Sorting:

one of the most extensively studied problems in computer science. It is the basis for many algorithms and it consumes a large proportion of computing time for many typical applications. There are dozens of sorting algorithms but we cover only a few.

The problem: Given n numbers $x_1, x_2, \ldots, x_n$ arrange them in increasing order. In other words, find a sequence of distinct $1 \leq i_1 < i_2 < \cdots < i_n$ such that $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_n}$. We assume all numbers are distinct though all algorithms in the class work for non-distinct numbers as well.

A sorting algorithm is called in-place if no additional work space is used besides the initial array that hold the elements.

--Insertion Sort (sort by induction):

Suppose we know how to sort n-1 numbers and we are given n numbers. We can sort the n-1 numbers and then put the nth number in its correct place by scanning the n-1 sorted numbers until the correct place to insert is found.

The total number of comparisons for sorting n numbers may be as high as $1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$. Also for insertion, thus moving, in the worst case, we need n-1 elements to be moved and hence the total number of movements is also $O(n^2)$. We can have the elements in the array and use binary search on the sorted elements. Then the total comparisons is $\sum_{i=1}^{n} \log_2 i = \Theta(n \log n)$ as we have seen before. However the number of movements is still $O(n^2)$.

Selection Sort: We can select the maximal number as the nth number and put it in the end of array (by swapping it with whatever there). We recursively sort the rest.

The advantage of insertion sort is that only n-1 data movements (swap) are required versus $O(n^2)$ of insertion sort. However, it takes n-1 comparisons to find the maximal element, with total $O(n^2)$ versus $O(n \log n)$ comparisons of insertion.

Using other data structures such as AVL trees or binary search trees we can do comparison in $O(\log n)$, we will cover binary search trees later in this course.

In bubble sort we swap in the unsorted part of the array (a bit of waste) [if $A[i] < A[i+1]$ it swaps $A[i] \text{ and } A[i+1]$] while in selection sort we only keep the index of the maximal element.
merge sort (you have seen it before, just in case)

merge operation: denote the first set by a[1], a[2], ..., an and the second set by b[1], b[2], ..., bm and assume both are sorted in increasing order. Scan the first set until the right place to insert b1 is found and insert it, then continue the scan from that place until the right place to insert b2 is found, and so on. Since bs are sorted, we never need to go back.
The total number of movements is O(n + m).  

Data movement is inefficient if we insert it, however, if we use a temporary array each element is copied exactly once, and thus the overall time is O(n + m). It is not an inplace sort.

merge sort is a divide-and-conquer (recursive) algorithm as follows.

First: divide the sequence into close-to-equal size.
Second: sort each part separately recursively.
Third: merge the two parts into one sorted array.

Time complexity, if T(n) is the total time for sorting n numbers:

\[ T(2n) = 2T(n) + O(n), \quad T(2) = 1. \]

As we have seen in chapter 3 (master theorem), it is \( O(n \log n) \) which is much better than insertion or selection sorts. However, it requires additional storage to copy the merge set and not an inplace sort.