# Quicksort — An Example

We sort the array

$A$ = **(38 81 22 48 13 69 93 14 45 58 79 72)**

with quicksort, always choosing the pivot element to be the element in position $\lfloor(left+right)/2\rfloor$.

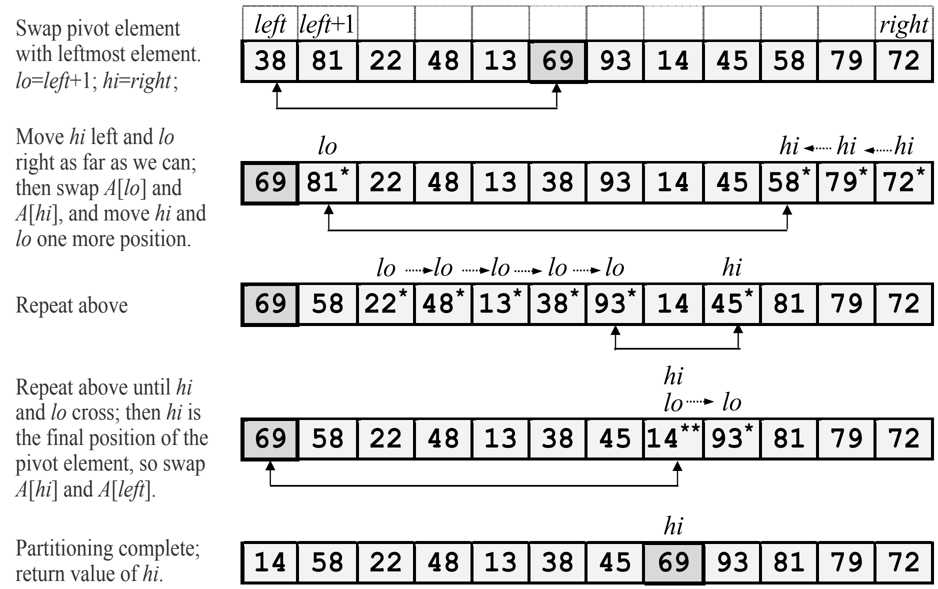
The partitioning during the top-level call to *quicksort*() is illustrated on the next page. During the partitioning process,

i) Elements strictly to the left of position *lo* are less than or equivalent to the pivot element (69).

ii) Elements strictly to the right of position *hi* are greater than the pivot element.

When *lo* and *hi* cross, we are done. The final value of *hi* is the position in which the partitioning element ends up.

An asterisk indicates an element compared to the pivot element at that step.



Number of comparisons performed by *partition*():

a) No comparison for leftmost column.

b) One comparison for each remaining column, except two for the columns where *hi* and *lo* end up. ($lo = hi+1$ at the end.)

$\boldsymbol{C(n) = n + 1}$, where $n$ is the size of the array ($right - left + 1$).

Expected number of exchanges performed by *partition*(), for a randomly ordered array, and a pivot element chosen from a designated position (and hence a random element of the array).

Say the pivot element turns out to be the $k^{\text{th}}$ largest element of the $n$ elements, so it ends up in the $k^{\text{th}}$ position.

Each exchange of $A[lo]$ and $A[hi]$ moves one element initially to the right of $k^{\text{th}}$ position, but less than (or equivalent but not equal to) the pivot element, to a position not right of the $k^{\text{th}}$ position.

Of the $k$–1 elements in the array less than the pivot element, we would expect $\big((n-k)/(n-1)\big)\ (k-1)$ of these to lie initially right of the $k^{\text{th}}$ position. Thus we expect $(k-1)(n-k)/(n-1) \approx k(n-k)/n$ exchanges.

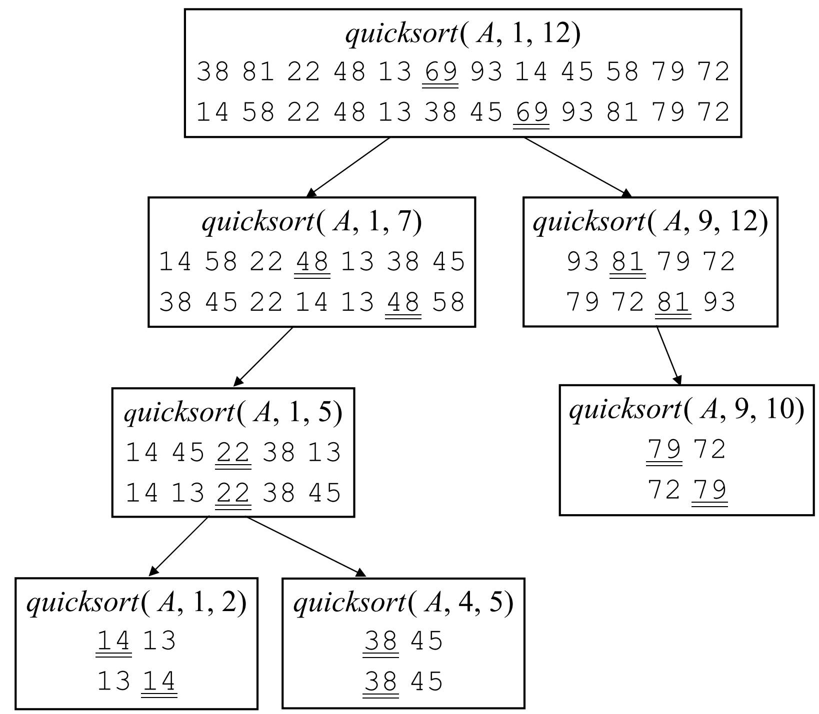
Since all values of $k$, $1 \le k \le n$, are equally likely, the expected number of exchanges would be approximately

$$
\begin{aligned}
E_{\text{ave}}(n) &\approx (1/n) \textstyle\sum_{k=1}^{n} k(n-k)/n \\
&\approx (1/n^2)\ \big(\textstyle\sum_{k=1}^{n} kn - \sum_{k=1}^{n} k^2\big) \\
&\approx (1/n^2)\ (n^3/2 - n^3/3) \\
\boldsymbol{E_{\text{ave}}(n)} &\approx \boldsymbol{n/6}
\end{aligned}
$$

In other words, *partition*() performs only about 1 exchange for every 6 comparisons. An alternate version, designed specifically to work with moves, performs about one move for each 3 comparisons.

*partition*() does an extremely good job of minimizing the movement of elements. This is probably why quicksort tends to be faster than merge-sort in the expected case, even though it performs move comparisons

Here is the tree of recursive calls to quicksort. Calls to sort subarrays of size 0 or 1 are not shown. (They could be omitted.)



# The Quicksort Algorithm

**(each interval partitioned using its middle element)**

***partition*( *A*, *left*, *right*)** rarranges $A[left\,..\,right]$ and finds and returns an integer $q$, such that

$$A[left], \ldots, A[q-1] \lesssim pivot, \quad A[q] = pivot, \quad A[q+1], \ldots, A[right] > pivot,$$

where *pivot* is the middle element of $a[left..right]$, before partitioning**.** (To choose the pivot element differently, simply modify the assignment to $m$.)

```
Integer partition( T[] A, Integer left, Integer right)
    m = ⌊left + right⌋ / 2;
    swap( A[left],  A[m]);
    pivot = A[left];
    lo = left+1;  hi = right;
    while ( lo ≤ hi )
        while ( A[hi] > pivot )
            hi = hi – 1;
        while ( lo ≤ hi  and  A[lo] ≲ pivot )
            lo = lo + 1;
        if ( lo ≤ hi )
            swap( A[lo], A[hi]);
            lo = lo + 1;  hi = hi – 1;
    swap( A[left], A[hi]);
    return hi
```

***quicksort*( *A*, *left*, *right*)** sorts $A[left\,..\,right]$ by using *partition*() to partition $A[left\,..\,right]$, and then calling itself recursively twice to sort the two subarrays.

```
void quicksort( T[] A, Integer left, Integer right)
   if ( left < right )
        q = partition( A, left, right);
        quicksort( A, left, q–1);
        quicksort( A, q+1, right);
```