3 Prune-and-Search

We use two algorithms for selection as examples for the prune-and-search paradigm. The problem is to find the i-smallest item in an unsorted collection of n items. 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 1-st or smallest item in the unsorted array $A[1..n]$.

```
min = 1;
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, which is optimal. Indeed, there is an adversary argument, that is, with fewer than $n-1$ comparisons we can change the minimum without changing the outcomes of the comparisons.

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. As in quicksort, we choose a random pivot and 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 $\ell = 1$ and $r = n$, limits included.

```
int RSELECT(int ℓ, r, i)
  q = RSPLIT(ℓ, r); m = q − ℓ + 1;
  if i < m then return RSELECT(ℓ, q − 1, i)
    elseif i = m then return q
    else return RSELECT(q + 1, r, i − m)
  endif.
```

For small sets, the algorithm is relatively ineffective and its running time can be improved by switching over to sorting when the size drops below some constant threshold. On the other hand, each recursive step makes some progress so that termination is guaranteed even without special treatment of small sets.

Expected running time. For each $1 \leq m \leq n$, the probability that the array is split into subarrays of sizes $m-1$ and $n-m$ is $\frac{1}{n}$. For convenience we assume that n is even. The expected running time increases with increasing number of items, $T(k) \leq T(m)$ if $k \leq m$. Hence,

$$\begin{aligned} T(n) &\leq n + \frac{1}{n}\sum_{m=1}^{n} \max\{T(m-1), T(n-m)\} \\ &\leq n + \frac{2}{n}\sum_{m=\frac{n}{2}+1}^{n} T(m-1). \end{aligned}$$

Assume inductively that $T(m) \leq cm$ for $m < n$ and a sufficiently large positive constant c. Such a constant c 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. We thus get

$$\begin{aligned} T(n) &\leq n + \frac{2c}{n}\sum_{m=\frac{n}{2}+1}^{n} m - 1 \\ &= n + c\cdot(n-1) - \frac{c}{2}\cdot\left(\frac{n}{2}+1\right) \\ &= n + \frac{3c}{4}\cdot n - \frac{3c}{2}. \end{aligned}$$

Assuming $c \geq 4$ we thus have $T(n) \leq cn$ as required. Note that we just proved that the expected running time of RSELECT is only a small constant times that of RSPLIT. More precisely, that constant factor is no larger than four.

Deterministic selection. The randomized selection algorithm takes time proportional to n^2 in the worst case, for example if each split is as unbalanced as possible. It is however possible to select in O(n) time even in the worst case. The *median* of the set plays a special role in this algorithm. It is defined as the i-smallest item where $i = \frac{n+1}{2}$ if n is odd and $i = \frac{n}{2}$ or $\frac{n+2}{2}$ if n is even. The deterministic algorithm takes five steps to select:

`Step 1.` Partition the n items into $\lceil \frac{n}{5} \rceil$ groups of size at most 5 each.

`Step 2.` Find the median in each group.

`Step 3.` Find the median of the medians recursively.

`Step 4.` Split the array using the median of the medians as the pivot.

`Step 5.` Recurse on one side of the pivot.

It is convenient to define $k = \lceil \frac{n}{5} \rceil$ and to partition such that each group consists of items that are multiples of k positions apart. This is what is shown in Figure 4 provided we arrange the items row by row in the array.

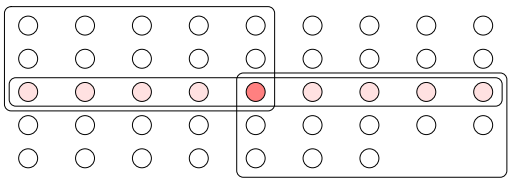

Figure 4: The 43 items are partitioned into seven groups of 5 and two groups of 4, all drawn vertically. The shaded items are the medians and the dark shaded item is the median of medians.

Implementation with insertion sort. We use insertion sort on each group to determine the medians. Specifically, we sort the items in positions $\ell, \ell+k, \ell+2k, \ell+3k, \ell+4k$ of array A, for each ℓ.

```
void ISort(int ℓ, k, n)
  j = ℓ + k;
  while j ≤ n do i = j;
    while i > ℓ and A[i] > A[i − k] do
      Swap(i, i − k); i = i − k
    endwhile;
    j = j + k
  endwhile.
```

Although insertion sort takes quadratic time in the worst case, it is very fast for small arrays, as in this application. We can now combine the various pieces and write the selection algorithm in pseudo-code. Starting with the code for the randomized algorithm, we first remove the randomization and second add code for Steps 1, 2, and 3. Recall that i is the rank of the desired item in $A[\ell..r]$. After sorting the groups, we have their medians arranged in the middle fifth of the array, $A[\ell+2k..\ell+3k-1]$, and we compute the median of the medians by recursive application of the function.

```
int Select(int ℓ, r, i)
  k = ⌈(r − ℓ + 1)/5⌉;
  for j = 0 to k − 1 do ISort(ℓ + j, k, r) endfor;
  m′ = Select(ℓ + 2k, ℓ + 3k − 1, ⌊(k + 1)/2⌋);
  Swap(ℓ, m′); q = Split(ℓ, r); m = q − ℓ + 1;
  if i < m then return Select(ℓ, q − 1, i)
    elseif i = m then return q
    else return Select(q + 1, r, i − m)
  endif.
```

Observe that the algorithm makes progress as long as there are at least three items in the set, but we need special treatment of the cases of one or of two items. The role of the median of medians is to prevent an unbalanced split of the array so we can safely use the deterministic version of splitting.

Worst-case running time. To simplify the analysis, we assume that n is a multiple of 5 and ignore ceiling and floor functions. We begin by arguing that the number of items less than or equal to the median of medians is at least $\frac{3n}{10}$. These are the first three items in the sets with medians less than or equal to the median of medians. In Figure 4, these items are highlighted by the box to the left and above but containing the median of medians. Symmetrically, the number of items greater than or equal to the median of medians is at least $\frac{3n}{10}$. The first recursion works on a set of $\frac{n}{5}$ medians, and the second recursion works on a set of at most $\frac{7n}{10}$ items. We have

$$T(n) \quad \leq \quad n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

We prove $T(n) = \mathrm{O}(n)$ by induction assuming $T(m) \leq c \cdot m$ for $m < n$ and c a large enough constant.

$$\begin{aligned} T(n) &\leq n + \frac{c}{5} \cdot n + \frac{7c}{10} \cdot n \\ &= \left(1 + \frac{9c}{10}\right) \cdot n. \end{aligned}$$

Assuming $c \geq 10$ we have $T(n) \leq cn$, as required. Again the running time is at most some constant times that of splitting the array. The constant is about two and a half times the one for the randomized selection algorithm.

A somewhat subtle issue is the presence of equal items in the input collection. Such occurrences make the function 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.

Summary. The idea of prune-and-search is very similar to divide-and-conquer, which is perhaps the reason why some textbooks make no distinction between the two. The characteristic feature of prune-and-search is that the recursion covers only a constant fraction of the input set. As we have seen in the analysis, this difference implies a better running time.

It is interesting to compare the randomized with the deterministic version of selection:

- the use of randomization leads to a simpler algorithm but it requires a source of randomness;
- upon repeating the algorithm for the same data, the deterministic version goes through the exact same steps while the randomized version does not;
- we analyze the worst-case running time of the deterministic version and the expected running time (for the worst-case input) of the randomized version.

All three differences are fairly universal and apply to other algorithms for which we have the choice between a deterministic and a randomized implementation.