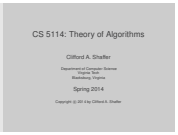
CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014



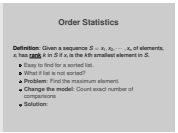
Title page

Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

Order Statistics

Definition: Given a sequence $S = x_1, x_2, \cdots, x_n$ of elements, x_i has **rank** k in S if x_i is the kth smallest element in S.

- Easy to find for a sorted list.
- What if list is not sorted?
- **Problem**: Find the maximum element.
- **Change the model**: Count exact number of comparisons
- **Solution**:

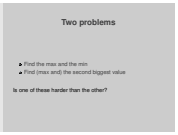


Finding max: Compare element n to the maximum of the previous $n-1$ elements. Cost: $n-1$ comparisons. This is optimal since you must look at every element to be sure that it is not the maximum.

Two problems

- Find the max and the min
- Find (max and) the second biggest value

Is one of these harder than the other?



Of course both can be done in $\Theta(n)$ time, but we want to count exact number of comparisons.

Both can also be done by finding max, then finding min or second max. So both can be done in 2n-1 comparisons.

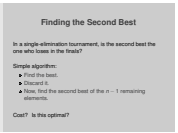
Finding the Second Best

In a single-elimination tournament, is the second best the one who loses in the finals?

Simple algorithm:

- Find the best.
- Discard it.
- Now, find the second best of the $n-1$ remaining elements.

Cost? Is this optimal?



As we discuss this problem, we consider *exact* counts, not asymptotics.

Not necessarily – the best 2 could compete in the first round! Note that we ignore variations in performance, the outcome between two players will always be the same.

$2n-3$.

To know, need a lower bound on the problem.
Naive: $\approx n$ might work. Clearly not optimal here! But, tighten lower bound.

Lower Bound for Second (1)

Lower bound:

- Anyone who lost to anyone who is not the max cannot be second.
- So, the only candidates are those who lost to max.
- `Find_max` might compare max to $n-1$ others.
- Thus, we might need $n-2$ additional comparisons to find second.
- Wrong!

Lower Bound for Second (1)
Lower bound:
- Anyone who lost to anyone who is not the max cannot be second.
- So, the only candidates are those who lost to max.
- Find_max might compare max to n − 1 others.
- Thus, we might need n − 2 additional comparisons to find second.
- Wrong!

What is wrong with this argument?
It relies on the behavior of a particular algorithm.

Lower Bound for Second (2)

The previous argument exhibits the **necessity fallacy**:

- Our algorithm does something, therefore all algorithms solving the problem must do the same.

Alternative: Divide and conquer

- Break the list into two halves.
- Run `Find_max` on each half.
- Compare the winners.
- Run `Find_max` on the winner's half for second.
- Compare that second to second winner.

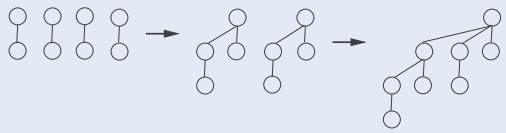
Cost: $\lceil 3n/2 \rceil - 2$.
Is this optimal?
What if we break the list into four pieces? Eight?

Lower Bound for Second (2)
The previous argument exhibits the necessity fallacy:
- Our algorithm does something, therefore all algorithms solving the problem must do the same.
Alternative: Divide and conquer
- Break the list into two halves.
- Run Find_max on each half.
- Compare the winners.
- Run Find_max on the winner's half for second.
- Compare that second to second winner.
Cost: ⌈3n/2⌉ − 2.
Is this optimal?
What if we break the list into four pieces? Eight?

In particular, it is not necessary that the max element compare with $n-1$ others, even in the worst case.
$\lfloor n/2 \rfloor - 1 + \lceil n/2 \rceil - 1 \ldots + 1 = n - 1$.
Worst case: $\lceil n/2 \rceil - 1$ elements, since winner need not compete again.
$+1$.
Cost of $\lceil 3n/2 \rceil - 2$ just closed half of the gap between our old lower bound and our old algorithm – pretty good progress!
4: about $5/4$.
8: $n - 1 + \lceil n/8 \rceil - 1 = \lceil 9n/8 \rceil - 2$.
What if we do this recursively?
$f(n) = 2f(n/2) + 2; f(1) = 0$ which is $3n/2 - 2$, which is no better than halves. So recursive divide & conquer (in a naive way) does not work! Quarters would be better!

Binomial Trees (1)

- Pushing this idea to its extreme, we want each comparison to be between winners of equal numbers of comparisons.
- The only candidates for second are losers to the eventual winner.
- A **binomial tree** of height m has 2^m nodes organized as:
  - a single node, if $m = 0$, or
  - two height $m-1$ binomial trees with one tree's root becoming a child of the other.



Binomial Trees (1)
- Pushing this idea to its extreme, we want each comparison to be between winners of equal numbers of comparisons.
- The only candidates for second are losers to the eventual winner.
- A binomial tree of height m has 2^m nodes organized as:
  - a single node, if m = 0, or
  - two height m − 1 binomial trees with one tree's root becoming a child of the other.

but, we want as *few* of these as possible.

Binomial Trees (2)

Algorithm:

- Build the tree.
- Compare the $\lceil \log n \rceil$ children of the root for second.

Cost?

Binomial Trees (2)
Algorithm:
- Build the tree.
- Compare the ⌈log n⌉ children of the root for second.
Cost?

$n + \lceil \log n \rceil - 2$.

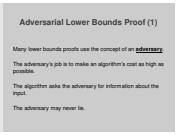
Adversarial Lower Bounds Proof (1)

Many lower bounds proofs use the concept of an **adversary**.

The adversary's job is to make an algorithm's cost as high as possible.

The algorithm asks the adversary for information about the input.

The adversary may never lie.



no notes

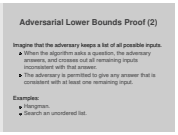
Adversarial Lower Bounds Proof (2)

Imagine that the adversary keeps a list of all possible inputs.

- When the algorithm asks a question, the adversary answers, and crosses out all remaining inputs inconsistent with that answer.
- The adversary is permitted to give any answer that is consistent with at least one remaining input.

Examples:

- Hangman.
- Search an unordered list.



Adversary maintains dictionary, and can give any answer that conforms with at least one entry in the dictionary.

Adversary always says "not found" until last element.

Lower Bound for Second Best

At least $n-1$ values must lose at least once.

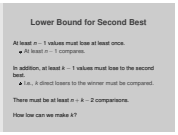
- At least $n-1$ compares.

In addition, at least $k-1$ values must lose to the second best.

- I.e., k direct losers to the winner must be compared.

There must be at least $n+k-2$ comparisons.

How low can we make k?



What does your intuition tell you as a lower bound for k? $\Omega(n)$? $\Omega(\log n)$? $\Omega(c)$?

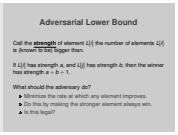
Adversarial Lower Bound

Call the **strength** of element $L[i]$ the number of elements $L[i]$ is (known to be) bigger than.

If $L[i]$ has strength a, and $L[j]$ has strength b, then the winner has strength $a+b+1$.

What should the adversary do?

- Minimize the rate at which any element improves.
- Do this by making the stronger element always win.
- Is this legal?



The winner has now proved stronger than $a+b+$ the one who just lost.

Yes. The adversary cannot "fix" the fight to give contradictory answers. But, it *can* give answers consistent with *some* legal input.

Lower Bound (Cont.)

What should the algorithm do?

If $a \geq b$, then $2a \geq a + b$.

- From the algorithm's point of view, the best outcome is that an element doubles in strength.
- This happens when $a = b$.
- All strengths begin at zero, so the winner must make at least k comparisons for $2^{k-1} < n \leq 2^k$.

Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons.

Lower Bound (Cont.)
What should the algorithm do?
If $a \geq b$, then $2a \geq a + b$.
- From the algorithm's point of view, the best outcome is that an element doubles in strength.
- This happens when $a = b$.
- All strengths begin at zero, so the winner must make at least k comparisons for $2^{k-1} < n \leq 2^k$.

Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons.

Need to get the final strength up to $n - 1$.
These k losers are candidates for 2nd place.

Min and Max

Problem: Find the minimum AND the maximum values.
Naive Solution: Do independently, requires $2n - 3$ comparisons.

Solution: By induction.

Base cases:

- 1 element: It is both min and max.
- 2 elements: One comparison decides.

Induction Hypothesis:

- Assume that we can solve for $n - 2$ elements.

Try to add 2 elements to the list.

Min and Max
Problem: Find the minimum AND the maximum values.
Naive Solution: Do independently, requires $2n - 3$ comparisons.
Solution: By induction.
Base cases:
- 1 element: It is both min and max.
- 2 elements: One comparison decides.

Induction Hypothesis:
- Assume that we can solve for $n - 2$ elements.

Try to add 2 elements to the list.

We are adding items n and $n - 1$.

Conceptually: ? compares for $n - 2$ elements, plus one compare for last two items, plus cost to join the partial solutions.

Min and Max (2)

Induction Hypothesis:

- Assume that we can solve for $n - 2$ elements.

Try to add 2 elements to the list.

- Find min and max of elements $n - 1$ and n (1 compare).
- Combine these two with $n - 2$ elements (2 compares).
- Total incremental work was 3 compares for 2 elements.

Total Work:

What happens if we extend this to its logical conclusion?

Min and Max (2)
Induction Hypothesis:
- Assume that we can solve for $n - 2$ elements.

Try to add 2 elements to the list.
- Find min and max of elements $n - 1$ and n (1 compare).
- Combine these two with $n - 2$ elements (2 compares).
- Total incremental work was 3 compares for 2 elements.

Total Work:

What happens if we extend this to its logical conclusion?

Total work is about $3n/2$ comparisons.

It doesn't get any better if we split the sequence into two halves. The recurrence is:

$$T(n) = \begin{cases} 1 & n = 2 \\ 2T(n/2) + 2 & n > 2 \end{cases}$$

This is $3/2n - 2$ for n a power of 2.

The Lower Bound (1)

Is $\lceil 3n/2 \rceil - 2$ optimal?

Consider all states that a successful algorithm must go through: The **state space** lower bound.

At any given instant, track the following four categories:

- Novices: not tested.
- Winners: Won at least once, never lost.
- Losers: Lost at least once, never won.
- Moderates: Both won and lost at least once.

The Lower Bound (1)
Is $\lceil 3n/2 \rceil - 2$ optimal?
Consider all states that a successful algorithm must go through: The **state space** lower bound.
At any given instant, track the following four categories:
- Novices: not tested.
- Winners: Won at least once, never lost.
- Losers: Lost at least once, never won.
- Moderates: Both won and lost at least once.

no notes

The Lower Bound (2)

Who can get ignored?

What is the initial state?

What is the final state?

How is this relevant?

The Lower Bound (2)

Who can get ignored?

What is the initial state?

What is the final state?

How is this relevant?

Moderates – Can't be min or max.

Initial: (n, 0, 0, 0).

Final: (0, 1, 1, n-2).

We must go from the initial state to the final state to solve the problem.
So, we can analyze how this gets done.

Lower Bound (3)

Every algorithm must go from $(n, 0, 0, 0)$ to $(0, 1, 1, n-2)$.

There are 10 types of comparison.

Comparing with a moderate cannot be more efficient than other comparisons, so ignore them.

Lower Bound (3)

Every algorithm must go from $(n, 0, 0, 0)$ to $(0, 1, 1, n-2)$.

There are 10 types of comparison.

Comparing with a moderate cannot be more efficient than other comparisons, so ignore them.

That gets rid of 4 types of comparisons.

Lower Bound (3)

If we are in state (i, j, k, l) and we have a comparison, then:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:L$	$(i,$	$j,$	$k,$	$l)$
or	$(i,$	$j-1,$	$k-1,$	$l+2)$

Lower Bound (3)

If we are in state (i, j, k, l) and we have a comparison, then:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:L$	$(i,$	$j,$	$k,$	$l)$
or	$(i,$	$j-1,$	$k-1,$	$l+2)$

no notes

Adversarial Argument

What should an adversary do?

- Comparing a winner to a loser is of no value.

Only the following five transitions are of interest:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$

Only the last two types increase the number of moderates, so there must be $n-2$ of these.

The number of novices must go to 0, and the first is the most efficient way to do this: $\lceil n/2 \rceil$ are required.

Adversarial Argument

What should an adversary do?

- Comparing a winner to a loser is of no value.

Only the following five transitions are of interest:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$

Only the last two types increase the number of moderates, so there must be $n-2$ of these.

The number of novices must go to 0, and the first is the most efficient way to do this: $\lceil n/2 \rceil$ are required.

Minimize information gained.

Adversary will just make the winner win – No new information is provided.

This provides an algorithm.

Kth Smallest Element

Problem: Find the kth smallest element from sequence S.

(Also called **selection**.)

Solution: Find min value and discard (k times).

- If k is large, find $n - k$ max values.

Cost: $O(\min(k, n-k)n)$ – only better than sorting if k is $\mathrm{O}(\log n)$ or $O(n - \log n)$.

Kth Smallest Element
Problem: Find the kth smallest element from sequence S.
(Also called **selection**.)
Solution: Find min value and discard (k times).
- If k is large, find $n - k$ max values.
Cost: $O(\min(k, n-k)n)$ – only better than sorting if k is $\mathrm{O}(\log n)$ or $O(n - \log n)$.

no notes

Better Kth Smallest Algorithm

Use quicksort, but take only one branch each time.

Average case analysis:

$$f(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(f(i-1))$$

Average case cost: $\mathrm{O}(n)$ time.

Better Kth Smallest Algorithm
Use quicksort, but take only one branch each time.
Average case analysis:
$$f(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(f(i-1))$$
Average case cost: $\mathrm{O}(n)$ time.

Like Quicksort, it is possible for this to take $O(n^2)$ time!!
It is possible to guarentee average case $O(n)$ time.