FIT2004: Supplementary notes for lecture 3

Order Statistics and Selection Algorithms

DANIEL ANDERSON [1]

Arising in many applications is the need to find the minimum, maximum or median element of a sequence. These are all special cases of a general problem called order statistics. We will consider several different cases of this problem and explore some algorithms for solving them.

Summary: Order Statistics and Selection Algorithms

In this lecture, we cover:

- What order statistics are
- Finding minimum and maximums
- The Quickselect algorithm
- The median of medians technique (NOT EXAMINABLE)

Recommended Resources: Order Statistics and Selection Algorithms

- CLRS, Introduction to Algorithms, Chapter 9
- Weiss, Data Structures and Algorithm Analysis, Section 10.2.3

Order Statistics and the Selection Problem

The k'th order statistic of a sequence of N elements is defined to be the k'th smallest element. For example, the first order statistic is the simply the smallest element of the sequence and the n'th order statistic is simply the maximum. The median element of a sequence is an element that is smaller than and larger than one half of the other elements of the sequence. In terms of order statistics, the median is the $k = (N+1)/2$ order statistic, rounding either up or down if N is even. The selection problem is the problem of finding for a given sequence and a given value of k, the k'th order statistic, ie. the k'th smallest element of the sequence. The easy way to solve the selection problem is of course to simply sort the sequence and then select the k'th element that results. Using a fast ($O(N \log(N))$) sorting algorithm would yield an $O(N \log(N))$ solution to the selection problem. In this section, we aim to explore faster algorithms for solving the selection problem.

Finding Minimums and Maximums

Minimums and maximums are the simplest order statistics to find. Rather than sorting the entire sequence in $O(N \log(N))$, an obvious linear time algorithm to find the minimum and/or maximum is to simply iterate over the sequence and track the best found minimum and/or maximum so far.

Algorithm: Select Minimum

```
1: function SELECT_MIN(array[1..N])
2:     Set min = array[1]
3:     for i = 2 to N do
4:         if array[i] < min then
5:             min = array[i]
6:         end if
7:     end for
8:     return min
9: end function
```

[1] Faculty of Information Technology, Monash University: daniel.anderson@monash.edu. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

That was rather easy, but it raises an interesting question: can we solve this problem any faster or with fewer comparisons in the general case (assuming random input?) It should not be hard to convince yourself that we can not determine the minimum with any fewer than $N-1$ comparisons, since performing less than $N-1$ comparisons leaves at least one element of the array that has never been compared to any other, which could potentially be the minimum.

A more interesting problem arises if we consider the case of finding both the minimum and maximum element at the same time. Clearly, our first algorithm for finding the minimum in $N-1$ comparisons could easily be adapted to find both the minimum and maximum in $2N-2$ comparisons, but can we do better? The answer is no longer obvious since we are doing more comparisons than necessary to find either one of the minimum or maximum on its own. It turns out that we can in fact select both the minimum and maximum element of a sequence in fewer, just $\sim 1.5N$ comparisons.

The trick is to just consider each pair of numbers in the input, and for each pair, to only compare the higher of the two with the current maximum, and the lower of the two with the current minimum. This results in cutting out $\frac{1}{4}$ of the comparisons which were unnecessary. Assuming that N is even, the following implementation demonstrates this trick.

Algorithm: Select Minimum and Maximum

```
1: function SELECT_MINMAX(array[1..N])
2:     Set min = array[1], max = array[1]
3:     for i = 1 to N, step 2 do
4:         if array[i] < array[i+1] then
5:             if array[i] < min then
6:                 min = array[i]
7:             end if
8:             if array[i+1] > max then
9:                 max = array[i+1]
10:            end if
11:        else
12:            if array[i] > max then
13:                max = array[i]
14:            end if
15:            if array[i+1] < min then
16:                min = array[i+1]
17:            end if
18:        end if
19:    end for
20:    return min, max
21: end function
```

If N is not even, it suffices to simply start the outer loop at `i = 2` to correct the algorithm.

Interestingly, it is possible to prove that $1.5N$ is the smallest possible number of comparisons that are required in the worst case to select the minimum and maximum element of a sequence, so this algorithm is in fact performing the optimal number of comparisons.

The Quickselect algorithm

Minimums and maximums are easy to find. Of much more theoretical interest is the problem of finding the median, or indeed the general k'th order statistic. Remarkably, although minimums and maximums are "easy" problems that require at least $O(N)$ time to solve, the general k'th order statistic problem can also be solved in worst case $O(N)$, so the "hard case" and "easy case" are actually asymptotically the same difficulty!

The Quickselect algorithm, if the name was not a massive hint already, is based on the Quicksort algorithm. Just like Quicksort, we select a pivot element, partition the array about the pivot and then repeat recursively. The major difference between Quicksort and Quickselect is that while Quicksort recursively sorts both halves of the partition, Quickselect only needs to sort the half of the partition that contains the k'th element, since contents of the other half does not matter. As was the case for Quicksort, pivot selection is the crucial step

of the algorithm that determines whether or not the run time will be fast or slow. Using the same `partition` function from Quicksort, a simple realisation of Quickselect can be implemented like so.

Algorithm: Quickselect

```
1: function QUICKSELECT(array[lo..hi], k)
2:     if hi > lo then
3:         Set pivot = array[lo]
4:         left, right = partition(array[lo..hi], pivot)
5:         if k < left then
6:             return quickselect(array[lo..left-1], k)
7:         else if k ≥ right then
8:             return quickselect(array[right..hi], k)
9:         else
10:            return array[k]
11:        end if
12:    else
13:        return array[k]
14:    end if
15: end function
```

Just like Quicksort though, this function has a worst-case run time of $O(N^2)$ since we might select the minimum or maximum element as the pivot and hence perform $O(N)$ levels of recursion. Luckily for us, there are strategies to avoid this sort of pathological behaviour. We will explore two such strategies: random pivot selection which results in an expected $O(N)$ run time and is empirically the fastest strategy, and the median of medians of fives technique, which yields a guaranteed worst-case linear performance, although is not as efficient in practice.

Randomised pivot selection

One of the simplest and empirically the fastest solution to avoid pathological pivot choices is to simply select the pivot randomly.

Theorem: Randomised Pivot Selection

Using randomised pivot selection, the Quickselect algorithm has expected run time $O(N)$.

Proof

Assume without loss of generality that the array contains no duplicate elements and denote by T_N the time taken by Quickselect to select the k'th order statistic of an array of size N. If the pivot selected is the i'th order statistic of the array, then the time taken by Quickselect will be

$$T_{N,i} = N + 1 + \begin{cases} T_{N-i} & \text{if } i < k, \\ 0 & \text{if } i = k, \\ T_{i-1} & \text{if } i > k \end{cases}.$$

Averaging out over all possible pivot selections, we obtain an expected run time of

$$T_N = N + 1 + \frac{1}{N}\left(\sum_{i=1}^{k-1} T_{N-i} + \sum_{i=k+1}^{N} T_{i-1}1\right).$$

Rearranging the indices of the sums, we find that this is the same as

$$T_N = N + 1 + \frac{1}{N}\left(\sum_{i=N-k+1}^{N-1} T_i + \sum_{i=k}^{N-1} T_i\right).$$

Since both of the sums appearing above in the bracketed expression contain a consecutive sequence of terms ending at T_{N-1} and there are a total of N terms, they must be bounded above by the largest

possible terms, so we have the inequality

$$T_N \leq N + 1 + \frac{2}{N} \sum_{i=N/2}^{N-1} T_i.$$

We can now prove by induction that $T_N \leq CN = O(N)$ for some constant C. First, we have that $T_0 = T_1 = 1 \leq CN$ for any choice of $C \geq 1$. Now suppose that $T_i \leq Ci$ for some constant C for all $i < N$. We have that

$$T_N \leq N + 1 + \frac{2}{N} \sum_{i=N/2}^{N-1} T_i,$$

which by our inductive hypothesis satisfies

$$T_N \leq N + 1 + \frac{2}{N} \sum_{i=N/2}^{N-1} Ci.$$

Evaluating the sum using the fact that $\sum_{i=0}^{N} i = \frac{N(N+1)}{2}$, we find

$$\begin{aligned} T_N &\leq N + 1 + \frac{2C}{N} \left(\frac{N(N-1)}{2} - \frac{\frac{N}{2}(\frac{N}{2} - 1)}{2} \right) \\ &\leq N + 1 + C(N-1) - \frac{C}{2}(\frac{N}{2} - 1) \\ &\leq \left(\frac{3}{4}C + 1 \right) N + 1 - \frac{C}{2} \end{aligned}$$

If we choose $C = 4$ then we will have

$$T_N \leq 4N - 1 \leq 4N = CN$$

as desired. Hence by induction on N, we have that $T_N = O(N)$, completing the proof.

The median of medians technique (NOT EXAMINABLE)

The randomised pivot selection technique yields expected linear time complexity and has excellent empirical performance. From a theoretical standpoint however, it is not worst-case linear as the run time is still $O(N^2)$ if pathological pivots are selected at every single step. Despite the absolute unlikelihood of this happening, it is theoretically satisfying to come up with an algorithm that is absolutely linear in the worst-case. The fact that there exists guaranteed linear time algorithms for arbitrary order statistics is as mentioned earlier, theoretically pleasing.

One such linear time selection algorithm is the median of medians algorithm, which is simply a special pivot selection strategy for Quickselect. We know that if Quickselect could select the median as the pivot, then it would yield guaranteed linear run time, but of course, finding the median is precisely the problem (or one specific case of the problem) that we are trying to solve. Instead, the median of medians technique involves finding an approximate median that is good enough to guarantee linear time performance.

Key Ideas: Median of medians

The median of medians algorithm selects an approximate median by:

- Dividing the original list of N elements into $\lceil \frac{N}{5} \rceil$ groups of size at most 5.
- Finding the median of each of these groups (use insertion sort since they are small sequences)
- Finding the true median of the $\lceil \frac{N}{5} \rceil$ medians using Quickselect[a]

[a] This makes Quickselect and the approximate median finder mutually recursive functions.

Using these ideas, an implementation of the pivot selection strategy might look like this.

Algorithm: Median of Medians Pivot Selection

```
1:  function SELECT_PIVOT(array[lo..hi])
2:      if hi - lo < 5 then
3:          return median_of_five(array[lo..hi])
4:      else
5:          Set medians = []
6:          for i = lo to hi, step 5 do
7:              Set j = min(i + 4, hi)
8:              Set median = median_of_five(array[i..j])
9:              medians.append(median)
10:         end for
11:         Set N = medians.size()
12:         return quickselect(medians[1..N], ⌊(N+1)/2⌋)
13:     end if
14: end function
```

Where the function `median_of_five` selects the true median of a small list using insertion sort.

Algorithm: Median of Five

```
1: function MEDIAN_OF_FIVE(array[lo..hi])
2:     insertion_sort(array[lo..hi])
3:     return array[(lo+hi)/2]
4: end function
```

It remains for us to prove that this technique does indeed result in guaranteed linear time performance. On the one hand, we must ensure that the pivot selected is good enough to yield low recursion depth, and on the other, we need to be convinced that finding the approximate median does not take too long and kill the performance of the algorithm.

Theorem: Worst-case performance of the median of medians algorithm

Quickselect using the median of medians strategy has worst-case $O(N)$ performance.

Proof

First, we note that out of the $\frac{N}{5}$ groups, half of them must have their median less than the pivot (by definition of the median). Similarly, half of the $\frac{N}{5}$ groups must have their medians greater than the pivot. Together, we have $\frac{N}{10}$ groups whose median is less than the pivot, and $\frac{N}{10}$ groups whose median is greater than the pivot.

Since within each group of five, two elements are greater than its median and two elements are less than its median, we have transitively that in each of the $\frac{N}{10}$ groups whose median is less than the pivot, three elements that are less than the pivot, and similarly in each of the $\frac{N}{10}$ groups whose median is greater than the pivot, three elements that are greater than the pivot. This yields a grand total of $\frac{3N}{10}$ elements that are below and above the pivot respectively.

Consequently, the median of medians lies in the 30th to 70th percentile of the data. From this, we know that the worst case of the recursive call made by Quickselect will be on a sequence of size $0.7N$. In addition to the recursive call, we also make an additional recursive call of size $0.2N$ to compute the exact median of the original $\frac{N}{5}$ medians. Adding in the linear number operations required to perform the partitioning step and to find the median of each group of 5, a recurrence relation for the amount of work done by Quickselect using the median of medians strategy is

$$T_N = aN + T_{0.2N} + T_{0.7N}, \qquad T_0 = 0,$$

for some constant a. We can show that this recurrence relation has solution $T_N \leq CN$ for some constant

C by induction. First let C be a constant such that $C \geq 10a$. Consider the base case

$$T_0 = 0 <= C \times 0 = 0,$$

which trivially holds. Now suppose that $T_n \leq Cn$ for $n < N$, then we have, by substituting this inductive hypothesis

$$\begin{aligned} T_N &= aN + T_{0.2N} + T_{0.7N} \\ &\leq aN + 0.2CN + 0.7CN. \\ &= aN + 0.9CN \end{aligned}$$

Since we chose the constant $C \geq 10a$, we have that

$$T_N \leq aN + 0.9CN \leq 0.1CN + 0.9CN = CN,$$

and hence by induction, for all $N > 0$,

$$T_N \leq CN = O(N).$$

Improving Quicksort

Finally, we note that we can use the Quickselect algorithm to select the optimal pivot for Quicksort, which is sometimes referred to as Balanced Quicksort. Doing so will result in guaranteed worst-case $O(N \log(N))$ performance! In practice, this strategy is outperformed by random pivot selection, but it is theoretically very satisfying to see Quicksort fall in line with the same worst-case behaviour as Heapsort and Merge sort.