4 Medians

The *median* of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. For instance, the median of $[45, 1, 10, 30, 25]$ is 25, since this is the middle element when the numbers are arranged in order. If the list has even length, there are two choices for what the middle element could be, in which case we pick the smaller of the two, say.

The purpose of the median is to summarize a set of numbers by a single, typical value. The *mean*, or average, is also very commonly used for this, but the median is in a sense more typical of the data: it is always one of the data values, unlike the mean, and it is less sensitive to outliers. For instance, the median of a list of a hundred 1's is (rightly) 1, as is the mean. However, if just one of these numbers gets accidentally corrupted to 10000, the mean shoots up above 100, while the median is unaffected.

Computing the median of n numbers is easy: just sort them. The drawback is that this takes $O(n \log n)$ time, whereas we would ideally like something linear. We have reason to be hopeful, because sorting is doing far more work than we really need – we just want the middle element, and don't care about the relative ordering of the rest of them.

When looking for a recursive solution, it is paradoxically often easier to work with a *more general* version of the problem — for the simple reason that this gives a more powerful step to recurse upon. In our case, the generalization we will consider is *selection*.

SELECTION

Input: A list of numbers S; an integer k

Output: The k^{th} smallest element of S

For instance, if $k = 1$, the minimum of S is sought, whereas if $k = \lfloor |S|/2 \rfloor$, it is the median.

A randomized divide-and-conquer algorithm for selection

Here's a divide-and-conquer approach to selection. For any number v, imagine splitting list S into three categories: elements smaller than v, those equal to v (there might be duplicates), and those greater than v. Call these S_L, S_v, and S_R respectively. For instance, if the array

$$S: \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \\ \hline \end{array}$$

is split on $v = 5$, the three subarrays generated are

$$S_L: \begin{array}{|c|c|c|}\hline 2 & 4 & 1 \\ \hline \end{array} \qquad S_v: \begin{array}{|c|c|}\hline 5 & 5 \\ \hline \end{array} \qquad S_R: \begin{array}{|c|c|c|c|c|c|}\hline 36 & 21 & 8 & 13 & 11 & 20 \\ \hline \end{array}$$

The search can instantly be narrowed down to one of these sublists. If we want, say, the *eighth*-smallest element of S, we know it must be the *third*-smallest element of S_R since $|S_L| + |S_v| = 5$. That is, $\text{SELECTION}(S, 8) = \text{SELECTION}(S_R, 3)$. More generally, by checking k against the sizes of the subarrays, we can quickly determine which of them holds the desired element:

$$\text{SELECTION}(S, k) = \begin{cases} \text{SELECTION}(S_L, k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{SELECTION}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

The three sublists S_L, S_v, S_R can be computed from S in linear time; in fact, this computation can even be done *in place*, that is, without allocating new memory (Exercise). We then recurse on the appropriate sublist. The effect of the split is thus to shrink the number of elements from $|S|$ to at most $\max\{|S_L|, |S_R|\}$.

Our divide-and-conquer algorithm for selection is now fully specified, except for the crucial detail of how to choose v. It should be picked quickly, and it should shrink the array substantially, the ideal situation being $|S_L|, |S_R| \approx \frac{1}{2}|S|$. If we could always guarantee this situation, we would get a running time of

$$T(n) = T(n/2) + O(n),$$

which is linear as desired. But this requires picking v to be the median, which is our ultimate goal! Instead, we follow a much simpler alternative: *we pick v randomly from S.*

Efficiency analysis

Naturally, the running time of our algorithm depends on the random choices of v. It is possible that due to persistent bad luck we keep picking v to be the largest element of the array (or the smallest element), and thereby shrink the array by only one element each time. In the example above, we might first pick $v = 36$, then $v = 21$, and so on. This *worst case* scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \cdots + n/2 = \Theta(n^2)$$

operations (when computing the median), but it is extremely unlikely to occur. Equally unlikely is the *best* possible case we discussed above, in which each randomly chosen v just happens to split the array perfectly in half, resulting in a running time of $O(n)$. Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the *average* running time lie? Fortunately, it lies very close to the best-case time.

To distinguish between lucky and unlucky choices of v, we will call v *good* if it lies anywhere in 25^{th} to 75^{th} percentile of the array that it is chosen from. We like these choices of v because they ensure that the sublists S_L, S_R have size at most $3/4$ that of S (do you see why?), so that the array shrinks substantially. Fortunately, good v's are abundant: half the elements of any list must fall between the 25^{th} to 75^{th} percentile!

Given that a randomly chosen v has a 50% chance of being good, how many v's do we need to pick on average before getting a good one? Here's a more familiar reformulation:

Lemma *On average a fair coin needs to be tossed two times before a "heads" is seen.*

Proof. Let E be the expected number of tosses before a heads is seen. We certainly need at least one toss, and if it's heads, we're done. If it's tails (which occurs with probability $1/2$), we need to repeat. Hence $E = 1 + \frac{1}{2}E$, which works out to $E = 2$. ▮

Therefore, after two split operations on average, the array will shrink to at most $3/4$ of its size. Letting $T(n)$ be the *expected* running time on an array of size n, we get

$$T(n) \leq T(3n/4) + O(n).$$

This follows by taking expected values of both sides of the statement

$$\begin{aligned}&\text{time taken on an array of size } n\\ &\quad \leq \ (\text{time taken on an array of size } 3n/4) + (\text{time to reduce array size to } \leq 3n/4),\end{aligned}$$

and, for the right-hand side, using the familiar property that *the expectation of the sum is the sum of the expectations* (it is not true of products. . .).

From this recurrence we conclude that $T(n) = O(n)$: On *any* input, our algorithm returns the correct answer after a linear number of steps, on the average.

The Unix `sort` command

Comparing the algorithms for sorting and median-finding we notice that, beyond the common divide-and-conquer philosophy and structure, they are exact opposites. Mergesort splits the array in two in the most convenient way (first half, second half), without any regard to the magnitudes of the elements in each half; but then it works hard to put the sorted subarrays together. In contrast, the median algorithm is careful about its splitting (smaller numbers first, then the larger ones), but its work ends with the recursive call.

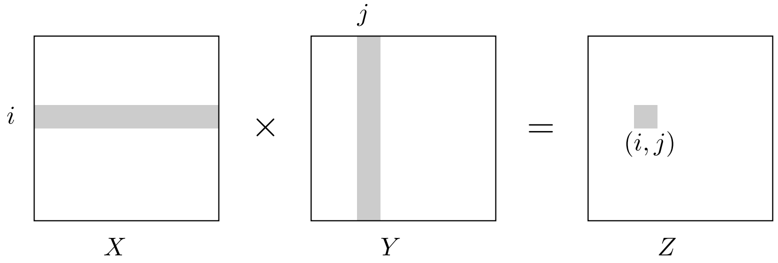
Quicksort is a sorting algorithm that splits the array in exactly the same way as our median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But it can be proved that its average case is $O(n \log n)$; furthermore, empirically it outperforms other sorting algorithms. This has made quicksort a favorite in many applications – for instance, it is the basis of the code by which really enormous files are sorted.

5 Matrix multiplication

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with $(i, j)^{th}$ entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

To make it more visual, Z_{ij} is the dot product of the i^{th} row of X with the j^{th} column of Y:



In general XY is not the same as YX; matrix multiplication is not commutative.

The formula above implies an $O(n^3)$ algorithm for matrix multiplication: there are n^2 entries to be computed, and each takes $O(n)$ time. For quite a while, this was widely believed