# The basic quicksort algorithm

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);
```

```
/** Sort b[h..k] */
public static void qSort(
          int[] b, int h, int k)
```

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".


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] ≤ b[j] ≤ b[j+1..k]
       qSort(b, h, j-1);
       qSort(b, j+1, k);
}
```

Quicksort is inherently unstable because the partition algorithm is unstable.

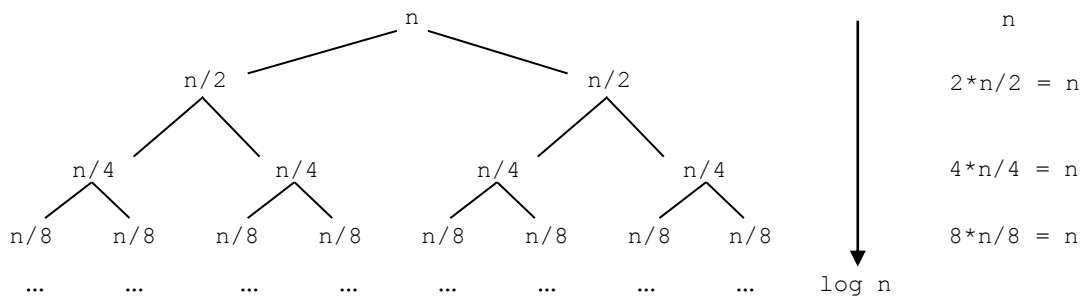
**History of Quicksort**

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 colleague, 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/mathematician, 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`.



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`.

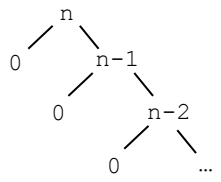
Partitioning each of the partitions of size at most `n/4` results in two partitions of size at most `n/2` (since the two partitions are of equal size …). The four children represent the four partitions of size `n/2`.

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 $n = 2^k$, 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 $O(n^2)$. So, the worst case is quadratic in the size of the array.




**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.

- Make the space requirements always to be `O(log n)`.
- Attempt to choose a better pivot value than the first one in the segment being partitioned.
- Stop the recursion when the segment to be partitioned has size at most 10, instead of 1, and sort that segment in a different way.