Median/Quickselect (Divide & Conquer Exemplar)

CSCI 382, Algorithms

October 12, 2020

Consider the following problem: given a (not necessarily sorted!) array of n integers, find the median element (that is, the element which would be in the middle if the array were sorted).

One obvious solution is to first sort the array in $\Theta(n \log n)$ time, and then simply access the array in the middle. However, intuitively it feels like this is doing too much work: we don't actually care about the order of any of the other elements, so sorting them is a waste of time. Can we do better than $\Theta(n \log n)$?

We consider a divide-and-conquer approach. First, if we simply split the list in half, it doesn't seem to help much. We could find the median of both sides but there's no way to compute the median of the whole list from the two medians. But simply splitting the list in half is not the only way to divide up a list. Since we were already thinking about sorting, what if we *partition* the list so that all the smaller elements are on one side and all the larger elements are on the other side? If we pick a random element as the *pivot*, we can partition the list into the elements $<$ the pivot on the left and $\geq$ the pivot on the right, as in quicksort, in $\Theta(n)$ time.

Now what? Simply finding the median of both sides still does not help. We can definitely say that the true median lies somewhere in between the left and right medians but that does not really help.

The solution—as often with recursive algorithms—is to *generalize* to compute something more than what is required. In particular, instead of just finding the *median* element, let's write an algorithm to select the element which would be at index k if the list were sorted. If we can do this, we can get the median by selecting the element at index $\lfloor n/2 \rfloor$. But this ability to select *any* element gives us the extra flexibility we need to use recursion to find the median, since the median of the entire list will not be the median of one of the two partitions, but some other index.

(Note, in practice, one would implement this so it all works in-place on the array A, instead of generating new arrays A_1 and A_2, *e.g.* by passing along extra *hi* and *lo* parameters to specify which part of the array to focus on.)

Theorem 1. *For any array A and any $0 \leq k < |A|$, QuickSelect(A, k) correctly returns the element with order index k, that is, the element which would be at index k in a sorted version of A.*

Proof. By induction on $|A|$.

```
Require: 0 ≤ k < |A|
 1: function QuickSelect(A, k)
 2:     if |A| = 1 then return A[0]
 3:     else
 4:         A_1, A_2 ← Partition(A)
 5:         if k < |A_1| then
 6:             return QuickSelect(A_1, k)
 7:         else
 8:             return QuickSelect(A_2, k − |A_1|)
 9: function Partition(A)
10:     Pick a random pivot value in A
11:     A_1 ← all values in A which are < pivot
12:     A_2 ← all values in A which are ≥ pivot
13:     return A_1, A_2
```

Figure 1: QuickSelect

- When $|A| = 1$, since $0 \leq k < |A| = 1$, we must have $k = 0$. The algorithm returns $A[0]$, which is indeed the item with index 0 in a sorted version of A, since a 1-element array is already sorted.
- Otherwise, we note that after the call to Partition, all the elements in A_1 are smaller than all the elements in A_2. Put another way, if we sorted A, all the elements in A_1 would come first, followed by all the elements in A_2. Thus, if $k < |A_1|$, then the element of A with order index k falls somewhere within A_1, and in fact is the element of A_1 with the same order index. On the other hand, if $k \geq |A_1|$, then the order-index k element of A falls somewhere inside A_2. If $k = |A_1|$ then we want the smallest element of A_2; if $k = |A_1| + 1$ then we want the second-smallest, and so on; in general, the kth smallest element of A will be the $k - |A_1|$ smallest element of A_2. Since the IH tells us that the recursive calls to QuickSelect will correctly select these elements from A_1 or A_2, we conclude that QuickSelect is correct.

□

Theorem 2. QuickSelect *runs in expected* $\Theta(n)$ *time.*

Proof. QuickSelect is a randomized algorithm, since the pivot value is chosen randomly; in theory we could make very bad pivot choices that would lead to bad performance. If the pivot is chosen randomly, however, we can expect that on average it will split the array into pieces which are $1/4$ and $3/4$ the size of the original array, respectively (or vice versa).

We can therefore analyze QuickSelect using the Master Theorem. We make one recursive call each time, so $a = 1$; in the worst case $b = 4/3$; and $d = 1$, since we spend $\Theta(n)$ time partitioning A. Therefore $a < b^d$ since $1 < 4/3$, so this is the first case of the Master Theorem, and we conclude that QuickSelect is $O(n^d) = O(n)$. We already know that the algorithm must be $\Omega(n)$, since there is no way to correctly find the kth element without looking at all the elements. Hence the algorithm runs in $\Theta(n)$ time. □