Chapter 9
Medians and Order Statistics

The $i^{\text{th}}$ order statistics of a set of $n$ elements is the $i^{\text{th}}$ smallest element. For example, the minimum of a set of elements is the first order statistics ($i = 1$), and the maximum one is the $n^{\text{th}}$ order statistics ($i = n$). The median of a set is the "halfway point" of the set. When $n$ is odd, the median is unique, and $i = (n + 1)/2$; but when $n$ is even, there are two such medians, occurring at $i = n/2$, and $i = n/2 + 1$. We refer to the first as the lower median and use it in place of the median when there are two of them.

It is obvious that if we sort the set first in $O(n \log n)$, then it is straightforward to find out the $i^{\text{th}}$ order statistics of that set in $\Theta(1)$, with the total running time being $O(n \log n)$.

Is there some better way to get them?
The selection problem

Assume that the set we are interested in contains only distinct elements, then the *selection* problem can be specified as follows:

**Input:** a set $A$ of $n$ distinct numbers and a number $i \in [1, n]$.

**Output:** The element $x \in A$ that is larger than exactly $i - 1$ other elements of $A$.

Thus, e.g., solving the selection problem for a set $A$ and $i = 1$ will find out the minimum element of $A$. 
To get the minimum

It is easy to show that it takes at least \( n - 1 \) comparisons to find out the minimum element of a set containing \( n \) elements. The idea is the same as to pick a winner among \( n \) candidates in steps, each of which will drop only one. Thus, to drop \( n - 1 \) candidates needs at least \( n - 1 \) comparisons. Hence, this problem takes \( n - 1 \) as its lower bound.

Below shows an algorithm that finds the minimum, among the first programming assignments.

\[
\text{MINIMUM}(A) \\
1. \text{min} \leftarrow A[1] \\
2. \text{for } i \leftarrow 2 \text{ to } \text{length}[A] \\
3. \quad \text{do if } \text{min} > A[i] \\
\quad \quad \text{then } \text{min} \leftarrow A[i]
\]
How about the maximum?

The above algorithm can also be used to find out the maximum of $A$ with just a flip of the comparator.

Since it takes exactly $n - 1$ comparisons, it is the optimal solution.

Sometimes, we want to take the minimum and maximum simultaneously. For example, a GUI program may need to fit a point $(x, y)$ in a rectangular screen, thus having the need to find out the minimum and the maximum of all the four corner coordinates at the same time.

It is immediate to design an algorithm, based on the previous one, to solve this *simultaneous minimum and maximum* problem in $2n - 2$ comparisons.
A better way

With the previous algorithm, with an element, we compare it with the current minimum, then the maximum, to make a progress. This takes 2 comparisons per element, thus $2(n - 1)$ in total.

Assume that $n$ is odd, we set both the minimum and the maximum to be $A[1]$. Then, for the rest $\frac{n-1}{2}$ pairs of numbers, make a comparison of them first, then compare the smaller with the current minimum and the larger with the current maximum.

This way, we make three comparisons per every two elements, with the total comparisons being $3 \lfloor \frac{n}{2} \rfloor$, a whopping 25% cut.

This approach also works when $n$ is even: we just make a comparison of the first two numbers and set up the initial value of minimum and maximum accordingly.
Running time

When $n$ is odd, the total number of comparisons is $3\frac{n-1}{2} = \frac{3n-3}{2}$; while in the other case, after making the initial comparison, we further make $3\frac{n-2}{2}$ comparisons for a total of $\frac{3n-6}{2} + 1$, i.e.,

$$\frac{3n - 4}{2} = \frac{3n}{2} - 2.$$  

When $n$ is even, we have that

$$\frac{3n}{2} - 2 < 3\frac{n}{2} = 3 \left\lfloor \frac{n}{2} \right\rfloor.$$  

When $n$ is odd, we have that, $n = 2m + 1$. Thus,

$$3n - 3 \quad 2 = 3m = 3 \left\lfloor \frac{n}{2} \right\rfloor.$$  

Hence, in all the cases, the number of comparisons is at most $3 \left\lfloor \frac{n}{2} \right\rfloor$, saving about 25% of the work.
Solving the general problem

The general selection problem, i.e., for a given $A$ and $i$, finds out the $i^{th}$ order statistics of $A$, seems to be more difficult than merely finding out the minimum element of $A$. We will show that we can do it as fast.

The basic idea is to follow a divide-and-conquer approach, similar to what we did for quicksort.

Time wise, the difference is that instead of recursively working on both sublists, we will only work on one, which leads to an expected running time of only $\Theta(n)$. 
The code

R-SELECTION(A, p, r, i)
1. if p=r
2. then return A[p]
3. q<-RANDOMIZED-PARTITION(A, p, r)
4. k<-q-p+1
5. if i=k
6. //the pivot value is the answer
7. //since the i items less than or
8. //equal to the pivot
9. then return A[q]
10. else if i<k
11. then return R-SELECTION(A,p,q-1,i)
12. else return R-SELECTION(A,q+1,r,i-k)
What is happening?

After the partition in line 3, the list is cut into two (possibly empty) sub lists, $A[p..q-1]$ and $A[q+1..r]$ such that everything in the first list is less than or equal to the pivot, $A[q]$, and everything in the second is larger than the pivot. We don’t know the ordering information of the elements in both lists, but it is not needed for the task at hand.

Line 4 then finds out the number of elements in the first list, plus the one of the pivot. We now have three cases: 1) If the number just found out is equal to $i$, the pivot is in the right position. 2) If $i$ is less than this number, then everything in the second list does not matter, we only need to find out the $i^{th}$ order statistics in the first list. 3) The last possibility is that $i$ is larger than that number, then, we only need to look for this value in the second list, while making an adjustment to its order.
An example

Given a list \((8, 6, 4, 5, 3, 2, 1, 10, 9, 7)\), and assume that after the partition, we get \((4, 3, 2, 1), 5, (8, 6, 10, 9, 7)\). We have \(k=5\).

If \(i=k=5\), then from the partitioned list, it is clear that there are 4 items less than 5, so \(A[5]=5\) should be sent back.

If \(i<k\), e.g., \(i=3\). It is also clear that for each and every item in the upper list, it is at least the \(k^{th}\) item, in this case, the \(5^{th}\), in the list, thus can’t be the \(i^{th}\) item, the \(3^{rd}\) in this case, thus can be dropped. The one we are looking for has to be the \(i^{th}\) item in the lower part.

Finally, if \(i>k\), e.g., \(i=7\), then it is also the \((i-k)^{th}\), in this case, the \(2^{nd}\) smallest in the upper half.

**Homework:** Exercise 9.2-4(*).
The running time

In the best case, we find it in Step 6, after running the partitioning once, thus $\Theta(n)$.

In the worst case, we can end up with $\Theta(n^2)$, since we may be extremely unlucky, and always get an empty list in the partition.

But, it works well on average, particularly, because it is an randomized algorithm.

Let $T_i(n)$ be the average number of comparisons we have to make when selecting the $i^{th}$ order in a list of size $n$ with $|S_1| = k \in [0, n-1]$, similar to what we did for the quicksort case, we have the following:

$$
\overline{T_i}(n) = \frac{1}{n} \sum_{k=0}^{n-1} T_i(k) + cn.
$$
We now have
\[ nT_i(n) = \sum_{k=0}^{n-1} T_i(k) + cn^2. \]
and
\[ (n-1)T_i(n-1) = \sum_{k=0}^{n-2} T_i(k) + c(n-1)^2. \]
Take their difference, we have
\[ nT_i(n) - (n-1)T_i(n-1) = T_i(n-1) + c(2n - 1), \]
namely,
\[ T_i(n) = T_i(n-1) + c\left(2 - \frac{1}{n}\right) \]
\[ = \cdots = 2(n-1) + c(H(n) - 1) \]
\[ = \Theta(n). \]

The book contains another derivation in terms of probabilistic analysis.