CS 218 : Design and Analysis of Algorithms

Lecture 8: Finding the Median

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering — Indian Institute of Technology, Bombay

1 Finding the Median

1.1 Problem Statement

The *rank* of an element in an array is the position of the element in the sorted array.

The *median* is an element of rank $\lceil n/2 \rceil$ in an n-element array. To make quick sort run in $n \log n$ time, one should be able to pick the median in linear time. You saw in your earlier course that if you pick the pivot uniformly at random then the expected time for quicksort $O(n \log n)$. The reason this happens is because at each step the random element you pick is an approximate median with good probability. [We have not speficied what approximate and good mean yet, but let it pass.]

In this section we describe an algorithm that finds the exact median in linear time.

We add that this is not how quicksort is implemented in practice. While the worst case running time of the algorithm for finding the median is linear, the algorithm is complicated and the constants involved are not very small. In practice one picks the pivot using a simple heuristic, like the first element.

An *approximate median* in an array is an element x such that, $\frac{1}{4} \leq \text{rank}(x)\frac{3}{4}$. Any one of these elements in the middle portion of the array qualifies as an approximate median.

Suppose there is a subroutine that finds an approximate median quickly (say in $O(n)$ time). Can one use this to find the exact median in $O(n)$ time? The answer is yes. Try it yourself before reading ahead. Note that the algorithm for the approximate median may return any of the candidate elements and you cannot access inside it.

Suppose the subroutine returned an approximate median x in the left half of the array. We can, using an additional n comparisons determine the exact position of x and partition the array based on x's position. We see that the median must lie on the right half of the partition. Ideally we would like to recurse on the right side. But there is a problem. The median on the right side is not the median of the array. So one cannot blindly apply recursion.

At least, let us convince ourselves that we should be thinking along these lines. If one spends $O(n)$ amount of work and decreases the size of the input by a constant fraction then the overall time is still $O(n)$. Prove this. That is if $T(n)$ satisfies $T(n) lecn + T(\alpha n)$, where c and $\alpha < 1$ are fixed constants and $T(2) = 1$, then $T(n)$ is $O(n)$.

Let us get back to the problem that one cannot blindly apply recursion. On the remaining portion of the array what we need is the element of rank $\lceil n/2 \rceil - rank(x)$. This is a new and more general problem that we have encountered. The more general problem is given an array A and a rank r, find an element of rank r in A. Note that if we solve this then finding the median is trivial. As usual we work on the more general problem. This surprising phenomenon we have encountered before. Solving a more general problem is easier than the problem itself. The reason is that the more general problem may be more ameanable to recursion. We have mentioned this as a design principle earlier. To recap we are given a procedure to find the approximate median which runs in linear time. Using that we will design an algorithm for the problem given below. **Input**: Array A, positive integer r. **Output**:An element of rank r in A. Here is a rough sketch

of the algorithm, assuming we can find the approximate median. Find-rank(A, r) Find x by calling Approximate Median(A). Partition A into A_1 consisting of elements less than x and A_2 consisting of elements greater than x. There are two cases depending on where x lies.
Case 1: r is smaller than rank(x). In this case we call Findrank(A_1, r)
Case 2 We call Findrank$(r - \text{rank}(x), A_2)$.

Let us restate **Design Principle 4** It may be easier to solve a more general problem. Build it into the recurrence.

And now add **Design Principle 5**. Reduction. Answer the following question: If we can solve problem B which seems simpler, can we solve problem A? In other words, reduce designing an algorithm for A to designing an algorithm for B.

To do this, we assume an efficient subroutine for B and design an efficient algorithm for A.

We have now reduced the problem of finding the median to the question: How do we find the approximate median?

This is non-trivial and we describe this below.

Partition the array into $n/5$ groups of 5 elements each.

We assume for simplicity that n is divisible by 5.

For each of these groups find the median. Let S denote the set of these medians. Note that $|S| = n/5$. Determine x, the (exact) median of the elements in S. In other words, we invoke Findrank$(\lceil n/10 \rceil, S)$ This element x is the approximate median we are looking for. This completes the description of the algorithm.

Many questions crop up. Let us deal with them one by one. We defined Findrank in terms of Approximatemedian and Approximatemedian in terms of Findrank. Are we not moving in circles. Have we achieved anything? We have! The progress we have made comes from the fact that in approximatemedian we call Findrank on inputs of size at most $3n/4$. Recall our first design principle. How about in Findrank?

We show next that x will indeed be an approximate median. Just to recap. We are done with the algorithm description. We now begin a proof of correctness and analysis of the running times. To see that it is correct in terms of running times, we must show that approximatemedian does indeed return the approximate median. To see this, arrange the array A as a $5 \times n/5$ matrix. The middle row consists of the elements of S in increasing order. The ith column consists of the elements of the ith partition placed in increasing order from top to bottom.

To see that $S_{\lceil n/10 \rceil}$ is an approximate median it suffices to observe that the top left quarter contains elements which are all less than $S_{\lceil n/10 \rceil}$ and the bottom-right quarter contains elements all of whom are greater that $S_{\lceil n/10 \rceil}$. Make sure you see why!

Now for the time analysis. Let $F(n)$ denote the time taken by median on inputs of size n. Find rank and $A(m)$ denote the time taken by approximate median on inputs of size m.

Then $F(n) \le A(n) + F(3n/4) + c_1 n$. $A(n) \le c_2 n + F(n/5)$.

Hence, $F(n) \le F(3n/4) + F(n/5) + cn$.

Now, check by induction that $F(n) \le 20cn$.

How did we get this 20? This is by reverse engineering. Somewhat like cooking up the observations in your physics experiment knowing the exact value of the acceleration of gravity. We expect the time to be dn. This is because of the following. We take cn time and then are left with two sub problems of size $3n/4$ and $n/5$. The combined size is $19n/20$. So we suspect that $F(n) \le d_n$. To find d, we mimic an inductive proof, $T(n) \le d(3n/4) + d(n/5) + cn$. That is $T(n) \le nd(19/20 + c/d)$. For the induction to go through we want this to be at most nd. We leave the rest of the the calculations to you.

How does one come up with such an algorithm? Ingenuity and lots of hard work. The technique can be given a name: bootstrapping and parametrization. Since these are beyond the present course, we will mention and leave them be.