Medians and Order Statistics Ch. 9

Let A be an ordered set containing n distinct elements:

Definition: The ith order statistic is the ith smallest element, e.g.,
- minimum = 1st order statistic
- maximum = nth order statistic
- median(s) = When n is odd, i = (n+1)/2
  When n is even, the lower median is i = n/2
  the upper median is i = n/2 + 1

Selection Problem: Find the ith order statistic

input: Set A of n (distinct) unsorted numbers, and a number i,
1 ≤ i ≤ n
output: The element x ∈ A that is larger than (i – 1) elements of A,
(i.e., the ith smallest element of A).

Easy O(nlgn) solution to selection problem

For n unique elements, an O(nlgn) solution should be readily apparent.

EasySelection(A, i)
1. A' = FavoriteSort(A)
2. return A'[i]

Idea: Use an O(nlgn) sorting algorithm, such as heapsort or mergesort. Then return the ith element in the sorted array.

Any ideas for an algorithm to find the minimum (without sorting)?
Finding the minimum or maximum is an easier problem than finding the ith order statistic.

Solutions to Selection Problem

First, we'll look at the problem of finding the minimum and maximum of a collection of unordered data.

Then, we'll see a simple general selection problem with a time bound of O(n) on average.

Finally, we'll look at a more complicated general selection algorithm with a time bound of O(n) in the worst case.

Finding Minimum (or Maximum)

Running Time:
- Just scan input array
- exactly n-1 comparisons

Is this the best possible time for finding the minimum? yes

Why are n - 1 comparisons necessary?
- Any algorithm that finds the minimum must compare all elements with the "leader" (think of a tournament).
- so...there must be at least n - 1 losers (and each loss requires a comparison)
- We must look at every key, otherwise the missed one may be the minimum. Each look (except the first) requires a comparison.

The maximum can be found in exactly the same way by replacing the > with a < in algorithm Minimum.

Finding Minimum & Maximum

What if we want to find both the minimum and maximum elements in a set simultaneously?

How many comparisons are necessary?
- Plan A: find the minimum and maximum separately using n - 1 comparisons for min and n - 1 for max = 2n - 2 comparisons
  Is it possible to do better?
- Plan B: Use the "tournament method". Process elements in pairs. Compare pairs of elements from the input first with each other and then compare the smaller to the current min and the larger to the current max, changing current values of max and/or min if necessary.
  Cost = at most 3 compares for every 2 elements.
  Total cost = 3(n/2).

Finding Minimum & Maximum

- Plan B: Process elements in pairs. Compare pairs of elements from the input first with each other and then compare the smaller to the current min and the larger to the current max, changing current values of max and/or min if necessary.
  Cost = 3 compares for every 2 elements. Total cost = 3[n/2].
Analysis of FindMin&Max

- If n is even, there is 1 initial compare and then \(3(n-2)/2 + 1\) compares = \(3n/2 – 2\).
- If n is odd, there are \(3(n-1)/2\) compares (no initial compare).
- In either case, the maximum number of compares is \(\leq 3n/2\).

**FindMin&Max(A)**
1. if A.length \(\%\) 2 == 0
   4. else
6. else
7. min = max = A[1]
8. Compare rest of elements in pairs, comparing only the larger of each pair to max and the smaller of each pair with min; reset max and min as larger or smaller elements are found

Finding \(i^{th}\) Order Statistic in (Expected) Linear Time

- Randomized-Partition first swaps \(A[r]\) with a random element of \(A\) and then proceeds as in Partition.

```
Randomized-Partition(A, p, r)
1. j = Random(p, r)
2. swap A[r] and A[j]
3. return Partition(A, p, r)
```

Finding \(i^{th}\) Order Statistic in (Expected) Linear Time

This is called prune-and-search because it lops off one side of the data and just searches the other side (like binary search).

```
Randomized-Select(A, p, r, i)
1. if p == r return A[p]
2. q = Randomized-Partition(A, p, r)
3. k = q – p + 1
4. if i == k return A[q]
5. else if i < k return Randomized-Select(A, p, q-1, i)
6. else return Randomized-Select(A, q+1, r, i - k)
```

Running Time of Randomized-Select

- Worst-case?

- Best-case?

- Average-case?

The book uses a lengthy probabilistic analysis that shows we can determine any order statistic in linear time, on average.

Selection in Worst-Case Linear Time

What if we don't want to rely on probability for a good running time?

There is a deterministic, linear-time algorithm to find the \(i^{th}\) order statistic.

```
Randomized-Select(A, p, r, i)
1. if p == r return A[p]
2. q = Randomized-Partition(A, p, r)
3. k = q – p + 1
4. if i == k return A[q]
5. else if i < k return Randomized-Select(A, p, q-1, i)
6. else return Randomized-Select(A, q+1, r, i - k)
```
Selection in Linear Worst-Case Time

Key: Guarantee a "good" split when array is partitioned – will yield an algorithm that always runs in linear time. The cost? A complex algorithm!

Select(A, i) /* i is the sought-after position and |A| > i */
1. if |A| == 1 return position of element in original array, x.
2. else
   a. divide input array A into n/5 groups of size 5 per group (and one leftover group if n % 5 != 0)
   b. find the median of each group of size 5 by insertion sorting each group of 5, picking the middle element of each group and putting it into an array A'.
   c. call Select recursively on A' to find x, the median of the n/5 medians.
3. q = Partition(A, p, r, x) /* Note: this partition is given the pivot, x */
4. if i == q return i
5. else if i < q call Select on the part of A < q
   else call Select on the part of A > q

Selection in Linear Worst-Case Time

Main idea: this algorithm guarantees that Partition causes a "good" split, with at least a constant fraction of the n elements ≤ x and a constant fraction > x.

Start the analysis by getting a lower bound on the number of elements that are greater than x, the median of medians.

- At least 1/2 of the medians found in step 2 are greater than the median of medians, x.

- Look at the groups containing medians greater than x (in gray). Each contributes 3 elements that are > x (the median of the group and the 2 elements in the group greater than the group's median), except for 2 of the groups: the group containing x (which has only 2 elements > x) and the group with < 5 elements. Discard these 2 groups.

Therefore, when we call Select recursively in step 5, it is on at most (7n/10) + 6 elements. Find this value by using

\[10n/10 - (3n/10 - 6) = (7n/10) + 6\]

Selection in Linear Worst-Case Time

Running Time of Select

Running Time (each step):
1. O(n) (break into groups of 5)
2. O(n) (sorting 5 numbers and finding median is O(1) time)
3. T(n/5) (recursive call to find median of medians)
4. O(n) (partition is linear time)
5. T(7n/10 + 6) (maximum size of subproblem)

Therefore, we get the recurrence

\[T(n) = T(n/5) + T(7n/10 + 6) + O(n)\]
Running Time of Select

Solve this recurrence using the "good guess" method. Guess $T(n) \leq cn$ for all $n \geq 140$

$T(n) = T([n/5]) + T(7n/10 + 6) + O(n)$

$\leq c[n/5] + c(7n/10 + 6) + an$

$\leq cn/5 + c + 7cn/10 + 6c + an$

$= 9cn/10 + 7c + an$

$= cn + (-cn/10 + 7c + an)$

$\leq cn$ if

$-cn/10 + 7c + an \leq 0$

$cn/10 - 7c \geq an$

$cn - 70c \geq 10an$

$c(n-70) \geq 10an$

$c \geq 10a(n/(n-70))$.

Choosing big enough $c$ makes $(-cn/10 + 7c + an)$ positive, so last line holds. (Try $c = 200$).

Thus, finding the $i^{th}$ order statistic can be done in linear time.

End of Lecture on Ch 9