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) = $\lfloor (n+1)/2 \rfloor$ and $\lceil (n+1)/2 \rceil$

Selection Problem: Find the ith order statistic for a given i
input: Set *A* of *n* (***distinct***) numbers, and a number *i*, $1 \le i \le n$
output: The element $x \in A$ that is larger than exactly $(i - 1)$ elements of *A*

O(nlgn) solution to selection problem

Selection Problem: Find the ith order statistic for a given i
input: Set *A* of *n* (distinct) numbers, and a number *i*, $1 \le i \le n$
output: The element $x \in A$ that is larger than exactly $(i - 1)$ elements of *A*

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

Running Time:
O(nlgn) for comparison-based sorting.
Can we do better???

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?

Finding Minimum (or Maximum)

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

```
Minimum(A)
1. lowest = A[1]
2. for i = 2 to n
3.       lowest = min(lowest, A[i])
```

Is this the best possible time for finding the minimum?

Finding Minimum & Maximum

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

How many comparisons are necessary?

- Plan A: find the minimum and maximum separately using n – 1 comparisons for min and n – 2 for max = 2n – 3 comparisons
  Is it possible to do better?
- 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 = at most 3 compares for every 2 elements.
  Total cost = $3\lfloor n/2 \rfloor$.

Finding Minimum & Maximum

```
FindMin&Max(A)
if length[A] % 2 == 0
   if A[1] > A[2]
      min = A[2]
      max = A[1]
   else
      min = A[1]
      max = A[2]
else // n % 2 == 1
   min=max=A[1]
Compare the rest of the
elements in pairs,
comparing only the
maximum element of each
pair with max and the
minimum element of each
pair with min
```

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
- In either case, the maximum number of compares is $\le 3\lfloor n/2 \rfloor$

```
FindMin&Max(A)
1.  if length[A] % 2 == 0
2.     if A[1] > A[2]
3.        min = A[2]
4.        max = A[1]
5.     else min = A[1]
6.          max = A[2]
7.  else // n % 2 == 1
8.       min=max=A[1]
9.  Compare the rest of the
    elements in pairs, comparing
    only the maximum element
    of each pair with max and the
    minimum element of each
    pair with min
```

Selection of ith-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] ↔ A[j]
3. return Partition(A, p, r)
```

Selection of ith-order statistic in (Expected) Linear Time

- Randomized-Select returns the ith smallest element of A.
  - like Randomized-QuickSort, except we only need to make one of the recursive calls. Why?

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

```
Randomized-Select(A, p, r, i)
1. if p = r then 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) \\ lower half
6. else return Randomized-Select(A, q+1, r, i - k) \\ upper half
```

Running Time of Randomized-Select

- Worst-case : unlucky with bad 0 : n - 1 partitions.
  $T(n) = T(n - 1) + \theta(n) = \theta(n^2)$
  (same as for worst-case of QuickSort)
- Best-case : really lucky and quickly reduce subarrays
  $T(n) = T(n/2) + \theta(n)$ (what is running time if we use the Master Theorem?)
- Average-case : like Quick-Sort, will be asymptotically close to best-case.

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.

```
Select(A, i)    /* i is the ith order statistic. */
1. divide input array A into ⌊n/5⌋ groups of size 5 per group
   (and one leftover group if n % 5 != 0)
2. find the median of each group of size 5 by insertion sorting
   the groups of 5, picking the middle elements of each group
   of 5 and putting it into an array A'.
3. call Select recursively on A' to find x, the median of the
   ⌈n/5⌉ medians.
4. partition A around x, splitting it into two arrays
   A[p, q-1] and A[q+1, r] and returning q, the index of the
   split point  (uses modified Partition on next slide).
5. if (i = q) return x
   else if (i < q) then
        call Select on the part of A < q
   else call Select on the part of A > q
```

Selection in Linear Worst-Case Time

Modified version of Partition that takes as an extra input parameter the value of the element to partition around, x.

```
Partition(A, p, r, x)
1. i = p - 1
2. for j = p to r – 1
3.      if A[j] ≤ x
4.           i = i + 1
5.           swap A[i] and A[j]
6. swap A[i+1] and  A[r]
7. return i + 1
```

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.

Note:

- 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. 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.

Selection in Linear Worst-Case Time

- Thus, we know that at least

$$3(\lceil 1/2 \lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6$$

elements are > x (Symmetrically, the number of elements that are < x is at least 3n/10 - 6).

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

$$10n/10 - (3n/10 - 6) = (7n/10) + 6$$

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($\lceil n/5 \rceil$) (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

$$T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

Running Time of Select

Solve this recurrence using a good guess. Guess $T(n) \leq cn$

$$\begin{aligned} T(n) &= T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) \\ &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + O(n) \\ &\leq c((n/5) + 1) + 7cn/10 + 6c + O(n) \\ &= cn - (cn/10 - 7c) + O(n) \\ &\leq cn \end{aligned}$$

When n >= 80 (cn/10 -7c) is positive

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