1.6 Median Selection

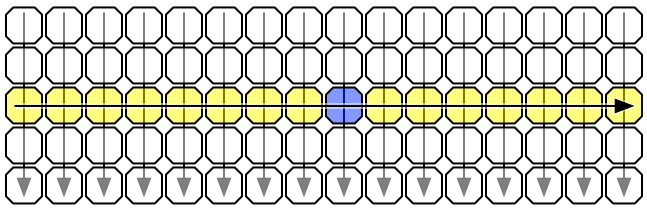
So how *do* we find the median element of an array in linear time? The following algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the kth largest element in an array, using the following recursive divide-and-conquer strategy. The subroutine PARTITION is the same as the one used in QUICKSORT.

```
SELECT(A[1..n], k):
  if n ≤ 25
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          B[i] ← SELECT(A[5i − 4..5i], 3)      ⟨⟨Brute force!⟩⟩
      mom ← SELECT(B[1..m], ⌊m/2⌋)             ⟨⟨Recursion!⟩⟩
      r ← PARTITION(A[1..n], mom)
      if k < r
          return SELECT(A[1..r − 1], k)        ⟨⟨Recursion!⟩⟩
      else if k > r
          return SELECT(A[r + 1..n], k − r)    ⟨⟨Recursion!⟩⟩
      else
          return mom
```

If the input array is too large to handle by brute force, we divide it into $\lceil n/5 \rceil$ blocks, each containing exactly 5 elements, except possibly the last. (If the last block isn't full, just throw in a few ∞s.) We find the median of each block by brute force and collect those medians into a new array. Then we recursively compute the median of the new array (the median of medians — hence 'mom') and use it to partition the input array. Finally, either we get lucky and the median-of-medians is the kth largest element of A, or we recursively search one of the two subarrays.

The key insight is that these two subarrays cannot be too large or too small. The median-of-medians is larger than $\lceil \lceil n/5 \rceil /2 \rceil - 1 \approx n/10$ medians, and each of those medians is larger than two other elements in its block. In other words, the median-of-medians is larger than at least $3n/10$ elements in the input array. Symmetrically, mom is smaller than at least $3n/10$ input elements. Thus, in the worst case, the final recursive call searches an array of size $7n/10$.

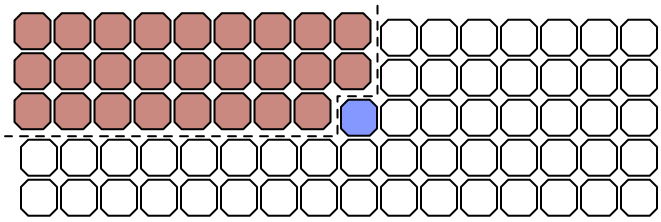
We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that the *algorithm* doesn't actually do this!) In this arrangement, the median-of-medians is the element closest to the center of the grid.



Visualizing the median of medians

The left half of the first three rows of the grid contains $3n/10$ elements, each of which is smaller than the median-of-medians. If the element we're looking for is larger than the median-of-medians,

our algorithm will throw away *everything* smaller than the median-of-median, including those $3n/10$ elements, before recursing. A symmetric argument applies when our target element is smaller than the median-of-medians.



Discarding approximately 3/10 of the array

We conclude that the worst-case running time of the algorithm obeys the following recurrence:

$$T(n) \leq O(n) + T(n/5) + T(7n/10).$$

The recursion tree method implies the solution $T(n) = O(n)$.

Finer analysis reveals that the hidden constants are quite large, even if we count only comparisons; this is not a practical algorithm for small inputs. (In particular, mergesort uses fewer comparisons in the worst case when $n < 4{,}000{,}000$.) Selecting the median of 5 elements requires at most 6 comparisons, so we need at most $6n/5$ comparisons to set up the recursive subproblem. We need another $n-1$ comparisons to partition the array after the recursive call returns. So the actual recurrence is

$$T(n) \leq 11n/5 + T(n/5) + T(7n/10).$$

The recursion tree method implies the upper bound

$$T(n) \leq \frac{11n}{5} \sum_{i \geq 0} \left(\frac{9}{10}\right)^i = \frac{11n}{5} \cdot 10 = 22n.$$

1.7 Multiplication

Adding two n-digit numbers takes $O(n)$ time by the standard iterative 'ripple-carry' algorithm, using a lookup table for each one-digit addition. Similarly, multiplying an n-digit number by a one-digit number takes $O(n)$ time, using essentially the same algorithm.

What about multiplying two n-digit numbers? At least in the United States, every grade school student (supposedly) learns to multiply by breaking the problem into n one-digit multiplications and n additions:

```
   31415962
 × 27182818
  251327696
   31415962
 251327696
 62831924
251327696
 31415962
219911734
62831924
---------
853974377340916
```