~ split array into partitions and sort those.

•Partitions the array, work on the front end.

↑ in position

// index of slot for partition

Value?

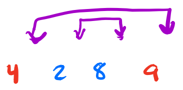
(costly)

O

moves and returns s = location pivot.

pivot i j

pivot



pivot ~~i~~ ~~i~~ i i j j j

Partition:

2 . . . . . j s i

swap

swap

Best Case

p

Worst case

Quicksort:

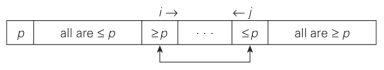
↳ Divide- and- conquer approach

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are } \ge A[s]}$$

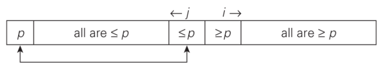
```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //         indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ← Partition(A[l..r])  //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

Example

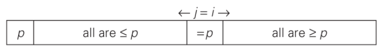
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:



Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



Pseudo Code for the process described above.

```
ALGORITHM  HoarePartition(A[l..r])
    //Partitions a subarray by Hoare's algorithm, using the first element
    //        as a pivot
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //        indices l and r (l < r)
    //Output: Partition of A[l..r], with the split position returned as
    //        this function's value
    p ← A[l]
    i ← l; j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])    //undo last swap when i ≥ j
    swap(A[l], A[j])
    return j
```

Best Case

We start our discussion of quicksort's efficiency by noting that the number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over and n if they coincide (why?). If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

→ Quicksort is not stable as it requires a stack to store parameters of subarrays.

- better pivot selection methods such as ***randomized quicksort*** that uses a random element or the ***median-of-three*** method that uses the median of the leftmost, rightmost, and the middle element of the array
- switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot (see Problem 9 in this section's exercises)

Worst case

$$C_{worst}(n) = (n + 1) + n + \cdots + 3 = \frac{(n + 1)(n + 2)}{2} - 3 \in \Theta(n^2).$$

Average Case

Thus, the question about the utility of quicksort comes down to its average-case behavior. Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n - 1$) after $n + 1$ comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$