CS 315 - Day 18, i-th Order Statistics

Today we discuss a nifty idea, finding the $\boldsymbol{i}$-th Order Statistic of $\boldsymbol{N}$ elements.

$\boldsymbol{i} = 1$ would be the minimum or "first" order statistic

$\boldsymbol{i} = N$ would be the maximum or "$\boldsymbol{N}$-th" order statistic

The first order (or $\boldsymbol{N}$-th order) statistic are pretty easy to find, just iterate through the elements, keeping track of the current minimum (maximum).

```
Minimum(A)
min = A[1]
for i =2 to length[A] do          → N − 1 comparisons
   if min > A[i]                  → θ(N)
      then min = A[i]
return min
```

What about finding the second order ($\boldsymbol{i} = 2$) statistic, the second smallest, or the $N - 1$ order, the second largest?

- We could sort, $O(N \cdot \lg N)$ and then index directly.
- We could find the minimum (or maximum), discard it, and then repeat.
  - How many comparisons? $N - 1$, then $N - 2$, gives $2N - 3$, or just $\theta(N)$ asymptotically.
  - Can we do better than this?
  - While finding the smallest (or largest) element, it's the only element the second smallest (or largest) element "lost" to. If we trade the memory for computations, and keep track of the elements the smallest (or largest) beat, then searching this smaller set for the smallest (or largest), it will be $O(\lg N)$, which gives $N - 1 + O(\lg N) - 2$, which is fewer comparisons (but still $O(N)$).

Is there an efficient approach to the general selection problem? Acutally, yes, there is an $O(N)$ algorithm to calculate the $\boldsymbol{i}$-th Order Statistic of $\boldsymbol{N}$ elements.

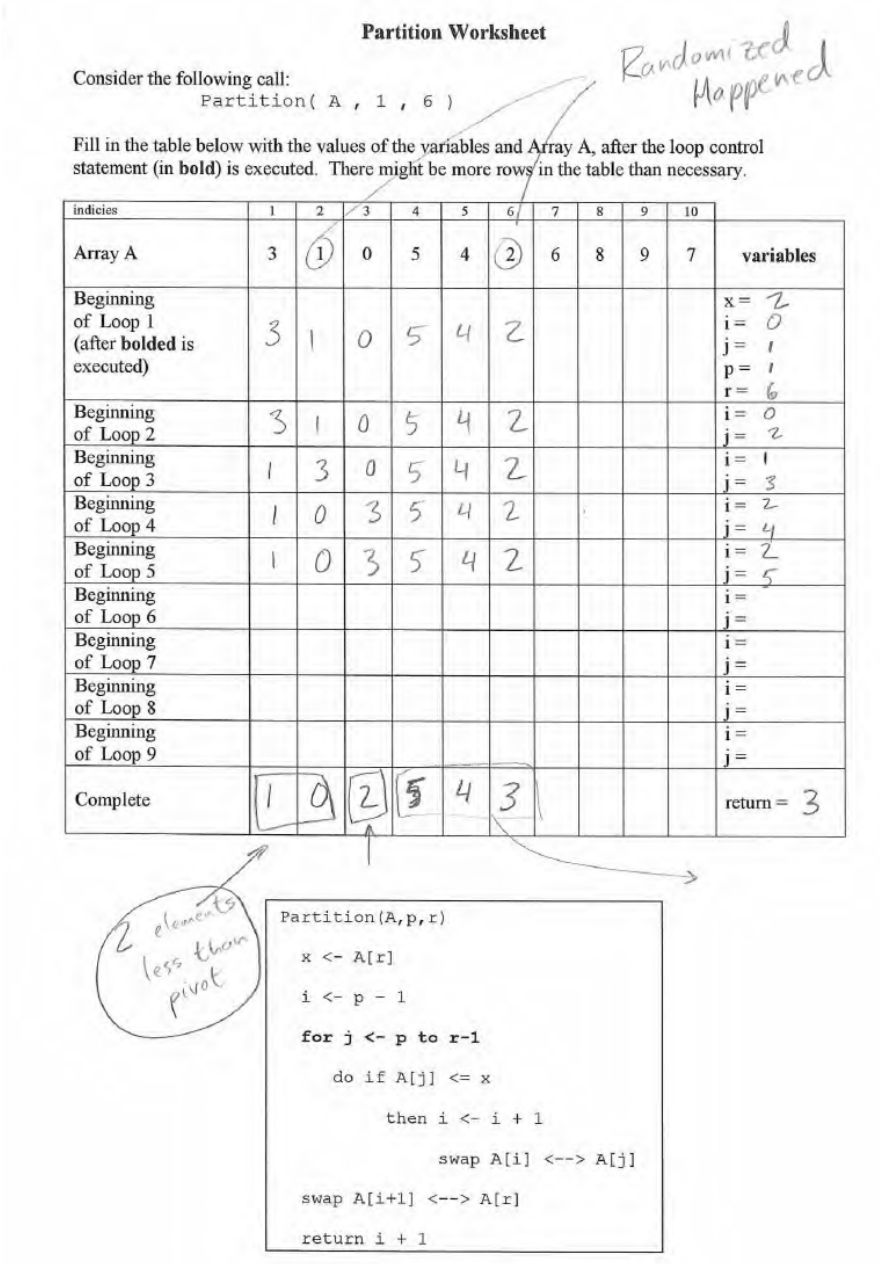
The detailed derivation is beyond the scope of the course, but covered in the textbook. We motivate this by thinking of `Partition`. There is one element in sorted order after a call to `Partition`, and that is the element partitioned around. The $\boldsymbol{i}$-th Order Statistic will either be that element, or in one of the sub-lists. Remember, the element partitioned around will be greater than all elements in the left sub-list, and smaller than all the elements in the right sub-list.

If we partition roughly in half, our comparisons will be on the order of $N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots + 1 \rightarrow 2N$. We only need to keep track of one partition, our problem size is cut in half each time.

Let's consider the `Randomized-Select` routine.

```
Randomized-Select(A,p,r,i)
   if p=r
      then return A[p]
     q = Randomized-Partition(A,p,r)
   k = q-p+1          // gives position of element partitioned around
                      // A[q] is in sorted location and remember p might
                      // not be 1
   if i = k
      then return A[q]
      elseif i < k  // continue with the left partition
           then return Randomized-Select(A,p,q-1,i)
      else return Randomized-Select(A,q+1,r,i-k) // right partition
                                                 // and shifting the
                                                 // i value
```

For the shift, if we are looking for the 7th Order Statistic, and $q = 4$, the 7th Order Statistic will be in the 3rd position in the right partition.



Partition Worksheet

Consider the following call:
Partition(A , 1 , 6)

Randomized Happened

Fill in the table below with the values of the variables and Array A, after the loop control statement (in **bold**) is executed. There might be more rows in the table than necessary.

indicies	1	2	3	4	5	6	7	8	9	10	
Array A	3	(1)	0	5	4	(2)	6	8	9	7	variables
Beginning of Loop 1 (after **bolded** is executed)	3	1	0	5	4	2					x = 2 i = 0 j = 1 p = 1 r = 6
Beginning of Loop 2	3	1	0	5	4	2					i = 0 j = 2
Beginning of Loop 3	1	3	0	5	4	2					i = 1 j = 3
Beginning of Loop 4	1	0	3	5	4	2					i = 2 j = 4
Beginning of Loop 5	1	0	3	5	4	2					i = 2 j = 5
Beginning of Loop 6											i = j =
Beginning of Loop 7											i = j =
Beginning of Loop 8											i = j =
Beginning of Loop 9											i = j =
Complete	1	0	2	5	4	3					return = 3

2 elements less than pivot

```
Partition(A,p,r)
  x <- A[r]
  i <- p - 1
  for j <- p to r-1
     do if A[j] <= x
           then i <- i + 1
                swap A[i] <--> A[j]
  swap A[i+1] <--> A[r]
  return i + 1
```

So, if the partition element is chosen randomly, here is an example, with the partition element 2 is in the 3rd position, $q = 3$.

If the $\boldsymbol{i}$-th Order Statistic is 1, it will stay 1, but if it was larger, say 5, the new $\boldsymbol{i}$-th Order Statistic will be $5 - 3 = 2$.

Let's pull this together and incorporate it into a review.

In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Given a set of *n* numbers, we wish to find the *i* largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of *n* and *i*. ***We assume that the numbers start out in an array.***

3 (a) Sort the numbers, and list the *i* largest.

3 (b) Build a max-priority-queue from the numbers and call *Extract-Max i* times.

3 (c) Use an order-statistic algorithm to find the *i*-th largest number, partition around that number, and sort the *i* largest numbers.

3 (a) Sort the numbers, and list the *i* largest.

Sort the numbers, using Mergesort, in n*lg(n) worst-case time, and put the i largest elements (directly accessible in the sorted array) into the output array, taking i time.
Total worst-case running time: ***Θ(i + n*lg(n))*** (because i $\leq$ n).

3 (b) Build a max-priority-queue from the numbers and call *Extract-Max i* times.

Implement the priority queue as a heap, in Θ(n) time with ***Build-Max-Heap***, and then call ***Extract-Max*** i times to get the i largest elements. Since ***Max-Heapify*** is Θ(lg(n)), this portion is Θ(i * lg(n)) worst case time, because i extractions from a heap with O(n) elements takes
i · O(lg n) = O(i lg n) time, and half of the i extractions are from a heap with $\geq$ n/2 elements, so those i/2 extractions take (i/2)*Ω(lg(n/2)) = Ω(i lg n) time in the worst case.
Combining we get total worst case running time: ***Θ(n + i * lg(n))***

3 (c) Use an order-statistic algorithm to find the *i*-th largest number, partition around that number, and sort the *i* largest numbers.

Find the i-th largest in Θ(n) time using the i-th order statistic algorithm, then
call ***Partition*** to partition around this i-th largest in Θ(n) time, and finally
sort the i largest in i * lg(i) time using Mergesort.
Combining we get total worst-case running time: Θ (n + n + i * lg(i))
which simplifies slightly to ***Θ (n + i * lg(i))***

Note that method (c) is always asymptotically at least as good as the other two methods, and that method (b) is asymptotically at least as good as (a). (Comparing (c) to (b) is easy, but it is less obvious how to compare (c) and (b) to (a). (c) and (b) are asymptotically at least as good as (a) because n, i lg i , and i*lg n are all O(n lg n). The sum of two things that are O(n lg n) is also O(n lg n).)