Here is an analysis of the time complexity of quick-sort in detail.

In quick sort, we pick an element called the pivot in each step and re-arrange the array in such a way that all elements less than the pivot now appear to the left of the pivot, and all elements larger than the pivot appear on the right side of the pivot. In all subsequent iterations of the sorting algorithm, the position of this pivot will remain unchanged, because it has been put in its correct place. The total time taken to re-arrange the array as just described, is always $O(n)$[1], or αn where α is some constant [**see footnote**]. Let us suppose that the pivot we just chose has divided the array into two parts - one of size k and the other of size $n-k$. Notice that both these parts still need to be sorted. This gives us the following relation:

$T(n) = T(k) + T(n-k) + \alpha n$

where $T(n)$ refers to the time taken by the algorithm to sort n elements.

WORST CASE ANALYSIS:

Now consider the case, when the pivot happened to be the least element of the array, so that we had $k = 1$ and $n - k = n - 1$. In such a case, we have:

$T(n) = T(1) + T(n-1) + \alpha n$

Now let us analyse the time complexity of quick sort in such a case in detail by solving the recurrence as follows:

$T(n) = T(n-1) + T(1) + \alpha n$

$= [T(n-2) + T(1) + \alpha(n-1)] + T(1) + \alpha n$

(Note: I have written $T(n-1) = T(1) + T(n-2) + \alpha(n-1)$ by just substituting $n-1$ instead of n. Note the implicit assumption that the pivot that was chosen divided the original subarray of size $n-1$ into two parts: one of size $n-2$ and the other of size 1.)

$= T(n-2) + 2T(1) + \alpha(n-1+n)$ (by simplifying and grouping terms together)

$= [T(n-3) + T(1) + \alpha(n-2)] + 2T(1) + \alpha(n-1+n)$

$= T(n-3) + 3T(1) + \alpha(n-2+n-1+n)$

$= [T(n-4) + T(1) + \alpha(n-3)] + 3T(1) + \alpha(n-2+n-1+n)$

[1]Here is how we can perform the partitioning: (1) Traverse the original array once and copy all the elements less than the pivot into a temporary array, (2) Copy the pivot into the temporary array, (3) Copy all elements greater than the pivot into the temporary array, and (4) Transfer the contents of the temporary array into the original array, thus overwriting the original array. This takes a total of $4n$ operations. There do exist ways to perform such a partitioning in-place, i.e. without creating a temporary array, as well. These techniques do speed up quicksort by a constant factor, but they are not important for understanding the overall quicksort algorithm.

$= T(n-4) + 4T(1) + \alpha(n-3+n-2+n-1+n)$

$= T(n-i) + iT(1) + \alpha(n-i+1+ ..... +n-2+n-1+n)$ (Continuing likewise till the i^{th} step.)

$= T(n-i) + iT(1) + \alpha(\sum_{j=0}^{i-1}(n-j))$ (Look carefully at how the summation is being written.)

Now clearly such a recurrence can only go on until $i = n-1$ (Why? because otherwise $n-i$ would be less than 1). So, substitute $i = n-1$ in the above equation, which gives us:

$T(n) = T(1) + (n-1)T(1) + \alpha\sum_{j=0}^{n-2}(n-j)$

$= nT(1) + \alpha(n(n-2) - (n-2)(n-1)/2)$ (Notice that $\sum_{j=0}^{n-2} j = \sum_{j=1}^{n-2} j = (n-2)(n-1)/2$ by a formula we earlier derived in class)

which is $O(n^2)$.

This is the worst case of quick-sort, which happens when the pivot we picked turns out to be the least element of the array to be sorted, in every step (i.e. in every recursive call). A similar situation will also occur if the pivot happens to be the largest element of the array to be sorted.

BEST CASE ANALYSIS:

The best case of quicksort occurs when the pivot we pick happens to divide the array into two exactly equal parts, in every step. Thus we have $k = n/2$ and $n-k = n/2$ for the original array of size n.

Consider, therefore, the recurrence:

$T(n) = 2T(n/2) + \alpha n$

$= 2(2T(n/4) + \alpha n/2) + \alpha n$

(Note: I have written $T(n/2) = 2T(n/4) + \alpha n/2$ by just substituting $n/2$ for n in the equation $T(n) = 2T(n/2) + \alpha n$.)

$= 2^2T(n/4) + 2\alpha n$ (By simplifying and grouping terms together).

$= 2^2(2T(n/8) + \alpha n/4) + 2\alpha n$

$= 2^3T(n/8) + 3\alpha n$

$= 2^kT(n/2^k) + k\alpha n$ (Continuing likewise till the k^{th} step)

Notice that this recurrence will continue only until $n = 2^k$ (otherwise we have $n/2^k < 1$), i.e. until $k = \log n$. Thus, by putting $k = \log n$, we have the following equation:

$T(n) = nT(1) + \alpha n \log n$, which is $O(n \log n)$.

This is the best case for quicksort.

It also turns out that in the average case (over all possible pivot configurations), quicksort has a time complexity of $O(n \log n)$, the proof of which is beyond the scope of our class.

AVOIDING THE WORST CASE

Practical implementations of quicksort often pick a pivot randomly each time. This greatly reduces the chance that the worst-case ever occurs. This method is seen to work excellently in practice. The other technique, which determinstically prevents the worst case from ever occurring, is to find the median of the array to be sorted each time, and use that as the pivot. The median can (surprisingly!) be found in linear time (the details of which are, again, beyond the scope of our course) but that is saddled with a huge constant factor overhead, rendering it suboptimal for practical implementations.