Durham University

Algorithms & Data Structures

Part – 3: Topic 5

Anish Jindal
anish.jindal@durham.ac.uk

Lower bounds

Plan

Lower bounds for sorting – decision tree argument

Lower bounds for selection – adversary argument

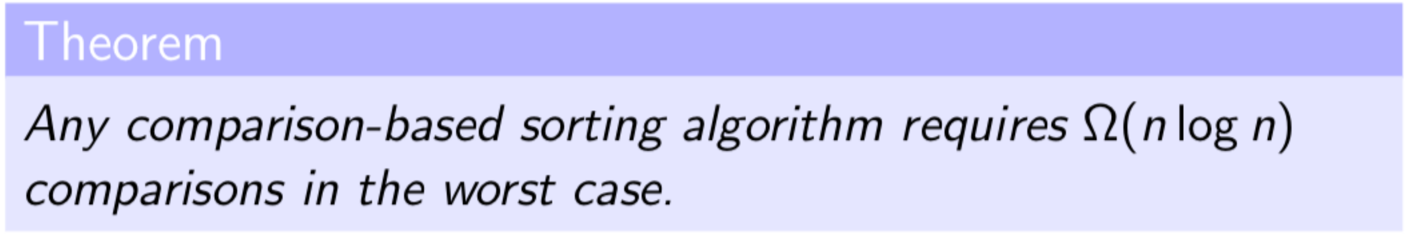
Lower bound for sorting

We've seen a few **comparison-based** sorting algorithms, e.g.:

MergeSort, HeapSort: upper bound O(n log n)

SelectionSort, QuickSort: upper bound O(n^2)

We discussed earlier:



Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Sorting and decision trees

A decision tree is a **full binary tree** (every node has either zero or two children)

It represents comparisons between elements performed by particular algorithm run on particular (size of) input

Only **comparisons** are relevant, everything else is ignored

Decision trees

Internal nodes are labelled i : j for 1 ≤ i, j ≤ n, meaning elements i and j are compared (indices w.r.t. original order)

Downward-edges are labelled “≤” or >, depending on the outcome of the comparisons

Leaves are labelled with some permutation of {1, . . . , n}

A **branch** from root to leaf describes sequence of comparisons (nodes and edges) and ends in some resulting sequence (leaf)

We have at least n! leaves – at least one for each outcome

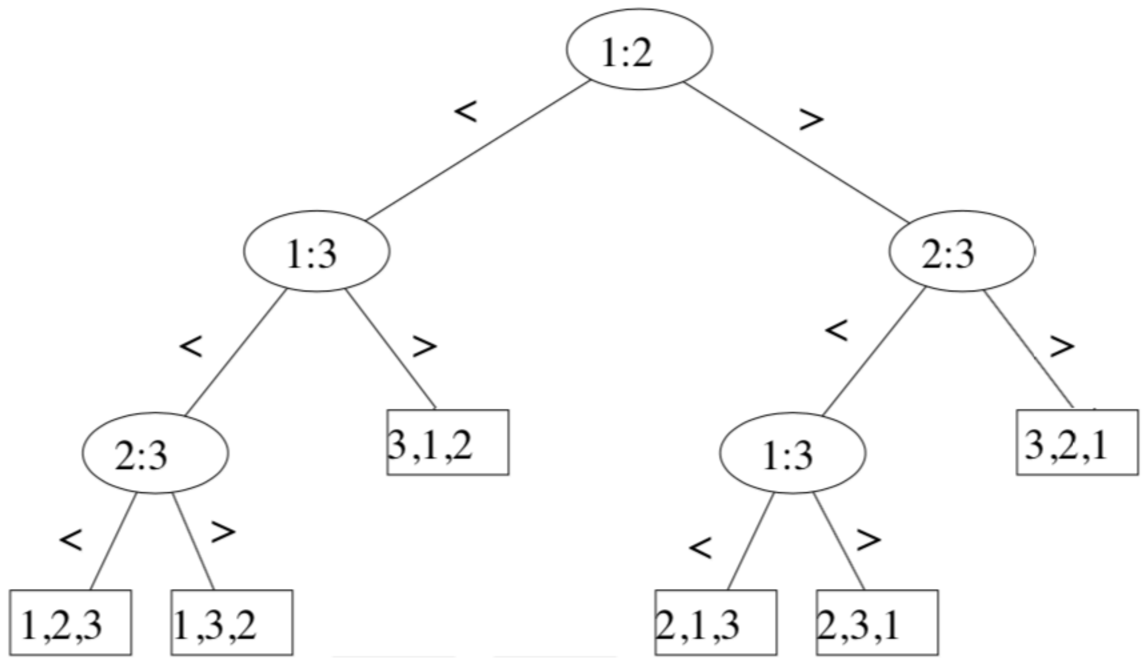
Example: SelectionSort

Reminder:

SelectionSort $(a_1, \ldots, a_n \in \mathbb{R}, n \geq 2)$

```
 1: for $i = 1$ to $n - 1$ do
 2:     $elem = a_i$
 3:     $pos = i$
 4:     for $j = i + 1$ to $n$ do
 5:         if $a_j < elem$ then
 6:             $elem = a_j$
 7:             $pos = j$
 8:         end if
 9:     end for
10:     swap $a_i$ and $a_{pos}$
11: end for
```

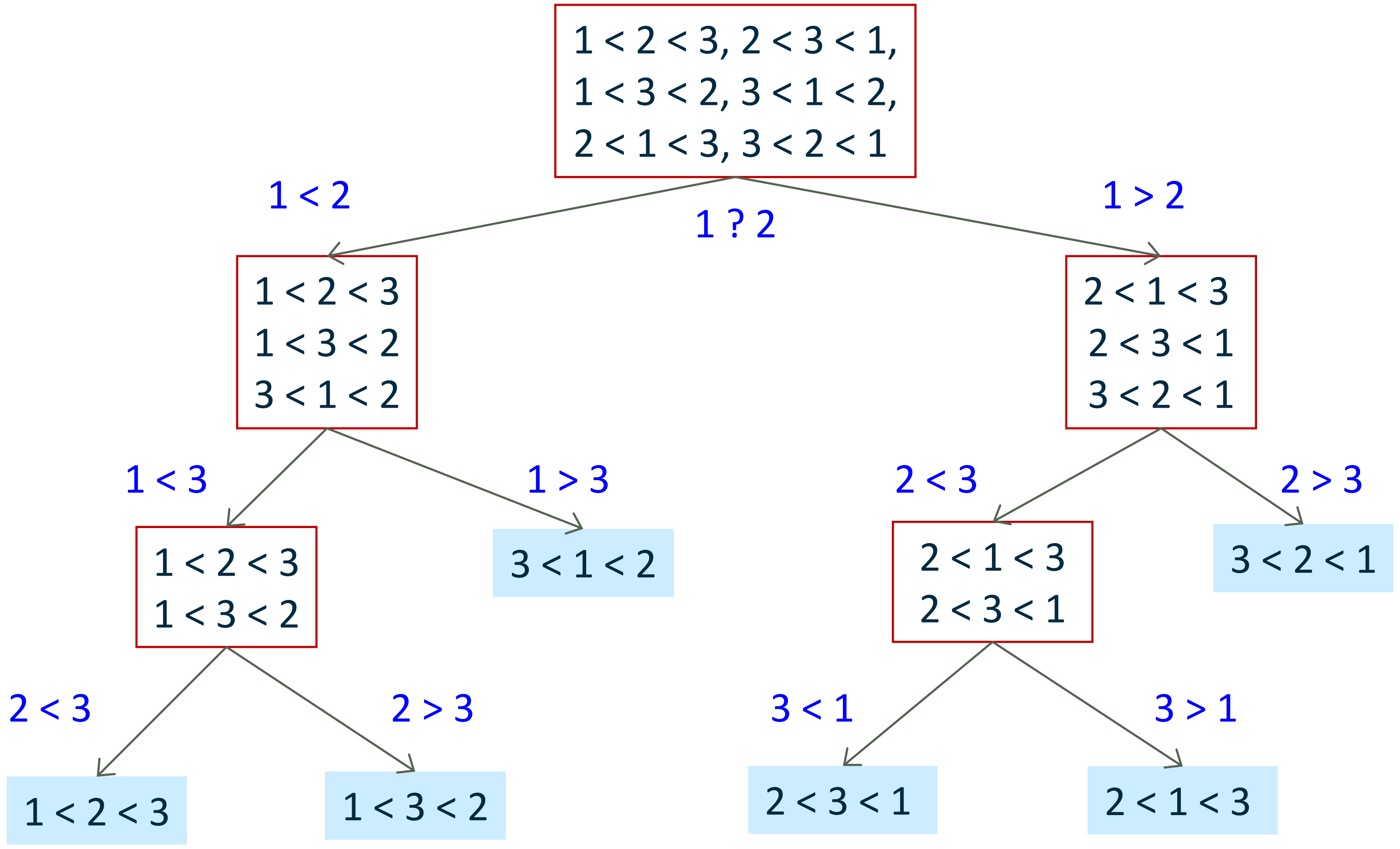
Decision tree for SelectionSort on 3-element input (hopefully. . . :-)



Sorting the input sequence <a_1 = 6; a_2 = 8; a_3 = 5>; the permutation <3, 1, 2> at the leaf indicates that the sorted ordering.

There are 3! = 6 possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

Decision Tree for n = 3



The leaves contain all the possible orderings of 1, 2, 3

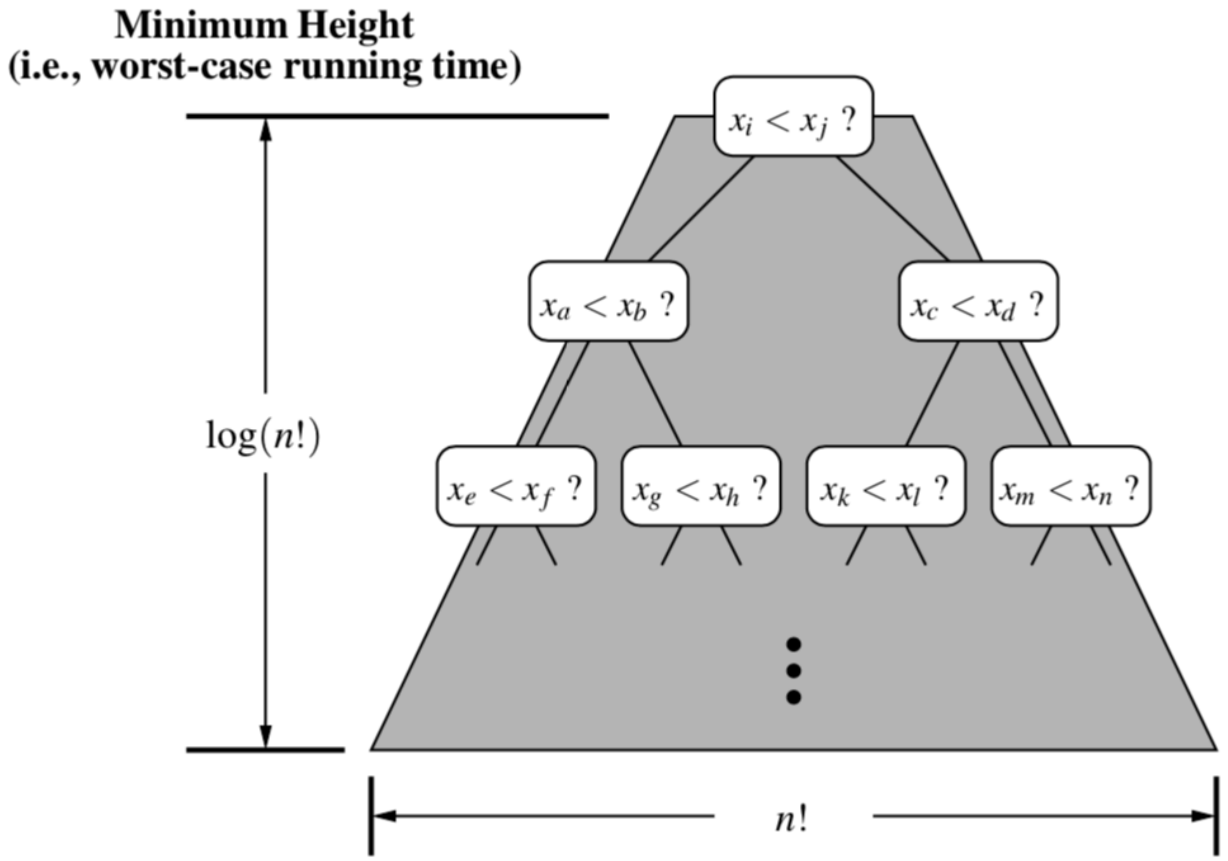
Any correct sorting algorithm must be able to produce each permutation of input (or: must sort any permutation of e.g. {1,...,n})

- necessary condition is that each of the n! permutations must appear as leaf of decision tree

Lower bound for worst case

Length of longest path from root of decision tree to any leaf (height) represents worst case number of comparisons for given value of n

For given *n*, lower bound on heights of **all** decision trees where each permutation appears as leaf is thus lower bound on running time of **any** comparison based sort algorithm



Theorem

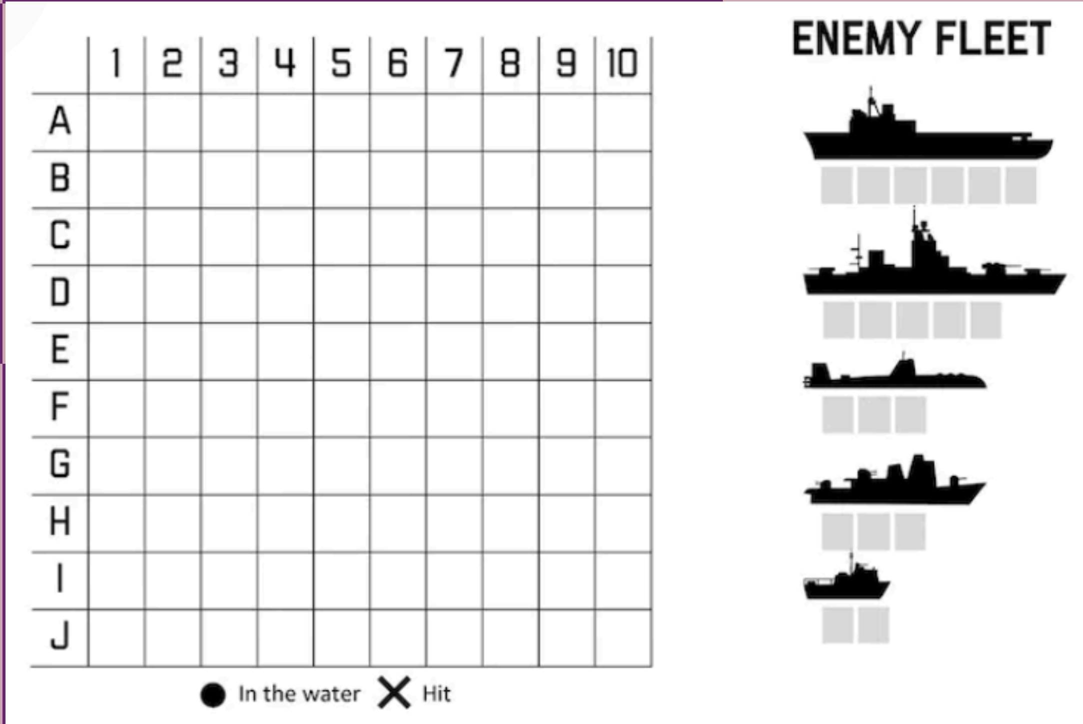
Any comparison-based sorting algorithm requires $\Omega(n \log n)$ *comparisons in the worst case.*

Proof:

- Sufficient to determine minimum height of a decision tree in which each permutation appears as leaf
- Consider decision tree of height h with ℓ leaves corresponding to a comparison sort on n elements
- Each of the n! permutations of input appears as some leaf: $\ell \geq n!$
- Binary tree of height h has at most 2^h leaves: $\ell \leq 2^h$
- Together: $n! \leq \ell \leq 2^h$ and therefore, $n! \leq 2^h$ (or $2^h \geq n!$)
- Takes log: $h \geq \log(n!) = \Omega(n \log n)$

Durham University

Selection and adversaries



Lower bounds via adversaries

Adversary:

- is a second algorithm intercepting access to input
- gives answers so that there’s always a consistent input

  i.e. adversary can never be caught cheating
- ensures that original algorithm has **not enough info** to make a decision, for **as long as possible**

  alternative view: dynamically constructs a bad input for it
- doesn’t know what original algorithm will do in the future

  i.e. must work for **any original algorithm**

To get a good lower bound, design a good adversary

Finding max

Goal: find max element in an unsorted array

In an array of size n, can do this with n − 1 comparisons

Same set-up as before: Only **comparisons** are relevant

Theorem

Any comparison-based algorithm for this problem requires at least $n-1$ comparisons in the worst case.

Proof.

After $\leq n-2$ comparisons, ≥ 2 elements never lost (a comp)
$\Rightarrow$ Adv can make any of them max and be consistent
$\Rightarrow$ not enough info for Alg to make a decision.
Hence Alg needs to make at least $n-1$ comparisons. □

Durham University

Designing Adversary's strategy

The only relevant info for Algorithm is

how many elements lost the (≥ 1) comparison.

As soon as n - 1 elements lost (at least once), the algorithm can make a decision - max is the one that has never lost.

Assume each index i has a status, N (never lost) or L (lost), initially all indices have status N.

After each comparison, there can be status changes N → L

Adversary's goal: delay changes N → L as much as possible

Durham
University

Designing Adversary’s strategy

Adversary’s strategy can be represented as a status update table (below), with the number of bits of new relevant info.

When Algorithm asks to compare *i : j*, reply as follows:

Status for i, j	Adv Reply	New Status	New Info
N;N	$a_i > a_j$	N;L	1
N;L	$a_i > a_j$	No Change	0
L,N	$a_i < a_j$	No Change	0
L;L	Consistent	No Change	0

One comparison gives ≤ 1 bit of new relevant info (i.e. new L).

In total, Algorithm needs n - 1 bits of info to make a decision.

Hence, ≥ n - 1 comparisons are necessary.

Durham University

Finding 2nd largest

Goal: find the 2nd largest element in an unsorted array

Theorem

Any comparison-based algorithm for this problem requires at least $n + \lceil \log_2 n \rceil - 2$ comparisons in the worst case.

Observations:

1. Any correct algorithm must also determine the max.

   n - 1 elements must lose 1 comparisons. (What if not?)

2. The 2nd largest must lose to max directly.
3. If max had direct comparisons with m elements, then m 1 of these elements must lose 2 comparisons.
4. Hence, at least n – 1 + m – 1 = n + m – 2 comparisons are required.

Durham University

Adversary's strategy

Enough to prove the following:

Lemma

Adversary can force $\lceil \log_2 n \rceil$ comparisons involving max.

Proof:

Adversary assigns weight w_i to each input element a_i .

Initially all $w_i = 1$, then update using these rules:

Weights	Adv Reply	Update
$w_i > w_j$	$a_i > a_j$	$w_i = w_i + w_j, w_j = 0$
$w_i < w_j$	$a_i < a_j$	$w_j = w_i + w_j, w_i = 0$
$w_i = w_j > 0$	$a_i > a_j$	$w_i = w_i + w_j, w_j = 0$
$w_i = w_j = 0$	Consistent	No change

Analysis

- $w_i = 0 \Leftrightarrow$ *i* lost ≥ 1 comparison
- Weight of an element at most doubles after a comparison
- If max is involved in *m* comparisons, its weight is $\leq 2^m$
- In the end, max accumulates all the weight, so $n \leq 2^m$.
- Taking logs, $\log_2 n \leq m$, as required.

So, Adversary can force $\lceil \log_2 n \rceil$ comparisons with max, which proves the lemma and hence the theorem.

Finding 2nd largest: Algorithm

Theorem

There is a comparison-based algorithm for finding 2nd largest that requires $n + \lceil \log_2 n \rceil - 2$ *comparisons in the worst case.*

Durham University

Finding 2nd largest: Algorithm

Theorem

There is a comparison-based algorithm for finding 2nd largest that requires $n + \lceil \log_2 n \rceil - 2$ comparisons in the worst case.

Here's an algorithm that matches our lower bound.

Consider a knock-out tournament: players = array elements

Think a balanced binary tree, leaves = array elements.

With an array of size n, the height of the tree is $[\log_2 n]$.

Play the tournament to find max: this takes $n - 1$ comparisons

Consider the $[\log_2 n]$ elements who lost directly to max.

2nd largest overall = largest among those.

Find it with $[\log_2 n] - 1$ comparisons.

Altogether, this uses $n + [\log_2 n] - 2$ comparisons.

Finding kth largest

Let V(n,k) be the number of comparisons necessary and sufficient to find kth largest in any array of size n.

The exact value of V(n,k) is known for

k = 1, 2, n 1, n.

For other values of k, only some bounds are known, e.g. :

Theorem

For all $1 \leq k \leq n$, *it holds that* $V(n,k) \geq n - k + \lceil \log_2 \binom{n}{k-1} \rceil$.

Durham University

Thank you