Selection
(read Section 9 on Medians and Order Statistics in Cormen, Leiserson, Rivest, Stein)

We study the problem of finding the $i$-smallest item in a given collection of $n$ items. The items are given in an unsorted list. We could first sort the list and then return the item in the $i$-th position, but just finding the $i$-th item can be done faster than sorting the entire list. As a warm-up exercise consider selecting the smallest item in the unsorted array.

```
for j = 2 to n do
    if A[j] < A[min] then min = j endif
endfor
```

The index of the smallest item is found in $n - 1$ comparisons, and it is not possible to use fewer than $n - 1$ comparisons. Why not?

**Randomized Selection.** We return to finding the $i$-smallest item for a fixed but arbitrary integer $1 \leq i \leq n$, which we call the *rank* of that item. We can use the splitting function of quicksort also for selection. Like in quicksort we choose a random pivot and use it to split the array, but we recurse only for one of the two sides. We invoke the function with the range of indices of the current subarray and the rank of the desired item, $i$. Initially, the range consists of all indices between $1$ and $n$, limits included.

```
int RSelect(int l, r, i)
    assert 1 \leq i \leq r - \ell + 1;
    if \ell = r then return \ell
    else q = RANDOMSPLIT(\ell, r);
        m = q - \ell;
        if i \leq m then return RSelect(\ell, q - 1, i)
            elseif i = m + 1 then return q
            else return RSelect(q + 1, r, i - m - 1)
    endif
    endif.
```

**Randomized Analysis.** For each $0 \leq m \leq n - 1$, the probability that the array is split into subarrays of sizes $m$ and $n - m - 1$ is $\frac{1}{2}$. The expected running time is then

$$T(n) \leq \frac{1}{n} \sum_{m=0}^{n-1} \max\{T(m), T(n - m - 1)\} + c \cdot n$$

$$\leq \frac{2}{n} \sum_{m=\lceil n/2 \rceil}^{n-1} T(m) + c \cdot n,$$

where $c > 0$ is some constant. Assume inductively that $T(m) \leq b \cdot m$ for $m < n$ and $b > 0$ large enough. Such a constant $b$ can certainly be found for $m = 1$, since for that case the running time of the algorithm is only a constant. This establishes the basis of the induction. The case of $n$ items reduces to cases of $m < n$ items for which we can use the induction hypothesis. Assuming $n$ is even, we get

$$T(n) \leq \frac{2b}{n} \sum_{m=\lceil n/2 \rceil}^{n-1} m + c \cdot n$$

$$= \frac{2b}{n} \sum_{m=1}^{n-1} m - \frac{2b}{n} \sum_{m=1}^{n/2 - 1} m + c \cdot n$$

$$= b \cdot (n - 1) - \frac{b}{2} \cdot \left( \frac{n}{2} - 1 \right) + c \cdot n$$

$$= \frac{3b}{4} \cdot n - \frac{b}{2} + c \cdot n$$

$$\leq b \cdot n,$$

provided $b \geq 4c$. We get the same inequality for odd $n$ but the calculation is a slightly more involved. In words, we just proved that the expected running time of $RSelect$ is at most four times the running time of $RANDOMSPLIT$. The latter is $O(n)$ which implies that the running time for randomized selection is also $O(n)$.

**Deterministic Selection.** The randomized selection algorithm takes time proportional to $n^2$ in the worst case,
We can now combine the various pieces and write the selection algorithm in pseudo-code. Recall that $i$ is the rank of the desired item in $A[\ell..r]$. After sorting the sets of 5 or 4 we have their medians arranged in the middle fifth of the array, $A[\ell + 2k..\ell + 3k - 1]$, and we can compute the median of the medians simply by recursive application of the function. For small arrays it is not worthwhile to do the splitting etc., and we just sort the array and return the item in the appropriate position. Is the only reason for this that expect a speed-up of the algorithm or is it in fact necessary because the algorithm fails for small array sizes?

```plaintext
int SELECT(int \ell, r, i)
if r - \ell + 1 \leq 50 then
    ISORT(\ell, 1, r); return A[\ell + i - 1]
else
    k = \lceil (r - \ell + 1)/5 \rceil;
    for j = 0 to k - 1 do
        ISORT(\ell + j, k, r)
    endfor;
    m' = SELECT(\ell + 2k, \ell + 3k - 1, [k/2]);
    SWAP(\ell, m'); q = SPLIT(\ell, r);
    m = q - \ell;
    if i \leq m then return SELECT(\ell, q - 1, i)
    elseif i = m + 1 then return q
    else return SELECT(q + 1, r, i - m - 1)
endif
endif.
```

The role of the median of medians is to prevent an unbalanced split of the array. We can thus safely use the deterministic version of splitting, but this really becomes clear only in the analysis of the algorithm.

**Analysis.** The items that are less than or equal to the median of medians are the first three items in the sets with medians less than or equal to the median of medians. In Figure 12, these items are highlighted by the box to the left and above but containing the median of medians. We are somewhat sloppy in the argument by assuming that their number is at least \( \frac{3n}{5} \). This is almost true except if \( n \) is not a multiple of 5, in which case it is true up to a small additive constant. Symmetrically, the number of items greater than or equal to the median of medians is at least \( \frac{3n}{10} \) (again up to an additive constant). The first recursion works on a set of \( \left\lceil \frac{n}{5} \right\rceil \) medians, and the second recursion works on a set of at most \( \frac{2n}{10} \) items. Ignoring additive constants, as before, we can ignore ceiling and floor functions and get

\[
T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + c \cdot n.
\]
We prove \( T(n) = O(n) \) by induction assuming \( T(m) \leq b \cdot m \) for \( m < n \) and \( b \) a large enough constant.

\[
T(n) \leq \frac{b}{5} \cdot n + \frac{7b}{10} \cdot n + c \cdot n
\]
\[
= \left( \frac{9b}{10} + c \right) \cdot n
\]
\[
\leq b \cdot n,
\]

provided \( b \geq 10c \). In words, we have proved that the worst-case running time of function \texttt{SELECT} is at most ten times the running time of function \texttt{SPLIT}, which is \( O(n) \). Hence the time for selecting the \( i \)-smallest item in an unsorted collection of \( n \) items is only \( O(n) \), even in the worst case.

A somewhat subtle issue is the presence of equal items in the input collection. Such occurrences make the function \texttt{SPLIT} unpredictable since they could occur on either side of the pivot. An easy way out of the dilemma is to make sure that the items that are equal to the pivot are treated as if they were smaller than the pivot if they occur in the first half of the array and they are treated as if they were larger than the pivot if they occur in the second half of the array.