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. In addition, we mention some other improvements.

**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, some people did experiments to figure out what “small” meant. Today, let’s just say for the sake of convenience that arrays of size 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.**

You know that quicksort is slowest when the pivot is always the smallest or largest possible value. Also,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 to be used. Instead, one generally used the median of three values: b[h], b[(h+k)/2], and b[k].

/\*\* 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) { … }

To the right, we give the specification of a method to move the median of three values of b[h..k] to b[h]. We leave its implementation to you. In the improveds quicksort algorithm, we’ll call this method before calling method partition.

**Use at most logarithmic space**

We write a procedure quicksort with the specification shown to the right. To sort the complete array b, use the call

qSort(b, 0, b.length-1);

Procedure qSort will be recursive.

To easily develop qSort, you need to remember only that it will be recursive and that it depends on the partition algorithm, which starts with precondition Pre true and swaps elements of b[h..k] to truthify Post: Method partition is developed in a pdf file linked to in the JavaHyperText entry “partition”.

h k

x ?

Pre: b

h j k

≤ x x ≥ x

Post: b

The base case is the situation in which b[h..k] doesn’t need sorting because its size is 0 or 1.

In the recursive case, use the partition algorithm to truthify Post, above. Then, to complete sorting b[h..k], the two partitions b[h..j-1] and b[j+1..k] need to be sorted. This yields the following method.

/\*\* 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); // partition algorithm is written as a method that returns j.

// b[h..j-1] ≤ x ≤ b[j+1..k]

qSort(b, h, j-1);

qSort(b, j+1, k);

}

Quicksort is inherently unstable because the partition algorithm is unstable.

**History of Quicksort**

Quicksort was developed by Tony Hoare in 1959–60. During a visit to Cornell in 2004, Sir Tony (he was knighted by the Queen of England for his services to education and computer science), gave a lecture in CS211 (later numbered CS2110). He was asked about quicksort. He said it was the second sorting algorithm he thought of. When he first thought of it, he attempted to explain it to a colleage, but the colleague couldn’t understand it. Some time later, Tony saw a draft of the new language Algol. It had recursive procedures. Sir Tony, a logician/mathema-tician, certainly knew about recursion but had not thought about it in connection with programming. He wrote a recursive version of quicksort. It took a few minutes to explain quicksort to his colleague.

**Best-case execution of quicksort**

Suppose that during execution of qSort each use of the partition algorithm creates two equal-sized (or almost equal) partitions b[k..j-1] and b[j+1..k]. Then the execution time to sort an array of size n is O(n log n). We prove this as follows. Suppose array b contains n elements. We investigate execution of the call qSort(b, 0, b.length-1); .

Consider the tree shown below. The root represents the n elements of array b. Algorithm partition creates two partitions b[0..j-1] and b[j+1..b.length], each containing at most n/2 elements (since they are of almost equal size and don’t contain b[j]). The two children of the root represent those two partitions of size at most n/2.

n/4

n/4

n/4

n/4

n/8

n/8

n/8

n/8

n/8

n/8

n/8

n/8

8\*n/8 = n

n/2

n/2

n

n

2\*n/2 = n

4\*n/4 = n

**… … … … … … … …** log n

Partitioning each of the partitions of size at most n/2 results in two partitions of size at most n/4 (since the two partitions are of equal size …). The four children represent the four partitions of size n/4.

Partitioning each of the partitions of size at most n/4 results in two partitions of size at most n/2 (since the two partitions are of equal size …). The four children represent the four partitions of size n/2.

This continues at each level for at most 1 + log n levels, because at at the bottom level, the two partitions have size 0 or 1 and the recursion stops. Example: if n = 2k, the tree has at most 1+k levels.

Calling the top level number 1, at each level k except the bottom one, k partitions of size at most n/k are partitioned. Since algorithm partition takes linear time, partitioning the k partitions takes time O(k\*n/k), which is O(n). Since partitioning occurs on all levels except the bottom one, it happens on log n levels, and since the time for each level is O(n), the total time is O(n log n).

**Worst-case execution of quicksort**

Suppose the pivot value is always the *smallest* value in the array segment being partitioned, which may happen when the array is already in ascending order. Then, the first call on algorithm results in a left partition of size 0 and a right partition of size n-1. Calling partition on the right one results in a left partition of size 0 and a right partition of size n-2. Partitiong the right partitions takes time proportional to n, n-1, n-2, n-2, … 2. Add these up to see that quicksort in this case take time O(n2). So, the worst case is quadratic in the size of the array.

n

0

n-1

n-2

0

…

0

**Average-case execution of quicksort**

It has been proven that the average or expected time of quicksort to sort an array of size n is O(n log n). The proof is beyond the scope of this JavaHyperText.

**Space requirements of quicksort**

Quicksort is an in-place, or *in-situ* sorting algorithm. It doesn’t need other arrays or data structures, except a few local variables.

However, quicksort *does* require space on the call stack for the frames for each call on it. So, in the best case, when the pivot value is always the median of the array values, the depth of recursion is O(log n) and O(log n) space is required.

But if the pivot value is always the smallest possible value, the depth of recursion is O(n), so O(n) space is required! Not good.

**Further improvements**

In another document in this JavaHyperText, we discuss three improvements to this basic quicksort.