Quick sort is a divide and conquer algorithm which spends most of the effort in the divide step and very little in the conquer step.

Suppose we know a number $x$ such that one-half of the elements are greater than or equal to $x$ and one-half of the elements are smaller than $x$. We can compare all elements by $n$ comparisons and partition the sequence into equal parts (we can do without additional space as we see). This is the divide step. We sort each subsequence recursively and the combine step is trivial since the two parts already occupy correct places in the array (no additional space).

Unfortunately we do not know $x$, but it works no matter which number is selected, however we do it in-place, we use two pointers to the array $L$ and $R$, where $L$ points to the left side of the array and $R$ points to the right side of the array. The pointers move toward each other such that:

**Induction hypothesis:** At step $k$ of the algorithm $\forall i, \forall j \forall k < i \land j > k$, and $\forall i \land j \geq k$.

The base case is trivial and when $L \geq R$ we are done.

Two cases to consider: if either $x_i < \text{pivot}$ or $x_i > \text{pivot}$ we can move one of them and IH is preserved. Otherwise $x_i > \text{pivot}$ and $x_j < \text{pivot}$ in this case we do a swap (exchange) and move both of them. The whole partition needs only $O(n)$ comparisons.

Divide and conquer algorithm works best when the parts have equal sizes (i.e. pivot should be the median). However choosing a random element from the sequence is usually more effective than choosing the pivot, so we can find it also, but not in this class.

At the end of the partition we put the pivot at its correct place, i.e. swap($x_i$, $x_j$).

Having the preceeding partition, Quick sort is as follows:

```plaintext
procedure Q_Sort(L, R, X)
begin
if L < R then
    Partition(X, L, R)
Q_Sort(L, pivot, index - 1, X)
end;
```
The running time of quicksort depends on the particular input. If the pivot always partitions the sequence into two equal parts then the running time is \( T(n) = 2T(n/2) + O(n) \), \( T(2) = 1 \) which is \( T(n) = O(n \log n) \) (similar to the merge sort).

However, if the pivot is very close to one side of the sequence the running time can be as bad as \( O(n^2) \): say each time the pivot is the smallest element in the sequence, we do \( n-1 \) comparisons and place only the pivot in the right place; then \( n-2 \) comparison, etc.

In the algorithm, we are choosing the pivot at random, but still there is a chance that we pick the smallest element and thus be in \( O(n^2) \) comparisons. However, the chance is small as we analyze.

First each element has the same probability \( \frac{1}{n} \). If ith element is selected as the pivot the running time is \( T(n) = n-1 + T(i-1) + T(n-i) \).

Thus the average running time (as worst-case running time considered is this class) of

\[
T(n) = \frac{1}{n} \sum_{i=1}^{n} (n-1 + T(i-1) + T(n-i)) = n-1 + \frac{1}{n} \sum_{i=1}^{n} T(i-1) + T(n-i)
\]

\[= n-1 + \frac{1}{n} \sum_{i=1}^{n} T(i-1) + \frac{1}{n} \sum_{i=1}^{n} T(n-i) = n-1 + \frac{2}{n} \sum_{i=1}^{n} T(i) \] 0

As we have seen in the recurrence relation with full history \( T(n) = O(n \log n) \).

In practice quicksort is very fast and it deserves the name.

**Lower bound for sorting:**

We could improve the running time from \( O(n^2) \) (insertion or selection sort) to \( O(n \log n) \) (for merge sort or quicksort). Can we do still better for comparison based sorting (and not radix sort, for example). Unfortunately, NO.

We can use information theory to show that indeed any algorithm needs at least \( \Omega(n \log n) \) comparisons (see the proof in the book if you are interested).

**Maximum and minimum elements:**

The problem: Find the maximum and minimum elements in a given array of distinct elements.

**Straightforward solution:** \( 2n-3 \) (n-1 for max and then n-2 for the min)
Can we do better?

If we try to inductively go from \( n \) to \( n-1 \) we pay \( 2n \).

but what about going from \( n \) to \( n-2 \). Assume we know the solution for \( n-2 \). Consider \( x_{n-1} \) and \( x_n \) and say we know \( \text{Max} \) and \( \text{Min} \) so far. First we compare \( x_{n-1} \) and \( x_n \).

and then compare the \( \text{Max} \) of them with \( \text{Max} \) and \( \text{Min} \) of them by \( \text{Min} \). So by 3 comparisons we update both \( \text{Max} \) and \( \text{Min} \). Overall we need \( \frac{3}{2} \) comparisons. Can we do better by considering \( x_{n-2}, x_{n-1}, x_n \)? The answer is no by any method.

Order statistic or selection problem:

The problem: Given a sequence \( S = x_1, x_2, \ldots, x_n \) of elements, and an integer \( k \) such that \( 1 \leq k \leq n \), find the \( k \)-th smallest element in \( S \).

If \( k \) is very close to \( 1 \) (or \( n \)), we can run the algorithm for \( \text{Max} \) or \( \text{Min} \) \( k \) times, so the running time is \( O(k) \). We can also sort the sequence in \( O(n \log n) \) and then obtain the \( k \)-th element in \( O(n) \), so if \( k = O(\log n) \), i.e. \( k = \Omega(\log n) \), sort is better.

But can we do better? Yes, we can do divide and conquer the same way as quick sort. In quick sort, the sequence is partitioned by a pivot into two subsequences. Here we need only to determine which subsequence contains the \( k \)-th element and then continue recursively only for that subsequence. The rest of the elements can be ignored.

Running time: As in quick sort, the worst case is \( O(n^2) \), but we can show the average number is \( O(n) \). In practice, it is very fast (though there are algorithms with running time \( O(n) \) in the worst case). Indeed, this algorithm is the best for finding only the median, i.e., \( k = \frac{n}{2} \).

Read also sections 6.90 and 6.11 yourself. Due to lack of time we cannot cover them in the class, but there are very good algorithms here and you enjoy reading them.