Selection and Adversary Arguments

COMP 215 Lecture 19

Selection Problems

- We want to find the *k*'th largest entry in an unsorted array.
  - Could be the largest, smallest, median, etc.
- Ideas for an $n \lg n$ algorithm?
- We will think about:
  - Largest, smallest
  - Largest AND smallest
  - Second largest
  - *K*th.

Finding the Largest Item

- We have seen a simple algorithm that requires n-1 comparisons in the worst case.
- That's optimal.

Largest and Smallest

- We could run the previous algorithm twice, which would give us $2n$-2 comparisons.
- We have seen an algorithm that requires $\frac{3n}{2} - 2$ comparisons in the worst case.
  - Anyone remember?
  - Is this optimal?
- First idea is to consider a decision tree.
  - There must be at least n leaves.
  - Therefore the height must be at least $\lceil \lg n \rceil$.
  - Obviously not a very tight bound.

Adversary Arguments

- We design an adversary that forces any algorithm to do as much work as possible.
- The adversary does not have a particular solution in mind – all answers are chosen to reveal as little information as possible while being consistent with earlier answers.
- The analysis proceeds by
  - Determining what information the algorithm needs to solve the problem.
  - Determining how long it will take to get that information from the adversary.

Adversary for Largest/Smallest

- In order to determine both the largest and the smallest we assign states to keys:
  - X – Key has not been involved in comparison. (0 unit)
  - L – Key has lost at least one comparison. (1 unit)
  - W – Key has won at least one comparison. (1 unit)
  - WL – Key has won and lost a comparison. (2 units)
- We cannot determine the largest and smallest keys until
  - All keys except one have lost a comparison: n-1 units.
  - All keys except one have won a comparison: n-1 units.
  - We need to learn $2n$-2 things overall.
- We design an adversary that reveals as little as possible.

Adversary for Largest/Smallest

Before		Winner	After		Information
X	X	1	W	L	2
X	L	1	W	L	1
X	W	2	L	W	1
X	WL	1	W	WL	1
L	L	1	WL	W	1
L	W	2	L	W	0
L	WL	2	L	WL	0
W	W	1	W	WL	1
W	WL	1	W	WL	0
WL	WL	be consistent	W	WL	0

Adversary for Largest/Smallest

- We need to get $2n$-2 items of information.
- How many comparisons are necessary?
- The most information per comparison (2) occurs when neither item has one or lost.
  - This can happen at most $n/2$ times for a total of n items of information.
- Any other comparison results in at most 1 unit of information.
  - We need at least $2n\text{-}2 - n = n\text{-}2$ of these.
- Total number of comparisons is $n\text{-}2 + n/2 = 3n/2 - 2$.
- This lower bound matches the worst case of our algorithm.

Second Largest Item

- One possibility is to find the largest, remove it, then find the largest remaining: $(n\text{-}1) + (n\text{-}2) = 2n - 3$ comparisons.
- A better alternative is the tournament approach.
- Hold a tournament to determine the largest key.
- Since the second largest key must be defeated by the largest key at some point,
- We look for the largest key among those keys beaten by the largest key.

Second Largest Analysis

- We will just do the analysis for powers of 2.
- First, how many comparisons in the tournament:

$$\frac{n}{2}+\frac{n}{2^2}+\cdots+\frac{n}{2^{\lg n}} \;=\; n\sum_{i=1}^{\lg n}\left(\frac{1}{2}\right)^i \;=\; n-1$$

- Next, how many keys did the max key defeat? $\lg n$
- Therefore searching for the max among the defeated keys will take $\lg n - 1$ comparisons.
- Total comparisons: $(n - 1) + (\lg n - 1) = n + \lg n - 2$.
- If n is not a power of 2: $n+\lceil \lg n \rceil - 2$
- Is this optimal?

Second Largest Adversary

- For any algorithm, the largest key is involved in some number of comparisons m.
- Discounting the largest key, there are n - 1 keys. It takes at least n - 2 comparisons to find the second largest among those keys.
- The m comparisons with the largest key cannot count toward that n-2.
- Therefore the total number of comparisons required is at least $m + n - 2$.
- We will construct an adversary that forces m to be at least $\lceil \lg n \rceil$.

Second Largest Adversary

- Adversary builds a tree, and uses it to guide answers.
- Initially all nodes are roots (throughout a root node will represent a key that hasn't lost a comparison.)
- Two roots compared:
  - If both trees are the same size, answer is arbitrary, smaller is made child of larger.
  - If one tree is larger, the root of the smaller tree loses, and is made a child of the root of the larger tree.
- A root and a non-root are compared.
  - The non-root is declared smaller, and trees are not changed.
- Two non-roots are compared.
  - Answer is consistent with previous answers, and trees are not changed.

Second Largest Adversary

- When any correct algorithm terminates, there can be at most one root.
  - Otherwise there are two keys that never lost a comparison.
- The rules on the previous slide are designed so that the tree with the largest key as a root grows as slowly as possible.

Second Largest Adversary

- The question is: how many comparisons must a key have been involved in to end up at the root of the final tree?
- The size of the tree rooted at the largest tree at most doubles after each comparison, after the *k*th comparison:

$$\text{size}_k \leq 2\,\text{size}_{k-1}$$

- Initial size of the tree rooted at the largest tree is 1.

$$\text{size}_0 = 1$$

- Homogeneous linear recurrence. Solution is: $\text{size}_k \leq 2^k$.

$$n = \text{size}_m \leq 2^m$$

$$\lg n \leq m$$

$$\lceil \lg n \rceil \leq m$$

Second Largest Adversary

- Final result is a $n+\lceil \lg n \rceil - 2$ lower bound.
- Recall that our tournament algorithm required $n+\lceil \lg n \rceil - 2$ comparisons.

*K*th Smallest Entry

- We can solve this problem with order *n* comparisons (average case) with a small modification to quicksort.
- Recall, that after each call to partition, the pivot item is in its final sorted position.
- If the pivot item is at the *k*th position, then we have the *k*th smallest item.
- Here is the algorithm:
  - Partition the data, if pivot point = *k*, terminate.
  - if pivot point > *k*, partition the array up to pivot position.
  - if pivot point < *k*, partition the array after the pivot position.
  - Repeat until pivot position = *k*.

Selection Analysis

- Worst case comparisons?
- Average case comparisons?

Selection Analysis

- Worst case comparisons?
  - Same as quicksort: $\Theta(n^2)$.
- Average case comparisons?
  - In order to do an average case analysis we sum up the cost associated with every possible input, divide by the number of possible inputs.
- Assume that every *k* is equally possible, and every pivot point is equally possible after a partition.

Selection Analysis Average Case

- Input size of recursive calls can be anything from 0 to n-1.
- Size is 0 if $k = p$ (pivot point) after first partition. There are n ways for that to happen.
- Size is 1 if $k = 1$ and $p = 2$, or if k=n and p = n-1... two possible ways.
- Four ways for size to be 2.
- ...
- 2(n-1) ways for size to be n-1.

Selection Analysis Average Case

- This leads to an ugly recurrence:

$$A(n)=\frac{nA(0)+2A(1)+4A(2)+\cdots+2(n-1)A(n-1)}{n+2+4+\cdots+2(n-1)}+n-1$$

- The book informs us that this works out to:

$$A(n)\approx 3n$$

Selection in Worst Case Linear Time

- In our previous algorithm we wanted the pivot item to be near the median.
  - This causes the size of the array to be roughly halved on each recursive call.
- No obvious way to select a pivot item near the median, without knowing what the median is.
- Determining the median requires solving the selection problem. Doh.
- Amazingly, there is a way forward...

Linear Time Selection

- Alter our partition function as follows:
  - First break the array into n/5 groups.
  - Compute the median of each of those groups.
    - This can be done with six comparisons per group.
    - Alternately, just use insertion sort on each group.
  - Now, recursively call the selection algorithm to determine the median of our n/5 medians.
    - It turns out that this median of medians will be reasonably close to the median.
  - Use the result as the pivot point and partition as usual.
- The resulting algorithm is worst case linear time.

Linear Selection Analysis

- First, how close is the median of medians to the median?
- Let's draw a picture...
- So, roughly speaking, there are at most 3(n/10) items larger than the median of medians, and at most that many smaller.
- So the worst case size of the input to the recursive call is about 7n/10.
  - Book comes up with 7n/10 – 3/2.

Linear Selection Analysis

- We can write the following recurrence to describe the worst case number of comparisons:

$$W(n) = W\left(\frac{7n}{10} - \frac{3}{2}\right) + W\left(\frac{n}{5}\right) + \frac{6n}{5} + n$$

Worst case cost of the next recursive call

cost of finding the median of n/5 medians

cost of finding the n/5 medians

cost of partition

Linear Selection Analysis

- We don't really have any tools for dealing with this recurrence:

$$W(n) = W\left(\frac{7n}{10} - \frac{3}{2}\right) + W\left(\frac{n}{5}\right) + \frac{11n}{5}$$

- Revert to guess and check.
- We guess that it is $W(n)$ is linear, i.e. $W(n) <= cn$.
- Therefore:

$$c\left(\frac{7n}{10} - \frac{3}{2}\right) + c\frac{n}{5} + \frac{11n}{5} \leq cn$$

- Solving this, we get 22 <= c.
- We then use induction to prove that $W(n) <= 22n$.

Linear Selection Analysis

- There are selection algorithms that are closer to $3n$ in the worst case.
- We could construct an adversary argument to provide a lower bound for this problem.
- The highest lower bound determined so far is a bit more than $2n$.

Computer Algorithms, Baase and Van Gelder, 2000.