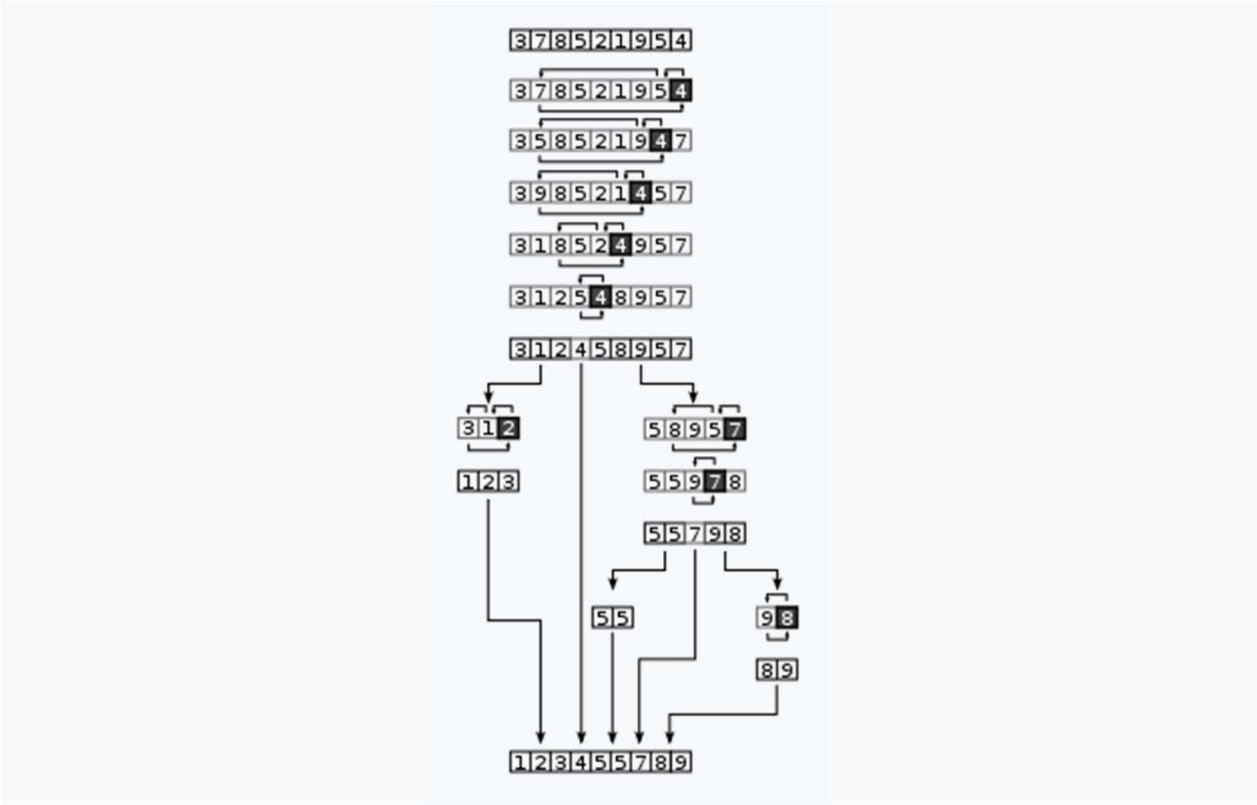
Quicksort

Quicksort	
Class	Sorting algorithm
Worst-case performance	$O(n^2)$
Best-case performance	$O(n \log n)$ (simple partition) or $O(n)$ (three-way partition and equal keys)
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$ auxiliary (naive) $O(\log n)$ auxiliary (Hoare 1962)

- **Quicksort** (sometimes called **partition-exchange sort**) is an efficient sorting algorithm.
- Developed by British computer scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting.
- When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.
- Quicksort is a divide-and-conquer algorithm.
- It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.
- This can be done in-place, requiring small additional amounts of memory to perform the sorting.
- Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined.
- Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.
- Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items.

- In the worst case, it makes O(n^2) comparisons, though this behavior is rare.



- Full example of quicksort on a random set of numbers.
- The shaded element is the pivot.
- It is always chosen as the last element of the partition.
- However, always choosing the last element in the partition as the pivot in this way results in poor performance ($O(n^2)$) on *already sorted* arrays, or arrays of identical elements.
- Since sub-arrays of sorted / identical elements crop up a lot towards the end of a sorting procedure on a large set, versions of the quicksort algorithm that choose the pivot as the middle element run much more quickly than the algorithm described in this diagram on large sets of numbers.
- Quicksort is a divide and conquer algorithm.
- It first divides the input array into two smaller sub-arrays: the low elements and the high elements. It then recursively sorts the sub-arrays. The steps for in-place Quicksort are:

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

- The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.
- The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

• Algorithm

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Pseudocode,

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[⌊(hi + lo) / 2⌋]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i ≥ j then
            return j
        swap A[i] with A[j]
```

Implementation issues

Choice of pivot

- In the very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element.
- Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case.
- The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot $\text{mid} := (\text{lo} + \text{hi}) / 2$.

```
if A[mid] < A[lo]
    swap A[lo] with A[mid]
if A[hi] < A[lo]
    swap A[lo] with A[hi]
if A[mid] < A[hi]
    swap A[mid] with A[hi]
pivot := A[hi]
```