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.

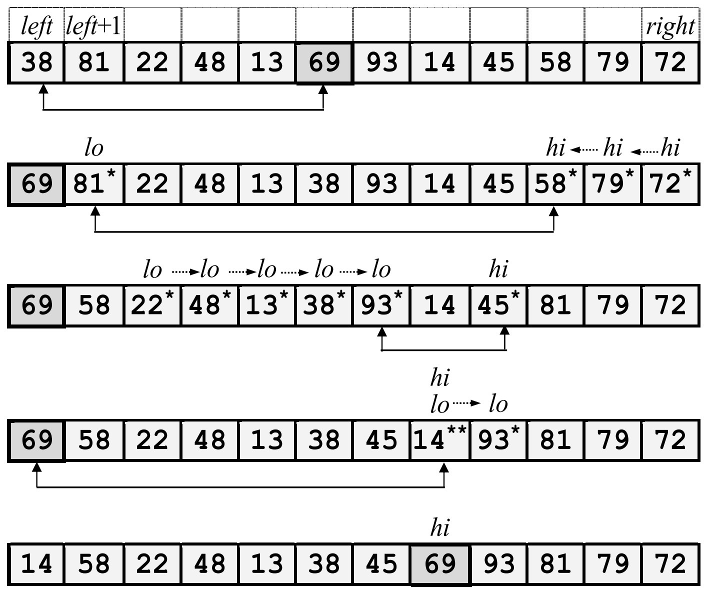
Swap pivot element with leftmost element. *lo*=*left*+1; *hi*=*right*;

Move *hi* left and *lo* right as far as we can; then swap *A*[*lo*] and *A*[*hi*], and move *hi* and *lo* one more position.

Repeat above

Repeat above until *hi* and *lo* cross; then *hi* is the final position of the pivot element, so swap *A*[*hi*] and *A*[*left*].

Partitioning complete; return value of *hi*.



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 kth largest element of the n elements, so it ends up in the kth position.

Each exchange of $A[lo]$ and $A[hi]$ moves one element initially to the right of kth position, but less than (or equivalent but not equal to) the pivot element, to a position not right of the kth 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 kth position. Thus we expect $(k-1)(n-k)/(n-1) \approx k(n-k)/n$ exchanges.

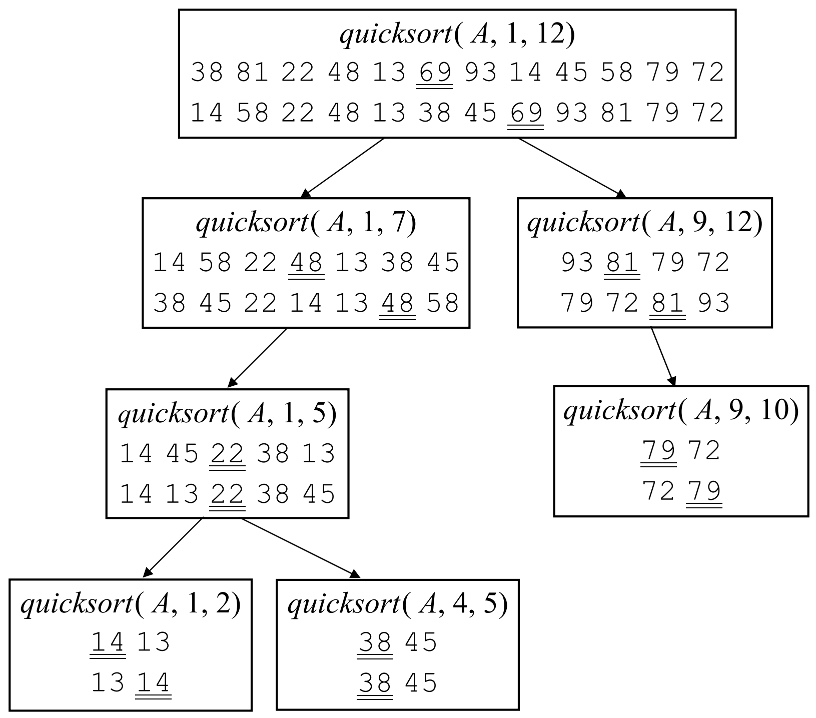
Since all values of k, $1 \le k \le n$, are equally likely, the expected number of exchanges would be approximately

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

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)
    $m = \lfloor left + right \rfloor / 2$;
    $swap( A[left], A[m])$;
    $pivot = A[left]$;
    $lo = left+1$; $hi = right$;
    while ( $lo \le hi$ )
        while ( $A[hi] > pivot$ )
            $hi = hi - 1$;
        while ( $lo \le hi$ and $A[lo] \lesssim pivot$ )
            $lo = lo + 1$;
        if ( $lo \le hi$ )
            $swap( A[lo], A[hi])$;
            $lo = lo + 1$; $hi = hi - 1$;
    $swap( A[left], A[hi])$;
    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 ( $left < right$ )
        $q = partition( A, left, right)$;
        $quicksort( A, left, q-1)$;
        $quicksort( A, q+1, right)$;
```

Later we will show that quicksort runs in **$\Theta(n^2)$ time** in the worst case, but that it runs in **$\Theta(n \lg(n))$ time** in the expected case, assuming the input is randomly ordered.

At the moment, let us consider the extra space (in addition to the array being sorted) used by quicksort.

- *partition*() uses only constant extra space.
- At first, it might appear that *quicksort*() also uses only constant extra space.
  - But this can never be the case for a recursive function (unless the maximum depth of recursion is bounded by a constant).
  - Each time a function is called (recursively or otherwise), we get a new activation of the function. The activation continues to exist until the function returns to its caller.
  - Each activation has its own *activation record*, or *stack frame*, on the run-time stack.
    - The activation record may contain space for automatic local variables and parameters, a return address, and an area to save the current status.
  - For a recursive function, the memory requirements are determined by the maximum number of activations existing at any one time, that is, by the maximum depth of recursion.
- For *quicksort*() as we have implemented it, the maximum depth of recursion will be n in the worst case.
  - This would occur if every call to *partition*() produced as uneven a split as possible (sublist sizes $n-1$ and 0).

  - It would make the extra space requirements $\Theta(n)$ — possibly with a fairly large constant multiplying n.

- With a little effort, we can limit the maximum depth of recursion to approximately $\lg(n)$, even in the worst case.
  - Then the extra space required will be $\Theta(\lg(n))$ — insignificant compared to the $\Theta(n)$ space for the array being sorted.
  - Note, however, the worst case running time remains $\Theta(n^2)$.

quicksort2(A, left, right) performs the same function as *quicksort*(*A*, *left*, *right*), but the depth of recursion is limited to $\lg(n)$, and the extra space is only $\Theta(\lg(n))$ even in the worst case.

```
void quicksort2( T[] A, Integer left, Integer right)
    while ( left < right )
        q = partition( A, left, right);
        if ( q – left < right – q )
            quicksort2( A, left, q–1);   // Recursive call sorts smaller sublist
            left = q + 1;                // Loop back to sort larger sublist
        else
            quicksort2( A, q+1, right);  // Recursive call sorts smaller sublist
            right = q – 1;               // Loop back to sort larger sublist
```

Each time *quicksort2*() calls itself recursively, it does so only for the smaller of the two sublists. This means that each time the depth of recursion rises by one, the size of the input to quicksort2() is cut at least in half. Thus the depth of recursion cannot exceed $\lg(n)$.

Note the larger sublist is sorted by repeating the while-loop with the value of *left* or *right* adjusted to correspond to the larger sublist, rather than the original list. This requires no additional space.

Performance of Quicksort

We will count the number $C(n)$ of comparisons performed by quicksort in sorting an array of size n.

We have seen that *partition*() performs n comparisons (possibly $n-1$ or $n+1$, depending on the implementation).

In fact, $n-1$ is the lower bound on the number of comparisons that any partitioning algorithm can perform.

The reason is that every element other than the pivot must be compared to the pivot; otherwise we have no way of knowing whether it goes left or right of the pivot.

So our recurrence for $C(n)$ is:

$$\boxed{C(n) = \underbrace{n}_{\textit{Partition}} + \underbrace{C(k-1)}_{\textit{Sort left subarray by recursive call to quicksort}} + \underbrace{C(n-k)}_{\textit{Sort right subarray by recursive call to quicksort}},\ \ C(0) = C(1) = 0}$$

(k = final position of pivot element)

A bad case (actually the worst case): At every step, *partition*() splits the array as unequally as possible ($k = 1$ or $k = n$).

Then our recurrence becomes

$$C(n) = n + C(n-1),\ \ C(0) = C(1) = 0$$

This is easy to solve.

$$\begin{aligned}
C(n) &= n + C(n-1) \\
&= n + n-1 + C(n-2) \\
&= n + n-1 + n-2 + C(n-3) \\
&= n + n-1 + n-2 + \dots + 3 + 2 + \overset{0}{\cancel{C(1)}} \\
&= (n + n-1 + n-2 + \dots + 3 + 2 + 1) - 1 \\
&= n(n+1)/2 \ - 1 \\
&\approx n^2/2
\end{aligned}$$

This is terrible. It is no better than simple quadratic time algorithms like straight insertion sort.

A good case (actually the best case): At every step, *partition*() splits the array as equally as possible ($k = (n+1)/2$; the left and right subarrays each have size $(n-1)/2$)).

This is possible at every step only if $n = 2^k - 1$ for some k. However, it is always possible to split nearly equally. The recurrence becomes

$$C(n) = n + 2C((n-1)/2),\ \ C(0) = C(1) = 0,$$

which we approximate by

$$C(n) = n + 2C(n/2),\ \ C(1) = 0$$

This is the same as the recurrence for mergesort, except that the right side has n in place of $n-1$. The solution is essentially the same as for mergesort:

$$C(n) = n\lg(n).$$

This is excellent — essentially as good mergesort, and essentially as good as any comparison sorting algorithm can be.

The expected case: Here we assume either (i) the array to be partitioned is randomly ordered, or (ii) the pivot element is selected from a random position in the array.

In either case, the pivot element will be a random element of the array to be partitioned. That is, for $k = 1, 2, ..., n$, the probability that the pivot element is the k^{th} largest element of the array is $1/n$. (Recall that, if the pivot element is the k^{th} largest element of the array, it ends up after partitioning in position k.)

In the recurrence

$$C(n) = n + C(k-1) + C(n-k),\ \ C(0) = C(1) = 0,$$

all values of k are equally likely. We must average over all k.

$$\begin{aligned} C(n) &= (1/n) \textstyle\sum_{k=1}^{n} \big(n + C(k-1) + C(n-k)\big), \quad C(0)=C(1)=0, \\ &= n + (1/n)\textstyle\sum_{k=1}^{n} C(k-1) + (1/n)\sum_{k=1}^{n} C(n-k) \end{aligned}$$

Note: $\sum_{k=1}^{n} C(k-1) = \sum_{i=0}^{n-1} C(i)$, by substituting $i = k-1$.
$\sum_{k=1}^{n} C(n-k) = \sum_{i=0}^{n-1} C(i)$, by substituting $i = n-k$.

So our recurrence becomes

$$C(n) = n + (2/n)\textstyle\sum_{i=0}^{n-1} C(i), \quad \text{or}$$

$$n\,C(n) = n^2 + 2\textstyle\sum_{i=0}^{n-1} C(i)$$

Writing down the same recurrence with $n-1$ replacing n, we get

$$(n-1)\,C(n-1) = (n-1)^2 + 2\textstyle\sum_{i=0}^{n-2} C(i).$$

Subtracting this recurrence from the one above it gives

$$n\,C(n) - (n-1)\,C(n-1) = n^2 - (n-1)^2 + 2\,C(n-1), \ \ \text{or}$$

$$n\,C(n) = (n+1)\,C(n-1) + 2n-1$$

Dividing by $n(n+1)$ gives

$$C(n)/(n+1) = C(n-1)/n + (2n-1)/(n(n+1)).$$

To a very good approximation,

$$C(n)/(n+1) = C(n-1)/n + 2/n.$$

Now if let $D(n) = C(n)/(n+1)$, then the recurrence becomes

$$D(n) = D(n-1) + 2/n, \quad D(1) = 0.$$

This is easy to solve:

$$\begin{aligned} D(n) &= D(n-1) + 2/n \\ &= D(n-2) + 2/(n-1) + 2/n \\ &= D(n-3) + 2/(n-2) + 2/(n-1) + 2/n \\ &= \overset{0}{\cancel{D(1)}} + 2/2 + 2/3 + \ldots + 2/(n-2) + 2/(n-1) + 2/n \\ &= 2\ln(n) - 2 \\ &\approx 2\ln(n) \\ &= 2\ln(2)\lg(n) \\ &\approx 1.39\lg(n) \end{aligned}$$

So $C(n) = (n+1)\,D(n) \approx 1.39\,(n+1)\,\lg(n)$, or $\boxed{C(n) \approx 1.39\,n\lg(n)}$

The expected case for quicksort is fairly close to the best case (only 39% more comparisons) and nothing like the worst case.

In most (not all) tests, quicksort turns out to be a bit faster than mergesort.

Quicksort performs 39% more comparisons than mergesort, but much less movement (copying) of array elements.

We saw that, in the expected case, quicksort performs one exchange for every six comparisons, or about $1.39\,n\lg(n)/6 \approx 0.23\,n\lg(n)$ exchanges.

A slightly different partitioning algorithm performs one move (copy) for each three comparisons, or about $0.46\,n\lg(n)$ moves.

By contrast, the version of mergesort given in class performs $2\,n\lg(n)$ moves, although this can be reduced to $n\lg(n)$ moves — still more than twice as many as quicksort is likely to perform.

With a randomized version of quicksort (pivot element chosen randomly), the standard deviation in the number of comparisons is also small.

The probability of performing substantially more than $1.39\,n\lg(n)$ comparisons is extremely low.

Quicksort is not stable, since it exchanges nonadjacent elements.

If stability is not required, quicksort provides a very attractive alternative to mergesort.

Quicksort is likely to run a bit faster than mergesort — perhaps 1.2 to 1.4 times as fast.

Quicksort requires less memory than mergesort.

A good implementation of quicksort is probably easier to code than a good implementation of mergesort.