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

/\*\* Sort b[h..k] \*/

public static void qSort(  
 int[] b, int h, int k)

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

Procedure qSort will be recursive.

To easily develop qSort, you need to remember only that is 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:

h j k

≤ x x ≥ x

Post: b

h k

x ?

Pre: 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 segments b[h..j-1] and b[j+1..k] need to be sorted. This yields the following method. Method partition is developed in a pdf file linked to in the JavaHyperText entry “partition”.

/\*\* 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] ≤ x ≤ b[j+1..k]

qSort(b, h, j-1);

qSort(b, j+1, k);

}

**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 could not understand it. Some time later, Tony saw a draft of the new language Algol. It had recursive procedures. Sir Tony, a logician/mathematician, certainly knew about recursion but had not thought about it in connection with programming. He realized that recursion was the key to a simple explanation of quicksort. It took a few minutes to explain quicksort to his colleague.

**Best-case execution of quicksort**

Suppose during an execution of qSort each use of the partition algorithm partitions the array into two equal (or almost equal) size segments b[k..j-1] and b[j+1..k]. Then the execution time is O(n log n) to sort an array of size 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 stands for the n elements of array b. The partition algorithm partitions b into two segments [0..j-1] and b[j+1..b.length], each containing at most n/2 elements (since they are of equal size). The two children of the root represent those two segments of size at most n/2 elements.

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

Each segment of size at most n/2 is partitioned into two segments of size at most n/4 (since the two segments are of equal size). The four children represent the four segments of size n/4.

Each segment of size at most n/4 is partitioned into two segments of size at most n/8 (since the two segments are of equal size). The eight children represent the four segments of size n/8.

This continues at each level for at most log n levels, because when

**Use the median of three for x**

The closer that x is to the median of b[h..k], the faster quicksort is. It would be nice to compute the median and use it for x, but that takes too much time. Instead, the standard approach is to swap the median of b[h], b[(h+k)/2], and b[k], into b[h], before calling method partition. That increases the chance that x is close to the median.