Algorithm Design and Analysis

Concrete Models and Lower Bounds

Roadmap for today

- Formal models of computation
  - We will work predominantly with the ***comparison model***
- Lower bounds for finding the maximum element in an array
  - Introducing two techniques: the ***adversary*** technique and the ***decision tree***
- A lower bound for sorting in the comparison model
  - Introducing the ***information-theoretic lower bound*** technique
- Another example of a lower bound in the comparison model
  - Showcasing a common trick: Finding a ***hard subset of inputs*** then using combinatorics to ***count the number of required outputs***, then applying the ***information-theoretic*** lower bound

Formal models of computation

- When theoretically analyzing algorithms, we don't consider their performance on a particular piece of hardware
  - E.g., how fast is this algorithm on an i9-14900K with DDR5 RAM? Who cares :)
- Instead, we define a **model of computation** which specifies:
  - Exactly what operations are permitted
  - How much each operation costs
- E.g., a *Turing Machine* is a model of computation
  - Allowed operations: Read/write/move tape
  - Cost model: all operations cost 1

What is the best model?

- No such thing… **it depends**
- It depends on the setting. Are you designing a single-threaded algorithm, a parallel algorithm, an algorithm for GPUs, an algorithm that will work on a gigantic dataset…
- It also depends on your goal. Are you trying to predict the performance of an algorithm in a particular scenario or are you trying to prove a ***lower bound***?

Today's models

- ***The Comparison Model*** (as seen in Lecture 1)
  - Input to the algorithm consists of an array of n items in some order
  - The algorithm may perform comparisons (is $a_i < a_j$?) at a cost of 1
  - Copying/moving items is *free*
  - The items are of an arbitrary type. **We are not allowed to assume a type**
    - E.g., the items **can not** be assumed to be numbers
    - This means we can not add, multiply, XOR the items
    - We also can not use hashing, or use elements as array indices, etc.

Today's goals

- Devise *lower bounds* for problems, i.e., prove that certain problems can not be solved in under a certain cost.

Definition (Lower bound): If we say that a specific problem on inputs of size n has a *lower bound* of $g(n)$, we mean that for **any algorithm** **A** that solves the problem, **there exists some input** of size n for which the cost of **A** is **at least** $g(n)$.

Note: A lower bound **does not** mean that **every input** requires cost at least $g(n)$, only that **at least one** input does. In other words, it means the **worst-case cost** is at least $g(n)$, but the best-case could be cheaper.

Select-max

Problem: Given an array of n elements, return the maximum element.

Algorithm: Scan left-to-right keeping track of the maximum so far

Cost: $n - 1$ comparisons

Question: How few comparisons could any algorithm possibly do? Is it possible to do fewer than $n - 1$?

Weak lower bound

Theorem: Any deterministic algorithm for select-max costs **at least** $n/2$ comparisons

Proof:

a_i not touched

then we can force the algorithm to give the wrong answer.

Stronger lower bound

Theorem: Any deterministic algorithm for select-max costs **at least** $n-1$ comparisons

Proof:

Graph with an edge for each comparison

raise by $+\infty$

$\Rightarrow$ connected $\Rightarrow$ have $\geq n-1$ edges

Adversary arguments

- We proved the lower bound using an ***adversary argument***
- Given any algorithm that performs “too few” comparisons, we argued that we can always construct an input on which it must give the wrong answer.
- We are playing the role of an *adversary* trying to “break” the algorithm!
- Remember that our argument must break ***every algorithm*** that we are trying to rule out, we can not assume a specific algorithm.

Another technique: Decision Trees

- Consider the set of all possible outputs. Before the algorithm makes any comparisons, they all could be the answer.
- After each comparison, some of the possibilities are ruled out

Possible answers:

$\max = a_1$
$\max = a_2$
$\max = a_3$

Comparison performed:

$(a_2 < a_1)$ == TRUE

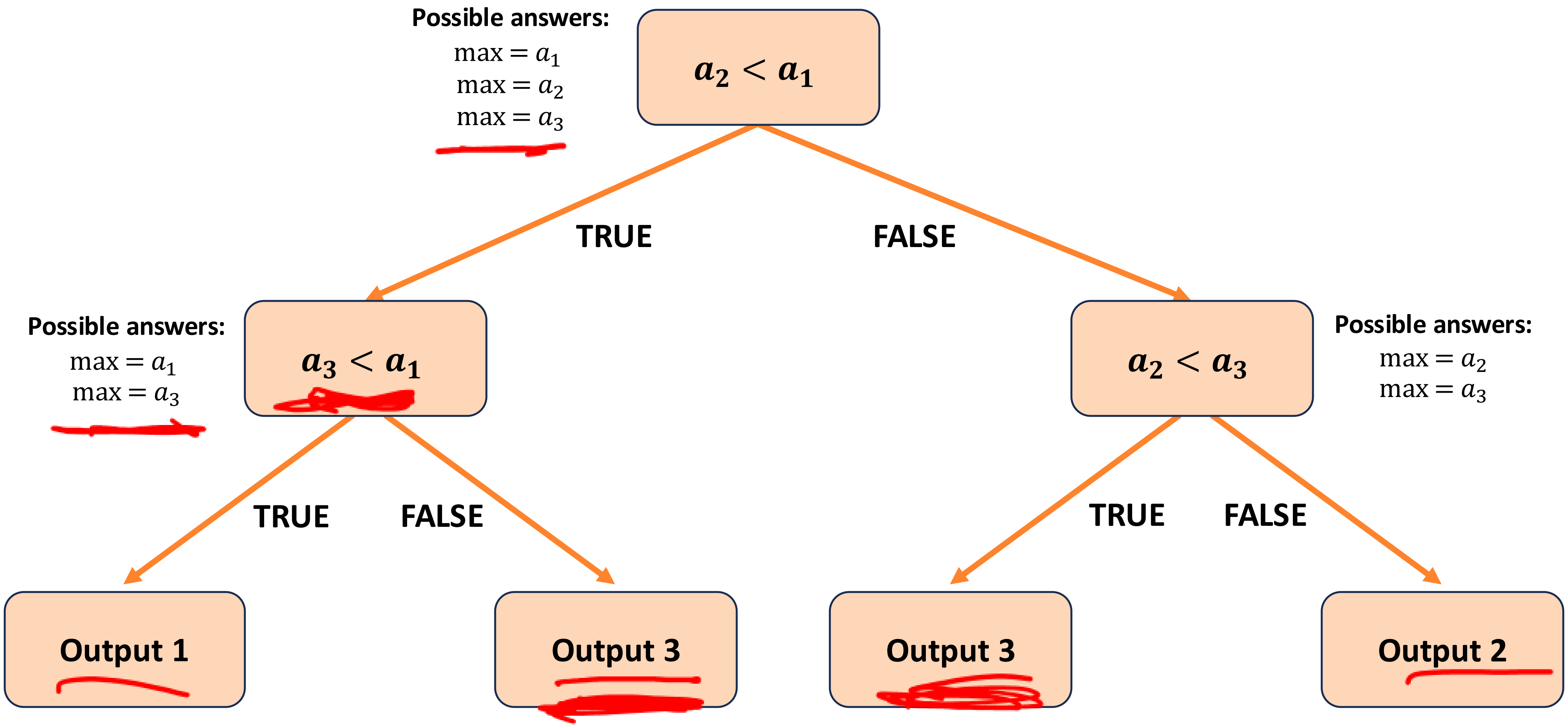
→

Possible answers:

$\max = a_1$
$\max = a_3$

- We can represent any ***specific*** comparison-based algorithm as a ***decision tree***.

Example decision tree



Decision tree implications

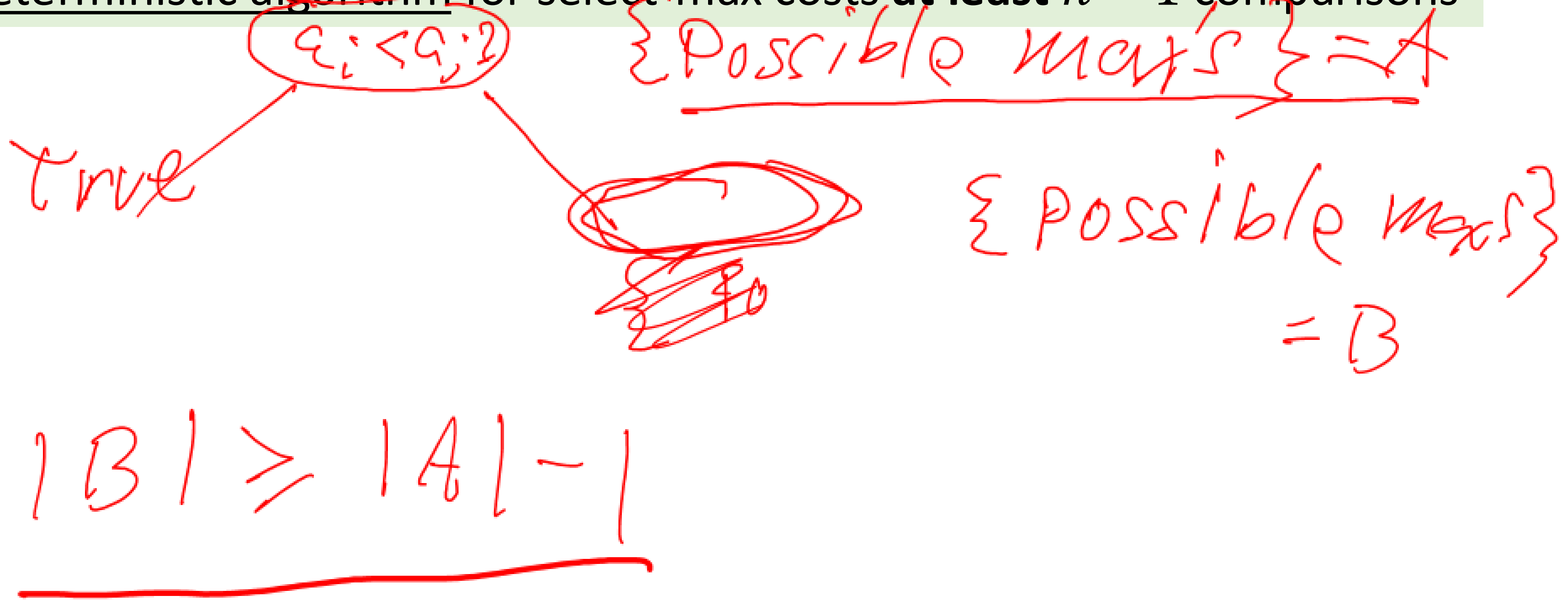
- The cost of an algorithm on a particular input is the depth of the leaf node that that input leads to in the tree
- Therefore, the **worst-case** case of the algorithm is… Longest Path from root to a leaf

- Remember that a particular decision tree corresponds to a particular algorithm (its just a way of writing down the algorithm as an alternative to pseudocode or plain English)
- A lower-bound proof using decision trees must therefore argue that ***every possible decision tree*** for the problem has ***at least a certain height***

Proof via decision trees

Theorem: Any deterministic algorithm for select-max costs **at least** $n - 1$ comparisons

Proof:



Question break

Sorting in the comparison model

- The comparison model is widely used to analyze sorting algorithms
  - You don't get to assume that the data are integers, or numbers, so the algorithms will be extremely general. They can sort anything!

- We know how to achieve $O(n \log n)$ comparisons: Quicksort (deterministic from Lecture 1), Mergesort, Heapsort.

- Can we do better?

Input/output of comparison sorting

- Simplify by assuming that all the elements are distinct (no duplicates)
- The *input* is an array of elements in some initial order

$$a_1, a_2, a_3, \dots, a_n$$

- The *output* is a permutation of the input elements in sorted order

$$a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$$

Warning!: Defining the “output” of a comparison sort is extremely subtle if we want to correctly prove lower bounds. We must be very careful.

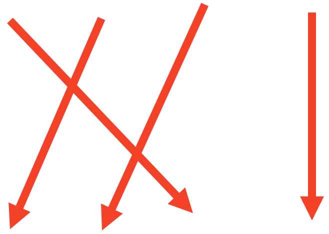
Understanding "output"

- Suppose we ask an algorithm to sort $[c, a, b, d]$ and $[b, d, a, c]$
- These both sort to $[a, b, c, d]$. Are these **the same** output?

No

c a b d

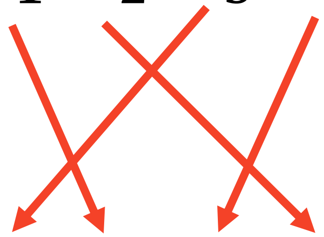
Input: $[a_1, a_2, a_3, a_4]$



Output: $[a_2, a_3, a_1, a_4]$

b d a c

Input: $[a_1, a_2, a_3, a_4]$

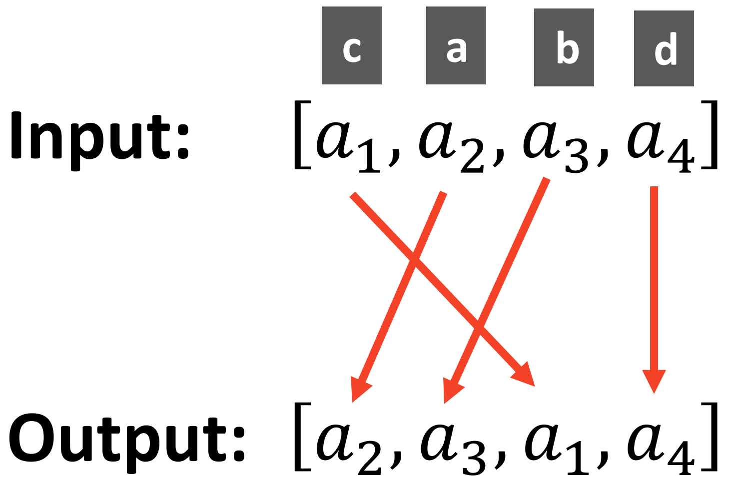


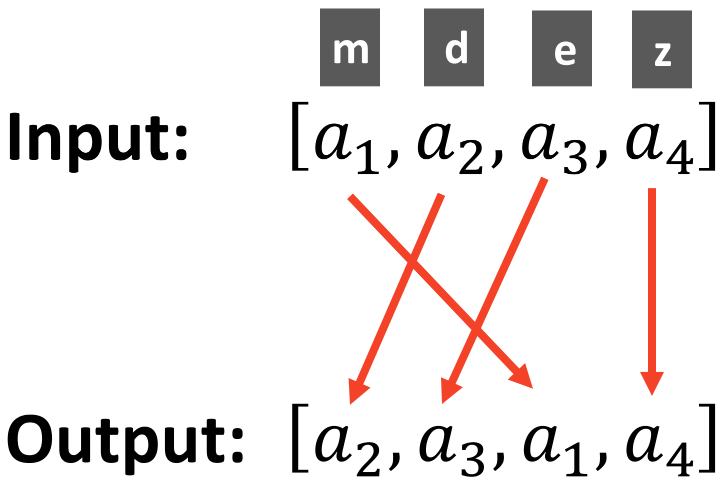
Output: $[a_3, a_1, a_4, a_2]$

Understanding "output"

- Suppose we ask an algorithm to sort $[c, a, b, d]$ and $[m, d, e, z]$
- These sort to $[a, b, c, d]$ and $[d, e, m, z]$. Are these **the same** output?

Yes





Sorting lower bound

Theorem: Any deterministic comparison sorting algorithm must perform at least $\log_2(n!)$ comparisons in the worst case.

- Different technique this time. Instead of an adversary, we are going to use ***information theory***
- This critically relies on how we define the input/output
- Remember that we must prove this fact for **every possible algorithm**, not just one.

Proof

- Remember, the algorithm is ***deterministic!*** Its behaviour is determined **entirely** by the results of the comparisons.

- If a deterministic algorithm makes ***c*** **comparisons**, how many ***distinct outputs*** can it possibly produce?

2^c

Proof

$a_i \neq a_j$ {all $n!$ Perms}

- How many distinct ***inputs*** consisting of n unique elements does the algorithm need to be able to sort?

$n!$

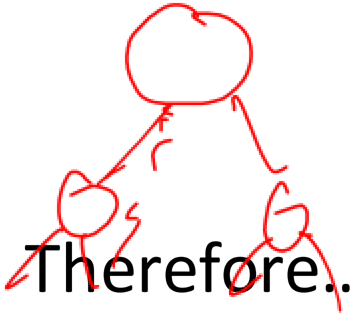
- Can two distinct ***inputs*** consisting of unique elements ever be sorted by the same output?

No

Proof

- Therefore, a correct sorting algorithm for sorting n unique elements must be capable of producing how many distinct outputs?

$n!$



- Therefore…

$n! \leq \#\text{leaves} \leq 2^c$

$n! \leq 2^c \Rightarrow c \geq \log_2 n!$

What on earth is $\log_2(n!)$

- **A loose bound:**

$$\log_2 n! = \log_2 n + \log_2(n-1) + \cdots + \log_2(1)$$

$$\frac{n}{2}\log\frac{n}{2} < \log_2 n + \log_2(n-1) + \cdots + \log_2(1) < n\log n$$

- **Tighter bounds (Stirling's approximation):**

$$\log_2(n!) = n\log_2 n - n\log_2 e + O(\log_2 n)$$

$$\left(\frac{n}{e}\right)^n \leq n! \leq n^n$$ ← **Very useful!**

Another example

Problem (Sorting D distinct items): Consider the problem of sorting an array of n items, but we are guaranteed that there are at most D distinct elements (where $1 \leq D \leq n$), i.e., the array may contain many duplicates.

Intuition check: Do we expect this to be more expensive or cheaper to solve than the previous problem?

cheaper

I.e. ∀ algo ∃ a sequence with at most D dist. elts, That does $\Omega(n \log D)$ comps.

Theorem: Any deterministic comparison sorting algorithm on n items where there are at most D distinct elements requires $\Omega(n\log(D))$ comparisons in the worst case.

Another example

- How many outputs does a correct algorithm need to be able to produce to solve this problem?
  - This seems **much** harder to reason about than the first problem, where we had $n!$ distinct inputs each requiring a distinct output

$\boldsymbol{a\ a\ b}$

$\boldsymbol{a\ b\ b}$

both sorted by $[a_1, a_2, a_3]$ → **Number of required outputs $\neq$ number of possible inputs**

Useful observation: Suppose we focus on just a **subset of possible inputs** to the problem and prove a lower bound on the cost of solving inputs from that set. Then this lower bound applies to the entire problem.

Picking a good set of inputs

So, we want to pick a set of inputs that:

- ***Requires a lot of outputs***. To use the information-theoretic lower bound, we want to show that lots of outputs are required.
  - Usually, we will do this by **counting the number of inputs** in the set and then **arguing about the relationship between the number of inputs and output**
  - Often (but not always) we will argue that each input requires a distinct output, so the number of inputs lower bounds the number of required outputs
- ***Is simple enough that we can count the number of required outputs.***
  - We will try to describe a set of inputs that has some nice combinatorial structure so we can count it using counting techniques from concepts

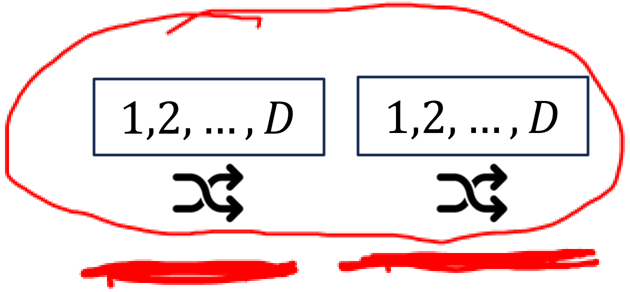
Picking a good set of inputs

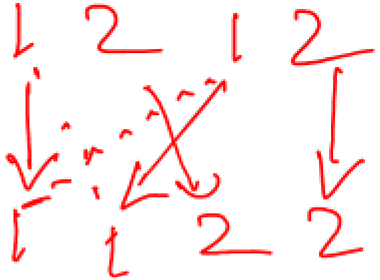
- We need to describe a family of inputs on n elements where there are at most D distinct elements.
- **Goals**:
  - ***Simple to describe and count***
  - Requires a ***distinct output*** for each input

Remember: A permutation on a list of distinct elements has a unique inverse (i.e., a unique output that sorts it)

Arguing distinctness

- Suppose I take two permutations on $1,2,\ldots,D$ (i.e., one copy of each distinct input element)





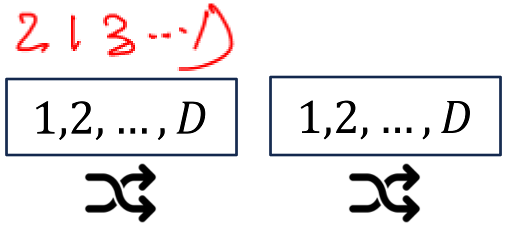
- Is there a unique output (permutation) that sorts one of these? NO

- Can a single output (permutation) sort two of these? NO

Proof: Since the elements in each half are distinct, if I swap any of them, it goes to the wrong place in the output permutation

Arguing distinctness

- Suppose I take two permutations on $1, 2, \ldots, D$ (i.e., one copy of each distinct input element)

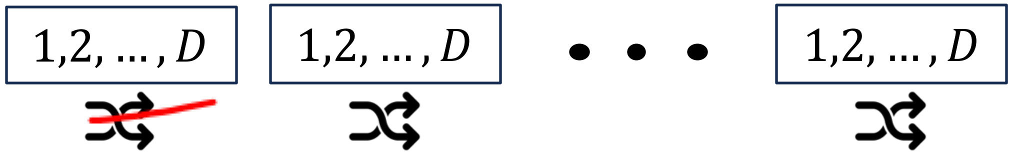


- Can a single output (permutation) sort two of these? **NO**

Number of required outputs $=$
number of possible inputs

Constructing our input family

- We will construct a family of inputs by concatenating n/D many permutations of $1, \dots, D$



- By our previous slide, each of these requires a different output to sort, so **number of required outputs = number of inputs in this family**

$(D!)^{n/D}$

The lower bound

- This family contains $(D!)^{\frac{n}{D}}$ inputs, each requiring a different output, so sorting everything in this family requires $(D!)^{\frac{n}{D}}$ outputs
- Our information-theoretic lower bound argument therefore gives us a lower bound of…

$$2^C \geq (D!)^{n/D}$$

$$C \geq \frac{n}{D}\log D! \simeq \frac{n}{D} D \log D$$

$$\simeq n \log D$$

Summary of lower-bound techniques

- ***Adversary***: Show that you can construct an input to "break" the algorithm if it performs too few comparisons
- ***Decision Tree***: Model any algorithm for the problem as a binary tree of possible outputs and lower bound the height of the tree
- ***Information-theoretic***: Count the minimum number of necessary distinct outputs that the algorithm must be able to produce
  - Sometimes we need to find a hard subset of the input and show a lower bound on that, since we can't figure out how to count the entire output set

Important result: Sorting requires $\log_2 n! = \Theta(n \log n)$ comparisons.