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?

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

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?

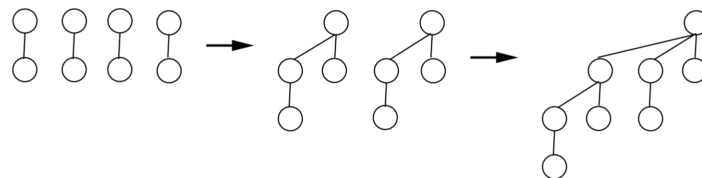
Binomial Trees

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.



Algorithm:

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

Cost?

Binomial Tree Representation

We could store the binomial tree as an explicit tree structure.

We can also store the binomial tree implicitly: In an array.

Assume two trees, each with 2^k nodes, are in the array as:

- First tree in positions 1 to 2^k.
- Second tree in positions $2^k + 1$ to 2^{k+1}.
- The root of a subtree is in the final array position for that subtree.

To join:

- Compare the roots of the subtrees.
- If necessary, swap subtrees so larger root element is second subtree.

Trades space for time.

Adversarial Lower Bounds Proof

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.

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.

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?

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?

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.