3~~8~~9

Quick_Sort:

As we have seen in the merge sort, by divide and conquer, we could divide the problem into two equal-sized subproblems, solve each subproblem separately and combine the solutions to get $O(n \log n)$ algorithm, but we needed to have extra space (it was Not an in-place sort).

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

suppose we know a number x (called pivot) 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 positions in the array (no additional space).

Unfortunatlly we do not know x, but it works no matter which number is selected however to 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, $\text{pivot} \geq x_i$ for all i such that $i < L$ and $\text{pivot} < x_j$ for all $j > R$.

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

Two cases to consider: if either $x_L \leq \text{Pivot}$ or $x_R > \text{Pivot}$ we can move one of them and IH is preserved. otherwise $x_L > \text{Pivot}$ and $x_R \leq \text{Pivot}$ in this case we do a <u>swap</u> (exchange) and move both of them. The whole partition needs $O(n)$ 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 a good choice as we see in the analysis.

$\boxed{\text{we can find it also, but not in this class}}$

At the end of partion we put the pivot at its correct place, i.e. swap(x_{pivot}, x_L).
having the procedure Partion, Q_sort is as follows:

```
procedure Q_Sort(L, R; X)
begin
    if L < R then Partition(X, L, R)
    Q_Sort(L, Pivot index - 1; X)
    Q-Sort(pivot index + 1, R; X)
end;
```

Running time:

the running time of quicksort depends on the particular input.
If the pivots always partitions the sequence into two equal parts then the running time is $T(n) = 2T(\frac{n}{2}) + O(n)$, $T(2)=1$ 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 $n-2$ comparisons, 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 (vs worst-case running time considered in this class so far)

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

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

As we have seen in the recurrence relation with full history $T(n) = O(n \log n)$.
In practice Quick sort 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 quick sort). 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.
straight forward 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 go from n to $n-2$. Assume we know the solution for $n-2$. consider x_{n-1} and x_n and say we know Max and Min so far. first we compare x_{n-1} and x_n and then compare the max of them with Max and min of them by Min. So by 3 comparisions we update both max and min. overall we need $\frac{3n}{2}$ 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 $S = x_1, x_2, \dots, x_n$ of elements, and an integer k such that $1 \leq k \leq n$, find the kth-smallest element in S.

If k is very close to 1 (or n), we can run the algorithm for max or min k times so the running time is $O(kn)$. we can also sort the sequence in $O(n \log n)$ and then obtain the kth element in $O(n)$. So if $O(kn) > O(n \log n)$, i.e. $k = \Omega(\log n)$ sort is better.

But can we do better? Yes, we can ~~sort~~ 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 kth 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 6.10 and 6.11. yourself. Due to lack of time we cannot cover them in the class but there are very qute algorithms there and you enjoy reading them.