1.7 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 n-element array, given the array and the integer g as input, using a variant of an algorithm called either "quickselect" or "one-armed quicksort". The basic quickselect algorithm chooses a pivot element, partitions the array using the PARTITION subroutine from QUICKSORT, and then recursively searches only *one* of the two subarrays.

```
QUICKSELECT(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← PARTITION(A[1..n], p)
      if k < r
          return QUICKSELECT(A[1..r−1], k)
      else if k > r
          return QUICKSELECT(A[r+1..n], k−r)
      else
          return A[r]
```

The worst-case running time of QUICKSELECT obeys a recurrence similar to the quicksort recurrence. We don't know the value of r or which subarray we'll recursively search, so we'll just assume the worst.

$$T(n) \leq \max_{1 \leq r \leq n} (\max\{T(r-1), T(n-r)\} + O(n))$$

We can simplify the recurrence by using ℓ to denote the length of the recursive subproblem:

$$T(n) \leq \max_{0 \leq \ell \leq n-1} T(\ell) + O(n) \leq T(n-1) + O(n)$$

As with quicksort, we get the solution $T(n) = O(n^2)$ when $\ell = n-1$, which happens when the chosen pivot element is either the smallest element or largest element of the array.

On the other hand, we could avoid this quadratic behavior if we could somehow magically choose a *good* pivot, where $\ell \leq \alpha n$ for some constant $\alpha < 1$. In this case, the recurrence would simplify to

$$T(n) \leq T(\alpha n) + O(n).$$

This recurrence expands into a descending geometric series, which is dominated by its largest term, so $T(n) = O(n)$.

The Blum-Floyd-Pratt-Rivest-Tarjan algorithm chooses a good pivot for one-armed quicksort by *recursively computing the median* of a carefully-selected subset of the input array.

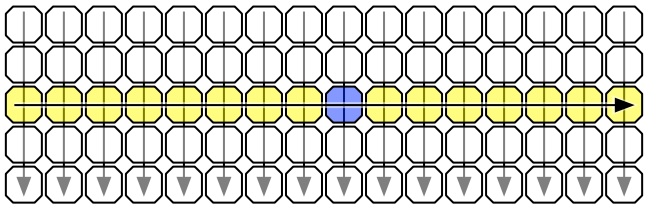
```
MOM5SELECT(A[1..n], k):
  if n ≤ 25
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MEDIANOFFIVE(A[5i−4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MOMSELECT(M[1..m], ⌊m/2⌋)        ⟨⟨Recursion!⟩⟩

      r ← PARTITION(A[1..n], mom)

      if k < r
          return MOMSELECT(A[1..r−1], k)     ⟨⟨Recursion!⟩⟩
      else if k > r
          return MOMSELECT(A[r+1..n], k−r)   ⟨⟨Recursion!⟩⟩
      else
          return mom
```

The recursive structure of the algorithm requires a slightly larger base case. There's absolutely nothing special about the constant 25 in the pseudocode; for theoretical purposes, any other constant like 42 or 666 or 8765309 would work just as well.

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 $M[1..\lceil n/5 \rceil]$. Then we recursively compute the median of this new array. Finally we use the median of medians — hence 'mom' — as the pivot in one-armed quicksort.

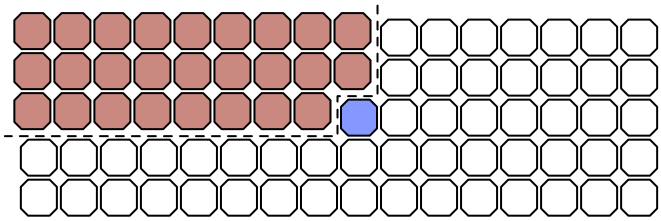
The key insight is that neither of these two subarrays can be too large. The median of medians is larger than $\lceil \lceil n/5 \rceil / 2 \rceil - 1 \approx n/10$ block medians, and each of those medians is larger than two other elements in its block. Thus, mom is larger than at least $3n/10$ elements in the input array, and 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 does not 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. Thus, the input to the recursive subproblem contains at most $7n/10$ elements. 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 constant hidden by the $O()$ is 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 a more accurate recurrence for the total number of comparisons is

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

The recursion tree method implies the upper bound

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

1.8 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? In most of the world, grade school students (supposedly) learn 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
```

We could easily formalize this algorithm as a pair of nested for-loops. The algorithm runs in $\Theta(n^2)$ time—altogether, there are $\Theta(n^2)$ digits in the partial products, and for each digit, we