DD2440 ADVANCED ALGORITHMS
LECTURE 3: LINEAR-TIME MEDIAN-FINDING ALGORITHMS

Notes by Didrik Lundberg (lecture by Johan Håstad).

1. Quickselect

How do we efficiently find the median in an array of numbers? This problem, and in general the problem of finding the kth smallest element in an array S of size $|S| = n$, is solved by a class of algorithms called selection algorithms. Depending on k and the size of S, the kth smallest element can be the minimum, maximum or median element. We will however only discuss finding the median element in this lecture.

Quickselect(S, k) is an interesting selection algorithm which is related to the Quicksort algorithm. Similar to how Quicksort works, in Quickselect set S is divided into two partitions by cleverly choosing a pivot element. Unlike Quicksort however, we can discard one of the partitions, the one we know will not contain the median, instead of passing a recursive call to both.

Quickselect loosely works like this: Let x be a random element in S. Then we can define the partitions $S_0 = \{y \in S, y < x\}$ and $S_1 = \{y \in S, y \geq x\}$. If the array S is already sorted, we can partition it into x, S_0 and S_1 thusly:

$$\underbrace{\cdots\cdots\cdots}_{S_0}, x, \underbrace{\cdots\cdots\cdots}_{S_1}$$

If the array S is not already sorted, we can perform this partition simply by doing $n-1$ comparisons, where we check (for every element except x) if it belongs to S_0 or S_1, by using the definitions of these sets. Depending on our choice of x and the resulting partition, we end up in one of three situations:

(1) If $|S_0| \geq k$: call Quickselect(S_0, k);
(2) If $|S_0| = k - 1$: return x;
(3) If $|S_0| < k - 1$: call Quickselect$(S_1, k - (|S_0| + 1))$.

As mentioned earlier, Quicksort has two recursive calls (one for each partition), but in this case we only need to make one recursive call. One of the partitions will not be interesting to us, since we can be entirely sure that it does not contain the median.

Theorem 1.1. *The expected number of comparisons for* Quickselect *is less than or equal to* $4|S|$.

Proof. We need $|S| - 1$ comparisons to find S_0 and S_1, and on top of this we have our recursive call. Let us calculate the expected size of this recursive call. We have three cases:

(1) If $|S_0| = 0, ..., k - 2$: recursive call size is $n - |S_0| + 1$;
(2) If $|S_0| = k - 1$: no recursive call is made;

Date: October 6, 2014.

(3) If $|S_0| \geq k$: recursive call size is $|S_0|$.

This gives us the expression

$$\frac{1}{n}\sum_{i=0}^{k-2} n-(i+1)+\frac{1}{n}\sum_{i=k+1}^{n} i$$

for the expected size of the recursive call. Let us now recall that the sum of an arithmetic sequence is equal to number of terms $\times$ average term, where average term $= \frac{\text{first term+last term}}{2}$. The expression for our arithmetic series (disregarding for the moment the factor $\frac{1}{n}$) then becomes

$$\begin{aligned}(k-1)\frac{n+1-k+n-1}{2}+(n-(k-1))\frac{k+n-1}{2}&\\ =(k-1)\frac{2n-k}{2}+(n-(k+1))(\frac{k+n-1}{2})&\\ \leq \frac{2nk^2}{2}+\frac{(n-k)(n+k)}{2}&\\ =\frac{n^2+2nk-2k^2}{2},&\end{aligned}$$

which in the worst case is equal to

$$\frac{3n^2-(n-2k)^2}{4} \leq \frac{3n^2}{4}.$$

However, we also have a factor $\frac{1}{n}$, giving us the expected size of recursive call $\frac{1}{n}\frac{3n^2}{4} = \frac{3n}{4}$. The total cost is then

$$\underbrace{|S|-1}_{n-1}+\underbrace{4\cdot\frac{3n}{4}}_{\text{by induction}} = n+3n-1 \leq 4n.$$

□

2. More efficient algorithms

Can we achieve a better constant than 4? Here are two examples of ideas for creating other algorithms that have better efficiency:

(1) Choose x to be the median of $n^{\frac{3}{4}}$ elements (x can be found in $O(n^{\frac{3}{4}})$ comparisons). Then, S_0 and S_1 will be very close to $\frac{n}{2}$ in size (shown below). We end up with a recursive call of size $\frac{n}{2}+\mathcal{O}(n)$.

$$\underbrace{\cdots\cdots\cdots}_{\approx\frac{n}{2}}, x, \underbrace{\cdots\cdots\cdots}_{\approx\frac{n}{2}}$$

(2) Play a "cup" to find the winner (the smallest or largest element), as shown in Figure 1. Whether we compete for being the smallest or largest element, the next "best" element will be found among the $\log(n)$ elements which lost to the winner. We can then play a cup among these to find out who is the next best, meaning $n-1+\log(n)$ comparisons in total to find next best. A similar procedure is taken for the third best, and so on.

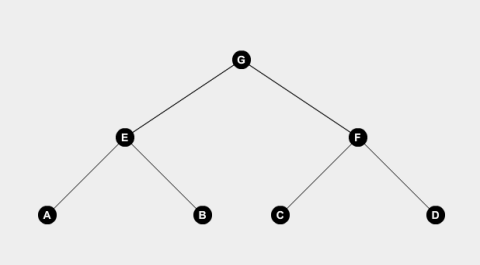

FIGURE 1. "Cup" comparisons between elements.

3. The most efficient algorithms

Theorem 3.1. *There is a provably optimal randomized algorithm that finds median in* $\frac{3n}{2} + \mathcal{O}(n)$ *comparisons.*

Proof. One may want to consult the original paper by Floyd and Rivest [2]. □

Is there a deterministic algorithm which runs in $\mathcal{O}(n)$? If we want to look for such an algorithm, a good start would be looking for a good pivot x. To make the presentation simpler suppose $|S| = n = 5m$ (that the size of S is divisible by five - five is the smallest integer for which the following scheme works). Then divide S into m groups. Compute the m medians in these groups. Let x be the median of these m medians.

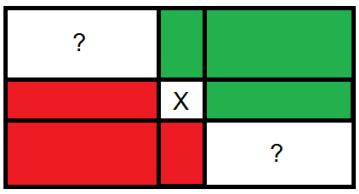

FIGURE 2. Ordering of elements by division into m columns of size 5. Only the central column and the central row are shown explicitly.

So, for each column we know that the upper elements are larger than the middle element, and that the lower elements are smaller. For the middle row of the m medians, we know that elements to the right of x are larger and elements to the left are smaller. If we were to compare all the elements to x, we would have one area in which all elements are smaller than x (the red area), and one area in which all elements are larger than x (the green area). In addition to this, there are two areas of which we do not know anything about in this regard.

The number of elements which we know to be larger than x is then $\frac{3n}{10}$, as is the number of elements which we know to be smaller than x. The areas whose relation to x is unknown both have a size of $\frac{2n}{10}$.

Now, we utilize x as a pivot. The partition into S_0 and S_1 is done in the same fashion as in Quickselect. We will need to do comparisons in all the unknown parts, but we have a favourable situation where there are a lot of elements we have already related to x. Then finally we make a recursive call on our remaining set of interest, where we know the median can be found. The size of this set of elements will be less than or equal to $\frac{7n}{10}$. The algorithm can be described in short in four steps:

(1) Find median in each group of 5 (takes $\frac{6n}{5}$ comparisons),
(2) Set x to be the median of these medians (takes $T(\frac{n}{5})$ comparisons),
(3) Partition S_0 and S_1 in the same fashion as for Quickselect (takes $\frac{2n}{5}$ comparisons),
(4) Do recursive call on set of interest (takes $\leq T(\frac{7n}{10})$ comparisons).

Since we are interested in calculating the efficiency of the algorithm, we are looking for a function $T(n)$ that solves

$$\frac{6n}{5} + T(\frac{n}{5}) + \frac{2n}{5} + T(\frac{7n}{10}) \leq T(n)\,,$$

with the boundary condition $T(1) = 0$ (finding the median in a list of 1 element requires zero comparisons). We set $T(n) = c \cdot n$ on the left-hand side of the above equation and get

$$\begin{aligned} \frac{8n}{5} + \frac{9nc}{10} &\leq nc\,, \\ \frac{8n}{5} &\leq \frac{nc}{10}\,, \\ 16 &\leq c\,. \end{aligned}$$

Now we can compare the above algorithm with the efficiency of other algorithms. What is the smallest c that any deterministic algorithm has achieved? In 1995, Zwick and Dor achieved $c \approx 2.95$ with the "green spider factory" selection algorithm [3]. This is the best upper bound a deterministic algorithm has achieved at this point.

Let us have a look at the partially ordered sets in Figure 1. Each vertex is one element, and each edge means that a comparison has been made between the two elements, where the highest ordered element is placed above the other.
Where and how can we tell what the median is?

In the leftmost graph, we see that vertex D is the median, since three elements are larger than D, and three elements are smaller, by the transitive property of the partial order. The rightmost graph does not have a clearly identifiable median, however.

Theorem 3.2. *If the median is known, then we must have made $n-1$ comparisons.*

Proof. The theorem is intuitively very clear; even if you were given the correct median by a psychic medium you would need $n-1$ comparisons to verify it! We need to know how many elements are bigger than, and smaller than, the median. For each element, we have one comparison that shows this. □

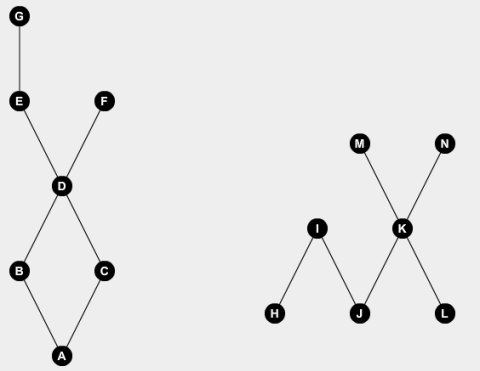

FIGURE 3. Two partially ordered sets.

So $n-1$ comparisons is a theoretical lower bound, in the sense that it would be logically impossible to be more efficient. This is only a start to calculating an actual, practical lower bound, which is far from trivial to do. In 1985, BENT and JOHN gave a $2n + o(n)$ lower bound for median-finding algorithms [1], refined to $2n + \epsilon$ by ZWICK and DOR (again) in 1996 [4], which holds today. Note that this lower bound is smaller than 2.95.

REFERENCES

[1] Samuel W. Bent and John W. John, Finding the median requires 2n comparisons, in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 213-216. 1985.

[2] Robert Floyd and Ronald Rivest, Expected Time Bounds for Selection, in *Communication of the ACM*, pages 165-172, 1975.

[3] Uri Zwick and Dorit Dor, Selecting the median, in *Proceedings of 6th SODA*, pages 88-97. 1995.

[4] Uri Zwick and Dorit Dor, Median selection requires $(2+\epsilon)n$ comparisons, in *Proceedings of 37th FOCS*, pages 125-134. 1996.