TimSort constructor, or you risk an
44      * ArrayOutOfBounds exception.  See listsort.txt for a discussion
45      * of the minimum stack length required as a function of the length
46      * of the array being sorted and the minimum merge sequence length.
47      */
48     private static final int MIN_MERGE = 32;
49
50     /**
51      * The array being sorted.
52      */
```

```
    private final Object[] a;

    /**
     * When we get into galloping mode, we stay there until both runs win less
     * often than MIN_GALLOP consecutive times.
     */
    private static final int  MIN_GALLOP = 7;

    /**
     * This controls when we get *into* galloping mode.  It is initialized
     * to MIN_GALLOP.  The mergeLo and mergeHi methods nudge it higher for
     * random data, and lower for highly structured data.
     */
    private int minGallop = MIN_GALLOP;

    /**
     * Maximum initial size of tmp array, which is used for merging. The array
     * can grow to accommodate demand.
     *
     * Unlike Tim's original C version, we do not allocate this much storage
     * when sorting smaller arrays.  This change was required for performance.
     */
    private static final int INITIAL_TMP_STORAGE_LENGTH = 256;

    /**
     * Temp storage for merges.
     */
    private Object[] tmp;

    /**
     * A stack of pending runs yet to be merged.  Run i starts at
     * address base[i] and extends for len[i] elements.  It's always
     * true (so long as the indices are in bounds) that:
     *
     *     runBase[i] + runLen[i] == runBase[i + 1]
     *
     * so we could cut the storage for this, but it's a minor amount,
     * and keeping all the info explicit simplifies the code.
     */
    private int stackSize = 0;  // Number of pending runs on stack
    private final int[] runBase;
    private final int[] runLen;

    /**
     * Asserts have been placed in if-statements for performace. To enable them,
     * set this field to true and enable them in VM with a command line flag.
     * If you modify this class, please do test the asserts!
     */
    private static final boolean DEBUG = false;

    /**
     * Creates a TimSort instance to maintain the state of an ongoing
```

```
         sort.
105      *
106      * @param a the array to be sorted
107      */
108     private ComparableTimSort(Object[] a) {
109         this.a = a;
110
111         // Allocate temp storage (which may be increased later if
            necessary)
112         int len = a.length;
113         @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
114         Object[] newArray = new Object[len < 2 *
            INITIAL_TMP_STORAGE_LENGTH ?
115                                        len >>> 1 :
                                           INITIAL_TMP_STORAGE_LENGTH];
116         tmp = newArray;
117
118         /*
119          * Allocate runs-to-be-merged stack (which cannot be expanded).
             The
120          * stack length requirements are described in listsort.txt.  The
             C
121          * version always uses the same stack length (85), but this was
122          * measured to be too expensive when sorting "mid-sized" arrays
             (e.g.,
123          * 100 elements) in Java.  Therefore, we use smaller (but
             sufficiently
124          * large) stack lengths for smaller arrays.  The "magic numbers"
             in the
125          * computation below must be changed if MIN_MERGE is decreased.
             See
126          * the MIN_MERGE declaration above for more information.
127          */
128         int stackLen = (len <    120  ?  5 :
129                         len <   1542  ? 10 :
130                         len < 119151  ? 19 : 40);
131         runBase = new int[stackLen];
132         runLen = new int[stackLen];
133     }
134
135     /*
136      * The next two methods (which are package private and static)
         constitute
137      * the entire API of this class.  Each of these methods obeys the
         contract
138      * of the public method with the same signature in java.util.Arrays.
139      */
140
141     static void sort(Object[] a) {
142          sort(a, 0, a.length);
143     }
144
145     static void sort(Object[] a, int lo, int hi) {
146         rangeCheck(a.length, lo, hi);
147         int nRemaining  = hi - lo;
148         if (nRemaining < 2)
149             return;  // Arrays of size 0 and 1 are always sorted
150
151         // If array is small, do a "mini-TimSort" with no merges
152         if (nRemaining < MIN_MERGE) {
```

```
153             int initRunLen = countRunAndMakeAscending(a, lo, hi);
154             binarySort(a, lo, hi, lo + initRunLen);
155             return;
156         }
157
158         /**
159          * March over the array once, left to right, finding natural
             runs,
160          * extending short natural runs to minRun elements, and merging
             runs
161          * to maintain stack invariant.
162          */
163         ComparableTimSort ts = new ComparableTimSort(a);
164         int minRun = minRunLength(nRemaining);
165         do {
166             // Identify next run
167             int runLen = countRunAndMakeAscending(a, lo, hi);
168
169             // If run is short, extend to min(minRun, nRemaining)
170             if (runLen < minRun) {
171                 int force = nRemaining <= minRun ? nRemaining : minRun;
172                 binarySort(a, lo, lo + force, lo + runLen);
173                 runLen = force;
174             }
175
176             // Push run onto pending-run stack, and maybe merge
177             ts.pushRun(lo, runLen);
178             ts.mergeCollapse();
179
180             // Advance to find next run
181             lo += runLen;
182             nRemaining -= runLen;
183         } while (nRemaining != 0);
184
185         // Merge all remaining runs to complete sort
186         if (DEBUG) assert lo == hi;
187         ts.mergeForceCollapse();
188         if (DEBUG) assert ts.stackSize == 1;
189     }
190
191     /**
192      * Sorts the specified portion of the specified array using a binary
193      * insertion sort.  This is the best method for sorting small
             numbers
194      * of elements.  It requires O(n log n) compares, but O(n^2) data
195      * movement (worst case).
196      *
197      * If the initial part of the specified range is already sorted,
198      * this method can take advantage of it: the method assumes that the
199      * elements from index {@code lo}, inclusive, to {@code start},
200      * exclusive are already sorted.
201      *
202      * @param a the array in which a range is to be sorted
203      * @param lo the index of the first element in the range to be
             sorted
204      * @param hi the index after the last element in the range to be
             sorted
205      * @param start the index of the first element in the range that is
206      *        not already known to be sorted (@code lo <= start <= hi}
207      */
```

```
208     @SuppressWarnings("fallthrough")
209     private static void binarySort(Object[] a, int lo, int hi, int
        start) {
210         if (DEBUG) assert lo <= start && start <= hi;
211         if (start == lo)
212             start++;
213         for ( ; start < hi; start++) {
214             @SuppressWarnings("unchecked")
215             Comparable<Object> pivot = (Comparable) a[start];
216
217             // Set left (and right) to the index where a[start] (pivot)
                belongs
218             int left = lo;
219             int right = start;
220             if (DEBUG) assert left <= right;
221             /*
222              * Invariants:
223              *   pivot >= all in [lo, left).
224              *   pivot <  all in [right, start).
225              */
226             while (left < right) {
227                 int mid = (left + right) >>> 1;
228                 if (pivot.compareTo(a[mid]) < 0)
229                     right = mid;
230                 else
231                     left = mid + 1;
232             }
233             if (DEBUG) assert left == right;
234
235             /*
236              * The invariants still hold: pivot >= all in [lo, left) and
237              * pivot < all in [left, start), so pivot belongs at left.
                 Note
238              * that if there are elements equal to pivot, left points to
                 the
239              * first slot after them -- that's why this sort is stable.
240              * Slide elements over to make room to make room for pivot.
241              */
242             int n = start - left;  // The number of elements to move
243             // Switch is just an optimization for arraycopy in default
                case
244             switch(n) {
245                 case 2:  a[left + 2] = a[left + 1];
246                 case 1:  a[left + 1] = a[left];
247                          break;
248                 default: System.arraycopy(a, left, a, left + 1, n);
249             }
250             a[left] = pivot;
251         }
252     }
253
254     /**
255      * Returns the length of the run beginning at the specified position
         in
256      * the specified array and reverses the run if it is descending
         (ensuring
257      * that the run will always be ascending when the method returns).
258      *
259      * A run is the longest ascending sequence with:
260      *
```

```
261       *    a[lo] <= a[lo + 1] <= a[lo + 2] <= ...
262       *
263       * or the longest descending sequence with:
264       *
265       *    a[lo] >  a[lo + 1] >  a[lo + 2] >  ...
266       *
267       * For its intended use in a stable mergesort, the strictness of the
268       * definition of "descending" is needed so that the call can safely
269       * reverse a descending sequence without violating stability.
270       *
271       * @param a the array in which a run is to be counted and possibly
          reversed
272       * @param lo index of the first element in the run
273       * @param hi index after the last element that may be contained in
          the run.
274                 It is required that @code{lo < hi}.
275       * @return  the length of the run beginning at the specified
          position in
276       *          the specified array
277       */
278      @SuppressWarnings("unchecked")
279      private static int countRunAndMakeAscending(Object[] a, int lo, int
          hi) {
280          if (DEBUG) assert lo < hi;
281          int runHi = lo + 1;
282          if (runHi == hi)
283              return 1;
284
285          // Find end of run, and reverse range if descending
286          if (((Comparable) a[runHi++]).compareTo(a[lo]) < 0) { //
             Descending
287              while(runHi < hi && ((Comparable)
                 a[runHi]).compareTo(a[runHi - 1]) < 0)
288                  runHi++;
289              reverseRange(a, lo, runHi);
290          } else {                              // Ascending
291              while (runHi < hi && ((Comparable)
                 a[runHi]).compareTo(a[runHi - 1]) >= 0)
292                  runHi++;
293          }
294
295          return runHi - lo;
296      }
297
298      /**
299       * Reverse the specified range of the specified array.
300       *
301       * @param a the array in which a range is to be reversed
302       * @param lo the index of the first element in the range to be
          reversed
303       * @param hi the index after the last element in the range to be
          reversed
304       */
305      private static void reverseRange(Object[] a, int lo, int hi) {
306          hi--;
307          while (lo < hi) {
308              Object t = a[lo];
309              a[lo++] = a[hi];
310              a[hi--] = t;
311          }
```

```
312     }
313
314     /**
315      * Returns the minimum acceptable run length for an array of the
         specified
316      * length. Natural runs shorter than this will be extended with
317      * {@link #binarySort}.
318      *
319      * Roughly speaking, the computation is:
320      *
321      *  If n < MIN_MERGE, return n (it's too small to bother with fancy
         stuff).
322      *  Else if n is an exact power of 2, return MIN_MERGE/2.
323      *  Else return an int k, MIN_MERGE/2 <= k <= MIN_MERGE, such that
         n/k
324      *   is close to, but strictly less than, an exact power of 2.
325      *
326      * For the rationale, see listsort.txt.
327      *
328      * @param n the length of the array to be sorted
329      * @return the length of the minimum run to be merged
330      */
331     private static int minRunLength(int n) {
332         if (DEBUG) assert n >= 0;
333         int r = 0;      // Becomes 1 if any 1 bits are shifted off
334         while (n >= MIN_MERGE) {
335             r |= (n & 1);
336             n >>= 1;
337         }
338         return n + r;
339     }
340
341     /**
342      * Pushes the specified run onto the pending-run stack.
343      *
344      * @param runBase index of the first element in the run
345      * @param runLen  the number of elements in the run
346      */
347     private void pushRun(int runBase, int runLen) {
348         this.runBase[stackSize] = runBase;
349         this.runLen[stackSize] = runLen;
350         stackSize++;
351     }
352
353     /**
354      * Examines the stack of runs waiting to be merged and merges
         adjacent runs
355      * until the stack invariants are reestablished:
356      *
357      *     1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
358      *     2. runLen[i - 2] > runLen[i - 1]
359      *
360      * This method is called each time a new run is pushed onto the
         stack,
361      * so the invariants are guaranteed to hold for i < stackSize upon
362      * entry to the method.
363      */
364     private void mergeCollapse() {
365         while (stackSize > 1) {
366             int n = stackSize - 2;
```

```
367             if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
368                 if (runLen[n - 1] < runLen[n + 1])
369                     n--;
370                 mergeAt(n);
371             } else if (runLen[n] <= runLen[n + 1]) {
372                 mergeAt(n);
373             } else {
374                 break; // Invariant is established
375             }
376         }
377     }
378
379     /**
380      * Merges all runs on the stack until only one remains.  This method is
381      * called once, to complete the sort.
382      */
383     private void mergeForceCollapse() {
384         while (stackSize > 1) {
385             int n = stackSize - 2;
386             if (n > 0 && runLen[n - 1] < runLen[n + 1])
387                 n--;
388             mergeAt(n);
389         }
390     }
391
392     /**
393      * Merges the two runs at stack indices i and i+1.  Run i must be
394      * the penultimate or antepenultimate run on the stack.  In other words,
395      * i must be equal to stackSize-2 or stackSize-3.
396      *
397      * @param i stack index of the first of the two runs to merge
398      */
399     @SuppressWarnings("unchecked")
400     private void mergeAt(int i) {
401         if (DEBUG) assert stackSize >= 2;
402         if (DEBUG) assert i >= 0;
403         if (DEBUG) assert i == stackSize - 2 || i == stackSize - 3;
404
405         int base1 = runBase[i];
406         int len1 = runLen[i];
407         int base2 = runBase[i + 1];
408         int len2 = runLen[i + 1];
409         if (DEBUG) assert len1 > 0 && len2 > 0;
410         if (DEBUG) assert base1 + len1 == base2;
411
412         /*
413          * Record the length of the combined runs; if i is the 3rd-last
414          * run now, also slide over the last run (which isn't involved
415          * in this merge).  The current run (i+1) goes away in any case.
416          */
417         runLen[i] = len1 + len2;
418         if (i == stackSize - 3) {
419             runBase[i + 1] = runBase[i + 2];
420             runLen[i + 1] = runLen[i + 2];
421         }
422         stackSize--;
423
424         /*
```

```
425          * Find where the first element of run2 goes in run1. Prior
             elements
426          * in run1 can be ignored (because they're already in place).
427          */
428         int k = gallopRight((Comparable<Object>) a[base2], a, base1,
             len1, 0);
429         if (DEBUG) assert k >= 0;
430         base1 += k;
431         len1 -= k;
432         if (len1 == 0)
433             return;
434
435         /*
436          * Find where the last element of run1 goes in run2. Subsequent
             elements
437          * in run2 can be ignored (because they're already in place).
438          */
439         len2 = gallopLeft((Comparable<Object>) a[base1 + len1 - 1], a,
440                 base2, len2, len2 - 1);
441         if (DEBUG) assert len2 >= 0;
442         if (len2 == 0)
443             return;
444
445         // Merge remaining runs, using tmp array with min(len1, len2)
             elements
446         if (len1 <= len2)
447             mergeLo(base1, len1, base2, len2);
448         else
449             mergeHi(base1, len1, base2, len2);
450     }
451
452     /**
453      * Locates the position at which to insert the specified key into
         the
454      * specified sorted range; if the range contains an element equal to
         key,
455      * returns the index of the leftmost equal element.
456      *
457      * @param key the key whose insertion point to search for
458      * @param a the array in which to search
459      * @param base the index of the first element in the range
460      * @param len the length of the range; must be > 0
461      * @param hint the index at which to begin the search, 0 <= hint <
         n.
462      *     The closer hint is to the result, the faster this method will
         run.
463      * @return the int k,  0 <= k <= n such that a[b + k - 1] < key <=
         a[b + k],
464      *    pretending that a[b - 1] is minus infinity and a[b + n] is
         infinity.
465      *    In other words, key belongs at index b + k; or in other words,
466      *    the first k elements of a should precede key, and the last n -
         k
467      *    should follow it.
468      */
469     private static int gallopLeft(Comparable<Object> key, Object[] a,
470             int base, int len, int hint) {
471         if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
472
473         int lastOfs = 0;
```

```
474         int ofs = 1;
475         if (key.compareTo(a[base + hint]) > 0) {
476             // Gallop right until a[base+hint+lastOfs] < key <=
                a[base+hint+ofs]
477             int maxOfs = len - hint;
478             while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) >
                0) {
479                 lastOfs = ofs;
480                 ofs = (ofs << 1) + 1;
481                 if (ofs <= 0)   // int overflow
482                     ofs = maxOfs;
483             }
484             if (ofs > maxOfs)
485                 ofs = maxOfs;
486
487             // Make offsets relative to base
488             lastOfs += hint;
489             ofs += hint;
490         } else { // key <= a[base + hint]
491             // Gallop left until a[base+hint-ofs] < key <=
                a[base+hint-lastOfs]
492             final int maxOfs = hint + 1;
493             while (ofs < maxOfs && key.compareTo(a[base + hint - ofs])
                <= 0) {
494                 lastOfs = ofs;
495                 ofs = (ofs << 1) + 1;
496                 if (ofs <= 0)   // int overflow
497                     ofs = maxOfs;
498             }
499             if (ofs > maxOfs)
500                 ofs = maxOfs;
501
502             // Make offsets relative to base
503             int tmp = lastOfs;
504             lastOfs = hint - ofs;
505             ofs = hint - tmp;
506         }
507         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
508
509         /*
510          * Now a[base+lastOfs] < key <= a[base+ofs], so key belongs
             somewhere
511          * to the right of lastOfs but no farther right than ofs.  Do a
             binary
512          * search, with invariant a[base + lastOfs - 1] < key <= a[base
             + ofs].
513          */
514         lastOfs++;
515         while (lastOfs < ofs) {
516             int m = lastOfs + ((ofs - lastOfs) >>> 1);
517
518             if (key.compareTo(a[base + m]) > 0)
519                 lastOfs = m + 1;  // a[base + m] < key
520             else
521                 ofs = m;          // key <= a[base + m]
522         }
523         if (DEBUG) assert lastOfs == ofs;    // so a[base + ofs - 1] <
            key <= a[base + ofs]
524         return ofs;
525     }
```

```
526
527     /**
528      * Like gallopLeft, except that if the range contains an element
         equal to
529      * key, gallopRight returns the index after the rightmost equal
         element.
530      *
531      * @param key the key whose insertion point to search for
532      * @param a the array in which to search
533      * @param base the index of the first element in the range
534      * @param len the length of the range; must be > 0
535      * @param hint the index at which to begin the search, 0 <= hint <
         n.
536      *     The closer hint is to the result, the faster this method will
         run.
537      * @return the int k,  0 <= k <= n such that a[b + k - 1] <= key <
         a[b + k]
538      */
539     private static int gallopRight(Comparable<Object> key, Object[] a,
540             int base, int len, int hint) {
541         if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
542
543         int ofs = 1;
544         int lastOfs = 0;
545         if (key.compareTo(a[base + hint]) < 0) {
546             // Gallop left until a[b+hint - ofs] <= key < a[b+hint -
                lastOfs]
547             int maxOfs = hint + 1;
548             while (ofs < maxOfs && key.compareTo(a[base + hint - ofs]) <
                0) {
549                 lastOfs = ofs;
550                 ofs = (ofs << 1) + 1;
551                 if (ofs <= 0)   // int overflow
552                     ofs = maxOfs;
553             }
554             if (ofs > maxOfs)
555                 ofs = maxOfs;
556
557             // Make offsets relative to b
558             int tmp = lastOfs;
559             lastOfs = hint - ofs;
560             ofs = hint - tmp;
561         } else { // a[b + hint] <= key
562             // Gallop right until a[b+hint + lastOfs] <= key < a[b+hint
                + ofs]
563             int maxOfs = len - hint;
564             while (ofs < maxOfs && key.compareTo(a[base + hint + ofs])
                >= 0) {
565                 lastOfs = ofs;
566                 ofs = (ofs << 1) + 1;
567                 if (ofs <= 0)   // int overflow
568                     ofs = maxOfs;
569             }
570             if (ofs > maxOfs)
571                 ofs = maxOfs;
572
573             // Make offsets relative to b
574             lastOfs += hint;
575             ofs += hint;
576         }
```

```
577         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
578
579         /*
580          * Now a[b + lastOfs] <= key < a[b + ofs], so key belongs
             somewhere to
581          * the right of lastOfs but no farther right than ofs.  Do a
             binary
582          * search, with invariant a[b + lastOfs - 1] <= key < a[b +
             ofs].
583          */
584         lastOfs++;
585         while (lastOfs < ofs) {
586             int m = lastOfs + ((ofs - lastOfs) >>> 1);
587
588             if (key.compareTo(a[base + m]) < 0)
589                 ofs = m;          // key < a[b + m]
590             else
591                 lastOfs = m + 1;  // a[b + m] <= key
592         }
593         if (DEBUG) assert lastOfs == ofs;    // so a[b + ofs - 1] <= key
            < a[b + ofs]
594         return ofs;
595     }
596
597     /**
598      * Merges two adjacent runs in place, in a stable fashion.  The
         first
599      * element of the first run must be greater than the first element
         of the
600      * second run (a[base1] > a[base2]), and the last element of the
         first run
601      * (a[base1 + len1-1]) must be greater than all elements of the
         second run.
602      *
603      * For performance, this method should be called only when len1 <=
         len2;
604      * its twin, mergeHi should be called if len1 >= len2.  (Either
         method
605      * may be called if len1 == len2.)
606      *
607      * @param base1 index of first element in first run to be merged
608      * @param len1  length of first run to be merged (must be > 0)
609      * @param base2 index of first element in second run to be merged
610      *        (must be aBase + aLen)
611      * @param len2  length of second run to be merged (must be > 0)
612      */
613     @SuppressWarnings("unchecked")
614     private void mergeLo(int base1, int len1, int base2, int len2) {
615         if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;
616
617         // Copy first run into temp array
618         Object[] a = this.a; // For performance
619         Object[] tmp = ensureCapacity(len1);
620         System.arraycopy(a, base1, tmp, 0, len1);
621
622         int cursor1 = 0;       // Indexes into tmp array
623         int cursor2 = base2;   // Indexes int a
624         int dest = base1;      // Indexes int a
625
626         // Move first element of second run and deal with degenerate
```

```
            cases
627         a[dest++] = a[cursor2++];
628         if (--len2 == 0) {
629             System.arraycopy(tmp, cursor1, a, dest, len1);
630             return;
631         }
632         if (len1 == 1) {
633             System.arraycopy(a, cursor2, a, dest, len2);
634             a[dest + len2] = tmp[cursor1]; // Last elt of run 1 to end
                of merge
635             return;
636         }
637
638         int minGallop = this.minGallop;  // Use local variable for
            performance
639     outer:
640         while (true) {
641             int count1 = 0; // Number of times in a row that first run
                won
642             int count2 = 0; // Number of times in a row that second run
                won
643
644             /*
645              * Do the straightforward thing until (if ever) one run
                 starts
646              * winning consistently.
647              */
648             do {
649                 if (DEBUG) assert len1 > 1 && len2 > 0;
650                 if (((Comparable) a[cursor2]).compareTo(tmp[cursor1]) <
                    0) {
651                     a[dest++] = a[cursor2++];
652                     count2++;
653                     count1 = 0;
654                     if (--len2 == 0)
655                         break outer;
656                 } else {
657                     a[dest++] = tmp[cursor1++];
658                     count1++;
659                     count2 = 0;
660                     if (--len1 == 1)
661                         break outer;
662                 }
663             } while ((count1 | count2) < minGallop);
664
665             /*
666              * One run is winning so consistently that galloping may be
                 a
667              * huge win. So try that, and continue galloping until (if
                 ever)
668              * neither run appears to be winning consistently anymore.
669              */
670             do {
671                 if (DEBUG) assert len1 > 1 && len2 > 0;
672                 count1 = gallopRight((Comparable) a[cursor2], tmp,
                    cursor1, len1, 0);
673                 if (count1 != 0) {
674                     System.arraycopy(tmp, cursor1, a, dest, count1);
675                     dest += count1;
676                     cursor1 += count1;
```

```
677                     len1 -= count1;
678                     if (len1 <= 1)  // len1 == 1 || len1 == 0
679                         break outer;
680                 }
681                 a[dest++] = a[cursor2++];
682                 if (--len2 == 0)
683                     break outer;
684
685                 count2 = gallopLeft((Comparable) tmp[cursor1], a,
                    cursor2, len2, 0);
686                 if (count2 != 0) {
687                     System.arraycopy(a, cursor2, a, dest, count2);
688                     dest += count2;
689                     cursor2 += count2;
690                     len2 -= count2;
691                     if (len2 == 0)
692                         break outer;
693                 }
694                 a[dest++] = tmp[cursor1++];
695                 if (--len1 == 1)
696                     break outer;
697                 minGallop--;
698             } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
699             if (minGallop < 0)
700                 minGallop = 0;
701             minGallop += 2;  // Penalize for leaving gallop mode
702         }  // End of "outer" loop
703         this.minGallop = minGallop < 1 ? 1 : minGallop;  // Write back
            to field
704
705         if (len1 == 1) {
706             if (DEBUG) assert len2 > 0;
707             System.arraycopy(a, cursor2, a, dest, len2);
708             a[dest + len2] = tmp[cursor1]; //  Last elt of run 1 to end
                of merge
709         } else if (len1 == 0) {
710             throw new IllegalArgumentException(
711                 "Comparison method violates its general contract!");
712         } else {
713             if (DEBUG) assert len2 == 0;
714             if (DEBUG) assert len1 > 1;
715             System.arraycopy(tmp, cursor1, a, dest, len1);
716         }
717     }
718
719     /**
720      * Like mergeLo, except that this method should be called only if
721      * len1 >= len2; mergeLo should be called if len1 <= len2.  (Either
           method
722      * may be called if len1 == len2.)
723      *
724      * @param base1 index of first element in first run to be merged
725      * @param len1  length of first run to be merged (must be > 0)
726      * @param base2 index of first element in second run to be merged
727      *        (must be aBase + aLen)
728      * @param len2  length of second run to be merged (must be > 0)
729      */
730     @SuppressWarnings("unchecked")
731     private void mergeHi(int base1, int len1, int base2, int len2) {
732         if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;
```

```
733
734         // Copy second run into temp array
735         Object[] a = this.a; // For performance
736         Object[] tmp = ensureCapacity(len2);
737         System.arraycopy(a, base2, tmp, 0, len2);
738
739         int cursor1 = base1 + len1 - 1;  // Indexes into a
740         int cursor2 = len2 - 1;          // Indexes into tmp array
741         int dest = base2 + len2 - 1;     // Indexes into a
742
743         // Move last element of first run and deal with degenerate cases
744         a[dest--] = a[cursor1--];
745         if (--len1 == 0) {
746             System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
747             return;
748         }
749         if (len2 == 1) {
750             dest -= len1;
751             cursor1 -= len1;
752             System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
753             a[dest] = tmp[cursor2];
754             return;
755         }
756
757         int minGallop = this.minGallop;  // Use local variable for
            performance
758     outer:
759         while (true) {
760             int count1 = 0; // Number of times in a row that first run
                won
761             int count2 = 0; // Number of times in a row that second run
                won
762
763             /*
764              * Do the straightforward thing until (if ever) one run
765              * appears to win consistently.
766              */
767             do {
768                 if (DEBUG) assert len1 > 0 && len2 > 1;
769                 if (((Comparable) tmp[cursor2]).compareTo(a[cursor1]) <
                    0) {
770                     a[dest--] = a[cursor1--];
771                     count1++;
772                     count2 = 0;
773                     if (--len1 == 0)
774                         break outer;
775                 } else {
776                     a[dest--] = tmp[cursor2--];
777                     count2++;
778                     count1 = 0;
779                     if (--len2 == 1)
780                         break outer;
781                 }
782             } while ((count1 | count2) < minGallop);
783
784             /*
785              * One run is winning so consistently that galloping may be
                a
786              * huge win. So try that, and continue galloping until (if
                ever)
```

```
787              * neither run appears to be winning consistently anymore.
788              */
789             do {
790                 if (DEBUG) assert len1 > 0 && len2 > 1;
791                 count1 = len1 - gallopRight((Comparable) tmp[cursor2],
                    a, base1, len1, len1 - 1);
792                 if (count1 != 0) {
793                     dest -= count1;
794                     cursor1 -= count1;
795                     len1 -= count1;
796                     System.arraycopy(a, cursor1 + 1, a, dest + 1,
                        count1);
797                     if (len1 == 0)
798                         break outer;
799                 }
800                 a[dest--] = tmp[cursor2--];
801                 if (--len2 == 1)
802                     break outer;
803
804                 count2 = len2 - gallopLeft((Comparable) a[cursor1], tmp,
                    0, len2, len2 - 1);
805                 if (count2 != 0) {
806                     dest -= count2;
807                     cursor2 -= count2;
808                     len2 -= count2;
809                     System.arraycopy(tmp, cursor2 + 1, a, dest + 1,
                        count2);
810                     if (len2 <= 1)
811                         break outer; // len2 == 1 || len2 == 0
812                 }
813                 a[dest--] = a[cursor1--];
814                 if (--len1 == 0)
815                     break outer;
816                 minGallop--;
817             } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
818             if (minGallop < 0)
819                 minGallop = 0;
820             minGallop += 2;  // Penalize for leaving gallop mode
821         }  // End of "outer" loop
822         this.minGallop = minGallop < 1 ? 1 : minGallop;  // Write back
            to field
823
824         if (len2 == 1) {
825             if (DEBUG) assert len1 > 0;
826             dest -= len1;
827             cursor1 -= len1;
828             System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
829             a[dest] = tmp[cursor2];  // Move first elt of run2 to front
                of merge
830         } else if (len2 == 0) {
831             throw new IllegalArgumentException(
832                 "Comparison method violates its general contract!");
833         } else {
834             if (DEBUG) assert len1 == 0;
835             if (DEBUG) assert len2 > 0;
836             System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
837         }
838     }
839
840     /**
```

```
841      * Ensures that the external array tmp has at least the specified
842      * number of elements, increasing its size if necessary.  The size
843      * increases exponentially to ensure amortized linear time
         complexity.
844      *
845      * @param minCapacity the minimum required capacity of the tmp array
846      * @return tmp, whether or not it grew
847      */
848     private Object[]  ensureCapacity(int minCapacity) {
849         if (tmp.length < minCapacity) {
850             // Compute smallest power of 2 > minCapacity
851             int newSize = minCapacity;
852             newSize |= newSize >> 1;
853             newSize |= newSize >> 2;
854             newSize |= newSize >> 4;
855             newSize |= newSize >> 8;
856             newSize |= newSize >> 16;
857             newSize++;
858
859             if (newSize < 0) // Not bloody likely!
860                 newSize = minCapacity;
861             else
862                 newSize = Math.min(newSize, a.length >>> 1);
863
864             @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
865             Object[] newArray = new Object[newSize];
866             tmp = newArray;
867         }
868         return tmp;
869     }
870
871     /**
872      * Checks that fromIndex and toIndex are in range, and throws an
873      * appropriate exception if they aren't.
874      *
875      * @param arrayLen the length of the array
876      * @param fromIndex the index of the first element of the range
877      * @param toIndex the index after the last element of the range
878      * @throws IllegalArgumentException if fromIndex > toIndex
879      * @throws ArrayIndexOutOfBoundsException if fromIndex < 0
880      *         or toIndex > arrayLen
881      */
882     private static void rangeCheck(int arrayLen, int fromIndex, int
         toIndex) {
883         if (fromIndex > toIndex)
884             throw new IllegalArgumentException("fromIndex(" + fromIndex
                 +
885                     ") > toIndex(" + toIndex+")");
886         if (fromIndex < 0)
887             throw new ArrayIndexOutOfBoundsException(fromIndex);
888         if (toIndex > arrayLen)
889             throw new ArrayIndexOutOfBoundsException(toIndex);
890     }
891 }
```