Essential Improvements to basic quicksort

The basic quicksort algorithm should be improved in three ways:

1. When the array segment is small (say size 10 or less) recursive calls are relatively expensive, so use a different algorithm.
2. A better choice of pivot value, the median of three, will improve execution time.
3. In the worst case, quicksort uses space linear in the array size; this can be modified to be logarithmic.

We address each of these issues in turn.

But first, remember that the partition algorithm is written as a method, as shown to the right.

```
/** Let x be the value in b[h].
  * Permute b[h..k] to truthify
  *     b[h..j-1] <= x = b[j] <= b[j+1..k]
  * and return j. */
public static void partition(
            int[] b, int h, int k) { … }
```

1. Insertion sort is faster than quicksort on small arrays.

It turn out that insertionsort is faster than quicksort on small arrays. This is because quicksort makes several recursive calls, and method calls *do* take time. After all, a frame for the call has to be placed on the call stack, arguments values have to be assigned to parameters, and so forth. Just what do we mean by "small"? That may depend on the computer on which the method is being executed. In the 1970's and 1980's, people did experiments to figure out what "small" meant in this case. Today, let's just say for the sake of convenience that arrays of size at most 10 are sorted faster using insertionsort. So, in the method given below, arrays of size at most 10 will be sorted using insertionsort. You'll see how we do this.

2. Use the median of three for the pivot value.

Quicksort is slowest when the pivot is always the smallest or largest possible value. The best possible pivot is the median of the segment `b[h..k]` being sorted. That median can actually be calculated and used, but the calculation is too slow. Instead, one generally uses the median of three values: `b[h]`, `b[(h+k)/2]`, and `b[k]`.

To the right, we define a method to swap the median of three values of `b[h..k]` into `b[h]`. We leave its implementation to you. In the improved quicksort algorithm, we'll call this method before calling method partition.

```
/** Permute b[h], b[(h+k)/2], and
  * b[k] to put their median in b[h]. */
public static void medianOf3(
            int[] b, int h, int k) { … }
```

3. Use at most logarithmic space

When the pivot is always the smallest value of the segment to be partitioned, the depth of recursion is linear in the size of the array, so the space required is linear in the size of the array. The first step in changing this is to sort the smaller of the two partitions created by the partition algorithm first, as shown in the algorithm on the right.

This means that the largest partition is sorted last and that *nothing else is done in the method after the largest partition is sorted.* If you know about *tail-calls*, you know these are called tail-calls, and if tail-calls are optimized by the compiler, no frames for them are created. If you don't know about *tail recursion* and *tail calls*, don't be concerned; we'll show how to implement this efficiently below.

```
/** Sort b[h..k] */
public static void qSort(int[] b, int h, int k) {
   if (k+1 – h < 2) return;
   int j= partition(b, h, k);
   // b[h..j-1] ≤ b[j] ≤ b[j+1..k]
   if (j-h < k-j) { // left partition is smaller
      qSort(b, h, j-1); qSort(b, j+1, k);
   } else { { // right partition is smaller
      qSort(b, j+1, k); qSort(b, h, j-1);
   }
}
```

4. Putting it all together

The version of quicksort that appears to the right contains the three modifications discussed above. It has a loop with this invariant:

P: if b[h1..k1] is sorted, then so is b[h..k]

You can see that the initialization truthifies P, because the following statement is true:

if b[h..k] is sorted, then so is b[h..k]

When the loop terminates, we know that

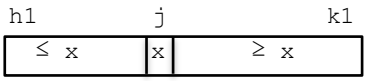
P and b[h1..k1] has at most 10 elements

Therefore, sorting b[h1..k1] will terminate with the whole array segment b[h..k] sorted. Segment b[h1..k1] is sorted using insertionsort, which is faster on "smaller" array segements than quicksort.

```
/** Sort b[h..k] */
public static void qSort(int[] b, int h, int k) {
   int h1= h;
   int k1= k;
   // invariant P: if b[h1..k1] is sorted, then so is b[h..k]
   while (h2 – k1 <= 9) {
      medianOf3(b, h1, k1);
      int j= partition(b, h1, k1);
      // b[h1..j-1] ≤ b[j] ≤ b[j+1..k1]
      if (j-h1 < k1-j) { // left is partition smaller
         qSort(b, h1, j-1);
         h1= j+1;
      } else { // right partition is smaller
         qSort(b, j+1, k);
         k1= j-1;
      }
   }
   // post:  P and b[h1..k1] has at most 10 elements
   insertionsort(b, h1, k1);
}
```

It remains to check that the repetend maintains invariant P and makes progress toward termination. We do that now.

The repetend has to make progress in sorting b[h1..k1]. First, the median of three values is placed in b[h1] in order to improve the pivot value. Second b[h1..k1] is partitioned so that this is true:



Consider the case that the left partition b[h1..j-1] is smaller (or the same size) as the right one, b[j+1..k1] — the other case is similar and won't be discussed. In this case, b[h1..j1-1] is sorted recursively. After that, we know that b[h..k] will be sorted if the right partition b[j+1..k1] is sorted, so to truthify invariant P, set h1 to j+1. This also reduces the size of b[h1..k1], so progress toward termination has been made.

Let's discuss the space used on the call stack. With each iteration, the smaller of two partitions is sorted recursively. Since the smaller one is less than half the size of b[h1..k1], the depth of recursion is no more than log(k1+1-h1). Thus the space used to sort an array segment of size n is O(log n).

Isn't that neat?