Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 10

Lecture 50

Welcome to the 50th lecture of the second-level algorithm course. In the last couple of lectures, we have been studying order statistics. In this lecture, we will study the general order statistics problem, OK? So, let us begin. So, let us recall the general order statistics problem: given an integer array of size n and an integer k, find the kth smallest integer of the array A. Again, let us see what the naive algorithm for this So, a naive algorithm could be: it first finds the minimum of the array A with $n-1$ comparisons, then it finds the second minimum which is the minimum of the remaining $n-1$ elements with $n-2$ comparisons, and then finds the third minimum with $n-3$ comparisons and continues. like this for k iterations. The total number of comparisons performed by this algorithm is $n-1$ in the first iteration, $n-2$ in the second and $n-k$ in the k-th iteration.

Which is $nk-\frac{k\times(k+1)}{2}$ which is theta n square for $k=\Theta(n)$. So, for example, if k equal to say $n/2$ or $n/5$ or $n/1000$ then the number of comparisons performed by this algorithm is theta n square. So, this is the naive algorithm and we will now see how we can get a better algorithm. So, with the goal of designing a better algorithm let us first try out a different approach and this approach is motivated by. the quick sort algorithm.

So, let us call it min which takes the array A of size n as input and an integer k and it assumes that k is in between 1 and n. So, the base case if n equal to 1 then k must be 1 and then the k-th minimum must be a 1. In this case the algorithm returns the first element of the array otherwise it picks a pivot of the array element an element of the array as pivot any element of the array can be picked as a pivot.

So, let us pick a 1 then this array is partitioned with respect to this pivot. So, this partition subroutine is the same partition subroutine used in quick sort algorithm which we will assume we know. This partition subroutine takes the array A as input and the pivot and it partitions the array into 2 parts. on the left side of pivot all elements are less than equal to the pivot on the right side of the pivot are all elements greater than equal to the pivot and this partition subroutine returns the index of the pivot. That means, after the function returns a l is the pivot.

Now if l equal to k then we know that the pivot is the k-th smallest element in this case the algorithm returns the pivot. Otherwise if l is greater than k then the k-th smallest element is present in the left side of the array and we restrict our search for the k-th smallest element only in the left side. So, the algorithm returns it calls itself recursively with the left part of the array up to $l-1$ and in that part we are looking for the k-th minimum element.

Otherwise l is less than k and then we need to search for the kth minimum element on the right hand side. So, the algorithm calls itself recursively with the right side of the array which are $l+1$ to n and in that part we are looking for the k minus lth minimum element of the array. The k minus lth minimum element of $a[l+1,\ldots,n]$ is the kth minimum element of the entire array $a[1,\ldots,n]$. So, this is the case and this is the So, as you can see this is very similar to quick sort algorithm.

In quick sort algorithm because we have to sort the entire array we recursively called the quick sort subroutine on both the parts, but here we are doing searching for the k-th smallest element. Hence, we can restrict our search to the suitable part of the array. So, let us see what is the runtime of the algorithm. So, $T(n)$ again for the study of order statistics we will focus on the number of comparisons as our performance metric instead of runtime. So, $T(n)$ is the maximum number of comparisons that the above algorithm performs. On any input array of size at most n, ok. So, T(n), because it is a recursive function, is 0 if $n=1$; otherwise, we are making a call to partition, which takes a linear amount of time. So, assume that the algorithm makes at most C*n comparisons on any array of size at most n. So, then T(n) is less than or equal to—because we want to upper-bound it—we are making a call to the partition subroutine, which makes at most C*n comparisons, plus we are either calling the function recursively on the left side or on the right side, and we do not know l. So, in the worst case, l could be 1 or 2, or in the worst case, the subarray where we are performing the recursive call could be as large as n. So, then this becomes $C*n+T(n-1)$. Otherwise. So, then expanding this for arbitrary n, T(n) is less than or equal to $C*n+T(n-1)$, and if we solve it, we get less than or equal to $C*n+C\times(n-1)$ up to C, which is $\frac{C\times n\times(n+1)}{2}$, and C is a constant. So, this is Theta(n squared), ok. So, it seems that in the worst case, we do not have any improvement, but here is some good news: it can be proved that the average case—average over all possible inputs—average case number of comparisons performed by the algorithm is theta n log n which is an improvement. Moreover if we study why the worst case number of comparisons is theta n square it is because the pivot may be selected. So, that it is the partitioning is skewed one part of the array contains substantially many elements compared to the other So, if we can choose a pivot not in too much time in $O(n)$ time. such that there exists a constant let us call it d in between 0 and with the guarantee that l is at least dn and at most $1-d$ where l is the index of the pivot element after the partition subroutine, then the worst case number of comparisons made by the algorithm is $O(n \log n)$, how?

So, in this case we have the following recurrence relation for $T(n)$. $T(n)$ is as usual 0 if n equal to 1, otherwise we are making a call to partition subroutine which makes at most C n comparisons plus if d is greater than or if d is less than or if d is greater than half then the size of the maximum subarray is at most d n. So, it will be $T(dn)$. on the other hand it will be at most $1-dn$ actually if you look at here d should be less than. So, this inequality implies that d should be less than equal to half.

The third case does not appear. So, now, if we solve the recurrence with d less than 1 and greater than or equal to half. So, this part also we can get rid of, then we have T(n) less than or equal to $cn+cdn+cd^2n$, and you can make this sum, ah, an infinite sum upper bound by infinite sum. So, this you take $cn(1+d+d^2+\ldots)$, this infinite sum, and this is c $n\frac{1}{1-d}$. Now, because d is constant. So, this is, we can see $O(n)$. So, the number of comparisons will be not even $O(n \log n)$; it will be $O(n)$. So, in the next lecture, we will see a linear-time algorithm to compute a pivot with this guarantee, thereby actually even better guarantee. We can even find a median of an integer array A of size n with a linear number of comparisons, and once we get that, we can use that to solve the order statistics problem, ok. So, let us stop here.

Thank you.