



排序 (续)

Sorting Cont'd

钮鑫涛

Nanjing University

2023 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!



About the sorting itself

- We have learned many sorting algorithms
 - ▶ Bubble $\Theta(n^2)$, Selection $\Theta(n^2)$, Insertion $\Theta(n^2)$, Heap $\Theta(n \log n)$, Merge $\Theta(n \log n)$, Quick $\Theta(n \log n)$
 - ▶ One may ask: can we have an algorithm with complexity smaller than $\Theta(n \log n)$?



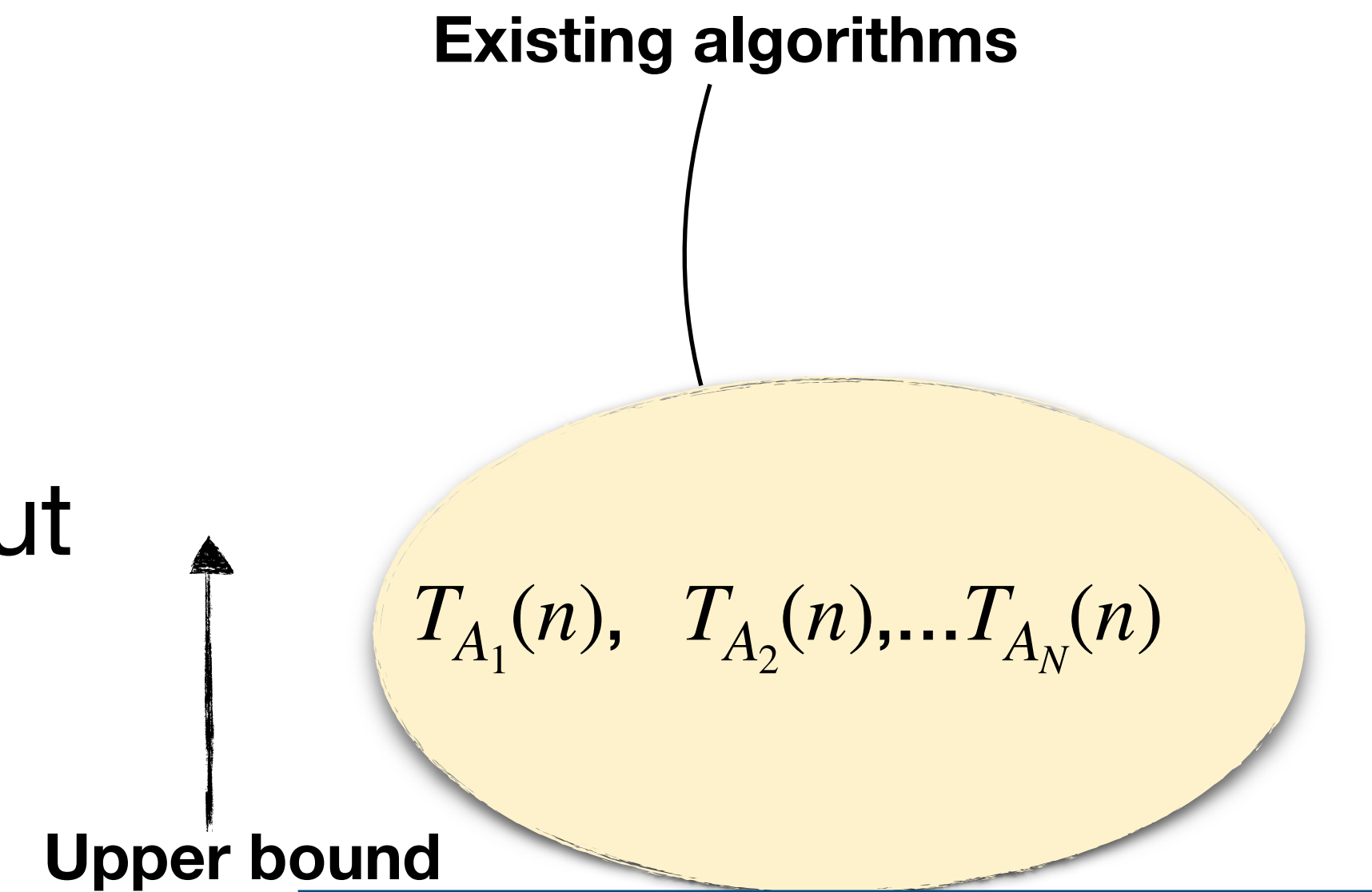
Complexity of a problem

- We have learned how to analyze the complexity of **one specific algorithm**.
- For a problem, e.g., the sorting problem, what is the complexity of it?
 - The complexity of the **best** algorithms that allow solving the problem.
- The study of the complexity of explicitly given algorithms is called ***analysis of algorithms***, while the study of the complexity of problems is called ***computational complexity theory***.



Upper bound and Lower bound

- Consider a problem P .
 - ▶ **Upper bound**: how **fast can** we solve the problem?
 - The (worst-case) runtime of an algorithm A on input of size n is: $T_A(n) = \max_{|I|=n} \{\text{cost}_A(I)\}$
 - $T_A(n)$ **upper bounds** the complexity of solving problem P .
 - Every valid algorithm gives an upper bound on the complexity of P .





Upper bound and Lower bound

- Consider a problem P .
 - **Lower bound:** how **slow** solving the problem **has to be**?
 - The worst-case complexity of P is the worst-case runtime of the **fastest** algorithm that solves P :

$$T_P(n) = \min_{A \text{ solves } P} \left\{ \max_{|I|=n} \{ \text{cost}_A(I) \} \right\}$$

- $T_P(n)$, usually in the form of $\Omega(f(n))$, **lower bounds** the complexity of solving problem P .
- $T_P(n) = \Omega(f(n))$ means **any** algorithm has to spend $\Omega(f(n))$ time to solve problem P .

Lower bound



$T_{A_{min}}(n)$

The fastest possible algorithm, what is it?



Lower bound of a problem

- A **lower bound** of $\Omega(f(n))$ for a problem P means **any** algorithm that solves P has **worst-case** runtime $\Omega(f(n))$.
- Larger lower bound is a **stronger** lower bound. (On the other hand, smaller upper bound is better.)
- But how do we prove a lower bound?!
 - It is usually unpractical to examine all possible algorithms...
 - Instead, rely on structures/properties of the problem itself...



Trivial Lower Bounds

- Lower Bound based on **Output** Size
 - ▶ Any algorithm that for inputs of size n has a worst-case output size of $f(n)$ needs to have a runtime of $\Theta(f(n))$.
 - ▶ This is because it has to output all the $f(n)$ elements of the output in the worst-case.
 - ▶ E.g., an algorithm generating all the permutations of an array must cost at least $\Omega(n!)$



Trivial Lower Bounds

- Lower Bound argument based on **Input** Size (Incorrect)
 - ▶ Since the algorithm "has" to read its entire input, any algorithm for inputs of size n runs in $\Omega(n)$ time?
 - ▶ The argument above is that it is **false**.
 - *Searching a sorted list*: Given n numbers a_1, a_2, \dots, a_n in sorted order, and a number v , check if there exists i such that $a_i = v$.
 - We do not need to scan all the inputs by using *binary-search* algorithm



Trivial Lower Bounds

- The problem with trivial techniques is that it often suggests a lower bound that it is too low.
- For example, the trivial lower bound for the complexity of comparison based sorts is $\Omega(n)$, because the algorithm must **output** the results. This bound is too low.
 - Note: the larger the lower bound, the more useful it is.



Adversary Argument





Get a tighter lower bound

- Recall the lower-bound: $T_P(n) = \min_{A \text{ solves } P} \left\{ \max_{|I|=n} \{ \text{cost}_A(I) \} \right\}$
- The key is to design the **worst input** for an algorithm, and this algorithm must **solve** it!
 - ▶ But how it possible to design a worst input in the case that we do not even know the algorithm?
 - ▶ We first have to specify *precisely* what **kinds** of algorithms we will consider by using: **key operations!**
 - ▶ And then devise an **adversary strategy** to construct a worst case input!



Key operation

- A **key operation** is a step that is representative of the computation overall.
- Properties of key operations:
 - ▶ Can be constant-time operations.
 - ▶ Represent or dominate other operations.
 - ▶ Number of key operations should give a function of the input size.
- A model of computation might be designed around key operations (other computation are omitted);



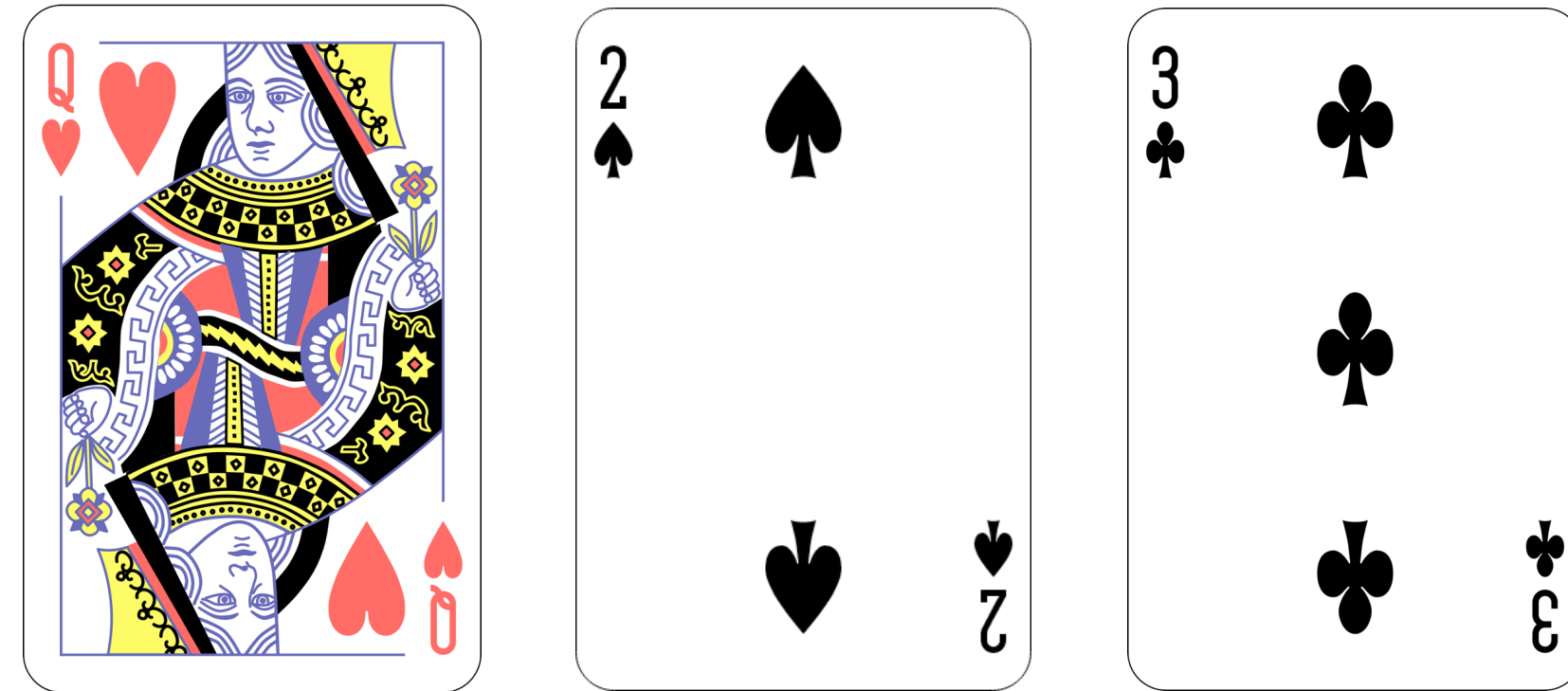
Key operation

- Examples of key operations:
 - ▶ Access one data item (e.g. query the i^{th} value in an array)
 - ▶ Compare two categorical items (outcome = or \neq)
 - ▶ Compare two ordinal items (outcome =, <, or >)
 - ▶ Determine if two vertices in a graph are adjacent



Adversary strategy example: Three-Card Monte

- The dealer show the tourist three cards, say:



- The dealer shuffles the cards face down on a table (usually slowly enough that the tourist can follow the Queen), and then asks the tourist to bet on which card is the **Queen**.

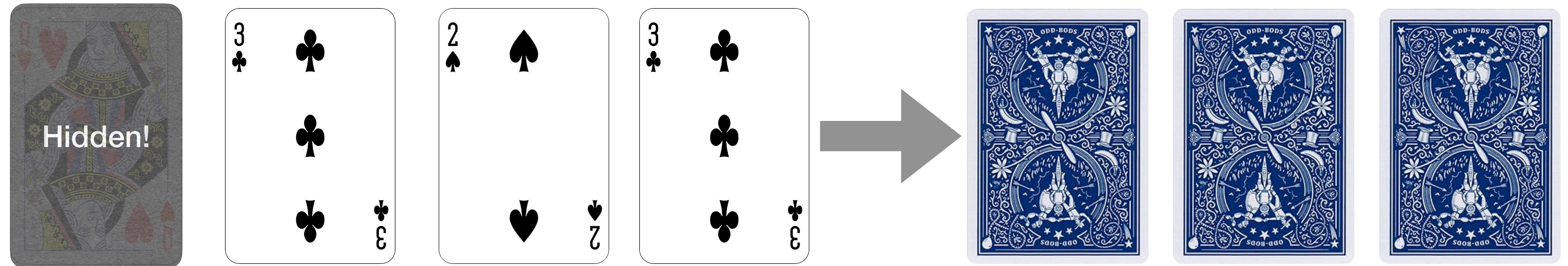


In principle, the tourist's chances of winning are at least $1/3$, more if the tourist was carefully watching the movement of the cards!



Three-Card Monte

- In practice, however, the tourist *never* wins, because the dealer *cheats*.
- The dealer actually holds at least four cards; before he even starts shuffling the cards, the dealer palms the queen or sticks it up his sleeve

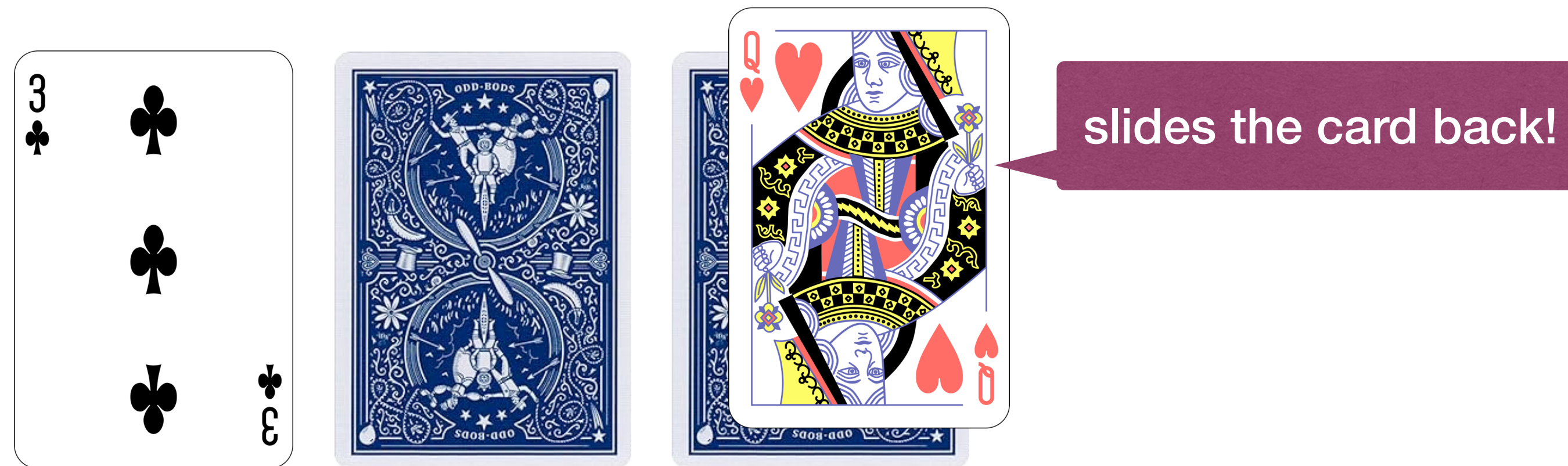


- No matter what card the tourist bets on, the dealer turns over a black card



Three-Card Monte

- After the tourist giving up, the dealer slides the queen under one of the cards and turns it over, showing the tourist ‘where the queen was all along’



- As long as the dealer doesn't reveal all the black cards at once, the tourist has no way to prove that the dealer cheated!




n -Card Monte

- Now let's consider a similar game, but with an algorithm acting as the tourist and with bits instead of cards.
- Suppose we have an array of n bits and we want to determine if **any** of them is a 1.
- Obviously we can figure this out by just **looking at** every bit, but can we do better? That is, without looking at every bit.

Key operation



Adversary argument

- Basic idea: 
 - ▶ An all-powerful malicious adversary pretends to choose an input for the algorithm.
 - ▶ When the algorithm wants checks a bit, the adversary sets that bit to whatever value will make the algorithm **do the most work**.
 - ▶ If the algorithm does not check enough bits before terminating, then there will be **several different** inputs, each **consistent with** the bits already checked, and should result in **different outputs**.
 - ▶ Whatever the algorithm outputs, the adversary can ‘**reveal**’ an input that is has all the examined bits but contradicts the algorithm’s **output**, and then claim that that was the input that he was using all along.



n -Card Monte

- For the n -card monte problem, the adversary originally pretends to choose an input array \rightarrow whenever the algorithm looks at a bit, it sees a 0 (This is the worst input since if it sees a 1, then algorithm can terminate immediately with a right answer).
- Now suppose the algorithms stops before looking at every bits.
 - ▶ If the algorithm says '**No, there's no 1,**' the adversary changes one of the unexamined bits to a 1 and shows the algorithm that it's wrong.
 - ▶ If the algorithm says '**Yes, there's a 1,**' the adversary reveals the array of zeros and again proves the algorithm wrong.



Some notes about the adversary strategy

- One absolutely crucial feature of this argument is that the adversary makes absolutely no assumptions about the algorithm.
- The adversary strategy can't depend on some predetermined order of examining bits, and it doesn't care about anything the algorithm might or might not do when it's not looking at bits.
- However, as long as there are **at least two possible answers** to the problem that are **consistent with all answers** given by the adversary, the algorithm cannot be done!



Adversary argument for Comparison-based sorting

- The comparison is widely used as the **key operation** to analyze sorting algorithms
 - ▶ Don't get to assume that the data are integers, or numbers. So the algorithm is more general!
 - ▶ The number of comparisons performed by a sorting algorithm usually (but not always) matches the asymptotic number of instructions performed



Setup for comparison sorting

- The input to the problem is n elements in some initial order.
- The algorithm knows nothing about the elements.
- The algorithm may **compare** two elements (“is $a_i > a_j$?”) at a cost of 1
 - Particularly, the algorithm cannot inspect the values of input items.
- Moving/copying/swapping items is free!
- Assume that the input contains no duplicates.



Set up of the adversary

- Notice that there are $n!$ different permutations (and thus solutions) that the sorting algorithm must decide between.
- The adversary maintains a list L of all of the permutations that are consistent with the comparisons that the algorithm has made so far.
- Initially L contains all $n!$ permutations.



The adversary's strategy

- The adversary's strategy for responding to "Is element i less than element j " is as follows:
 - ▶ Let L_{yes} be the permutations in L for which element i is less than element j , let L_{no} be the permutations in L for which element i is greater than or equal to element j . (Thus, $L = L_{yes} \cup L_{no}$).
 - ▶ Then the adversary responds "yes" exactly when $|L_{yes}| \geq |L_{no}|$, otherwise responds "no". In other words, the adversary answers in such a way to keep L as large as possible (the worst input construction).
 - ▶ Then the adversary updates L so that only those permutations consistent with this answer remain. So if "yes" is answered then the permutations in L_{no} are removed, and if "no" is answered the permutations in L_{yes} are removed.



The bound determined by the adversary's strategy

- Since **at least half** of the permutations in L remain, and the algorithm **cannot be done** until $|L| = 1$, the number of comparisons required is at least $\lceil \lg(n!) \rceil$.
- Therefore, the lower bound of comparison-based sorting is $\Omega(\lg(n!)) = \Omega(n \lg n)$



Information-Theoretic Arguments



The amount of the information

- Consider the minimum number M of **distinct outputs** that a sorting algorithm must be able to produce to be able to sort any possible input of length n .
 - $M = n!$
- Why minimum? We don't want redundant outputs that don't allow us to solve more inputs.
- In other words, the algorithm must be capable of outputting at least M different permutations, or there would exist some input that it was not capable of sorting.



The amount of the information

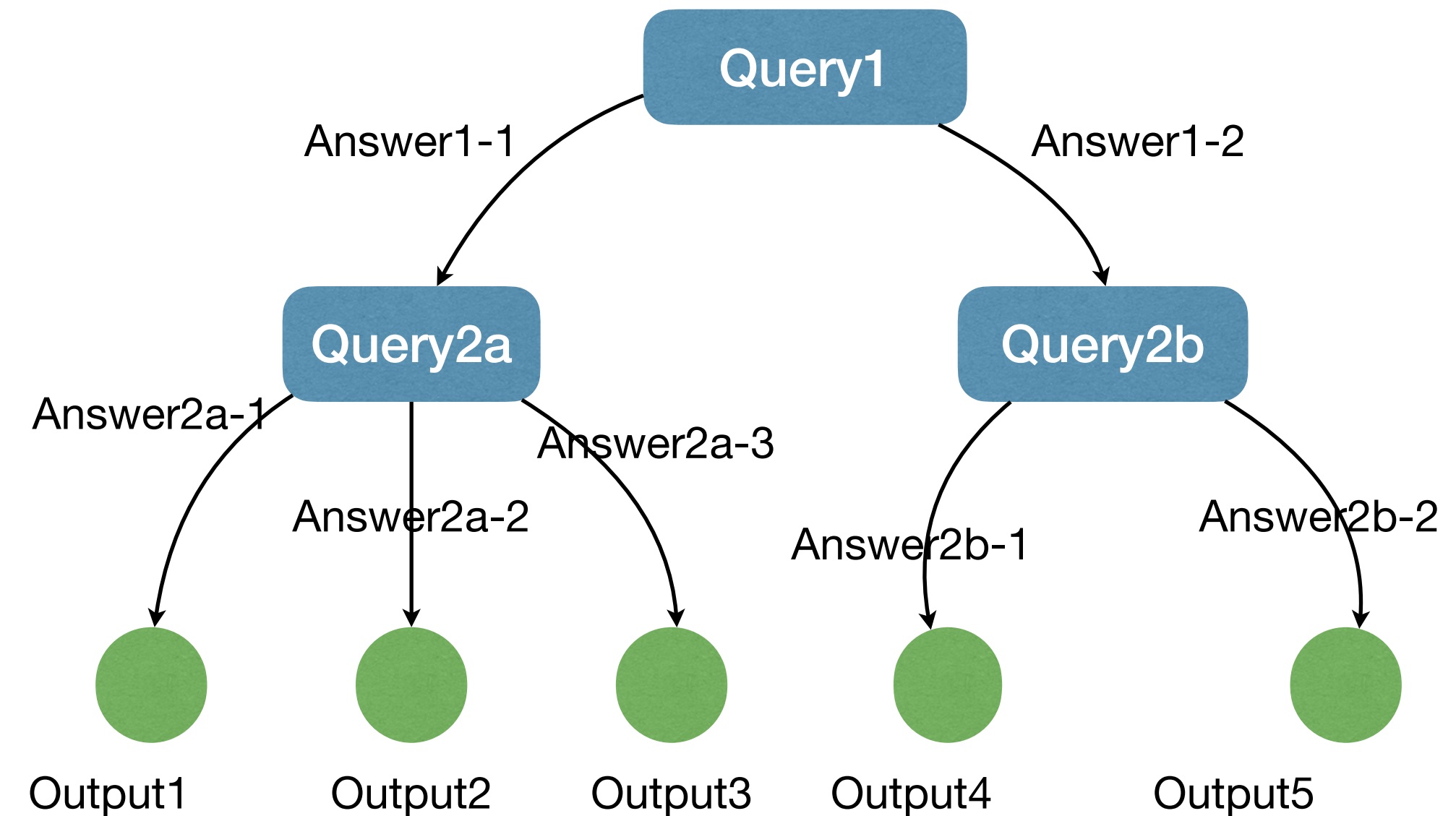
- Remember, the algorithm is deterministic and its behavior is determined **entirely** by the results of the comparisons.
- If a deterministic algorithm makes c comparisons, how many outputs distinct outputs can it possibly produce?
 - At most 2^c different possible outputs.
 - This is because one comparison can only has **two** different outcomes → **true** for $a_i > a_j$, otherwise **false**.
- Therefore, $2^c \geq n!$, and c is at least $\lg n!$



An alternative view: Decision Trees

- Decisions trees can be used to describe algorithms.
 - ▶ A decision tree is a tree.
 - ▶ Each internal node denotes a query the algorithm makes on input.
 - ▶ Outgoing edges denote the possible answers to that query.
 - ▶ Each leaf denotes an output.

Algorithm described in decision tree

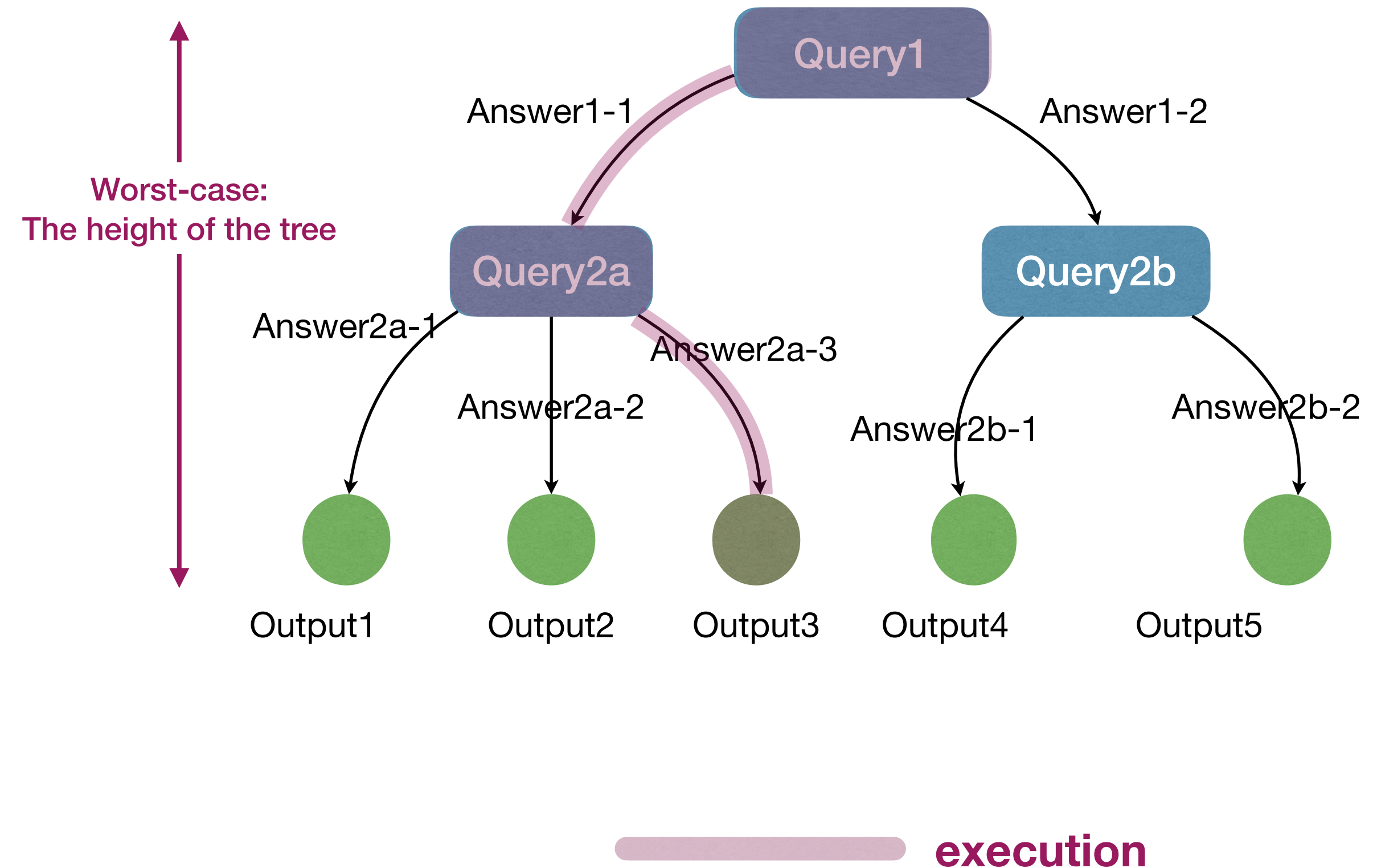




An alternative view: Decision Trees

- One execution of the algorithm is a path from root to a leaf.
 - At each internal node, answer to query tells us where to go next.
- The worst-case time complexity is at least the length of the longest path from root to some leaf, i.e., height of the tree!

Algorithm described in decision tree

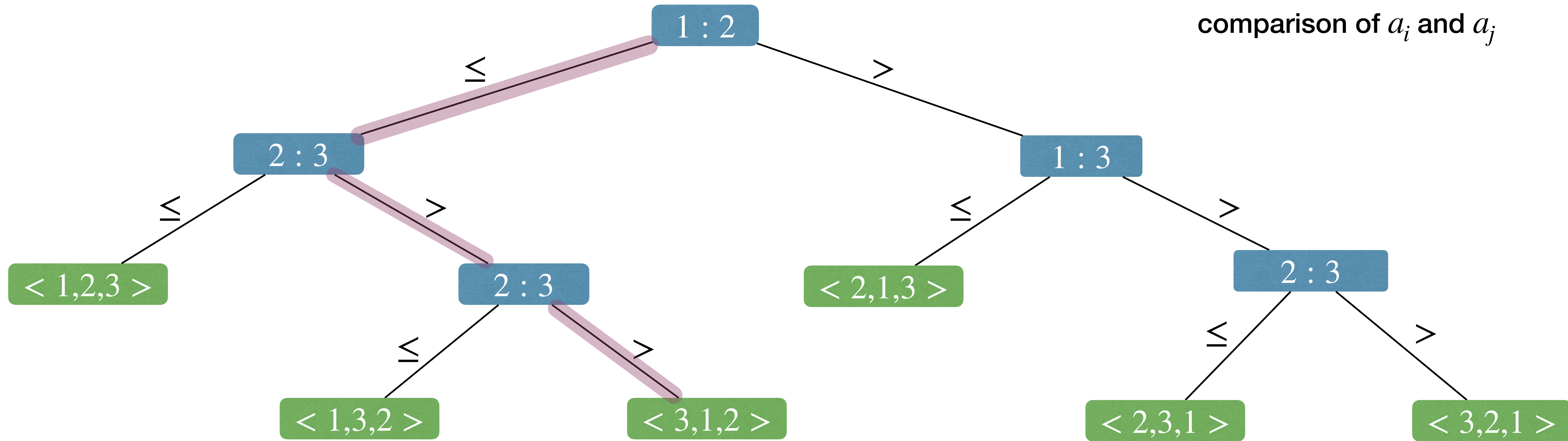




Decision Tree

Some algorithm Sorts $\langle a_1, a_2, a_3 \rangle$, where $a_1 = 6, a_2 = 8, a_3 = 5$

$i : j$ denotes the query for comparison of a_i and a_j





Comparison-based sorting lower bound

- Assume input items are distinct.
- Assume the algorithm only uses “ \leq ” to do comparison.
- We can use a **binary comparison tree** to describe the algorithm.
 - Each internal node has two outgoing edges.
 - Each internal node denotes a query of the form “ $a_i \leq a_j$ ”.
- The tree must have $\geq n!$ leaves.
- The height of the tree must be $\geq \lg(n!)$, which is $\Omega(n \lg n)$.

Any comparison-based sorting algorithm has time complexity $\Omega(n \lg n)$, in the worst case.

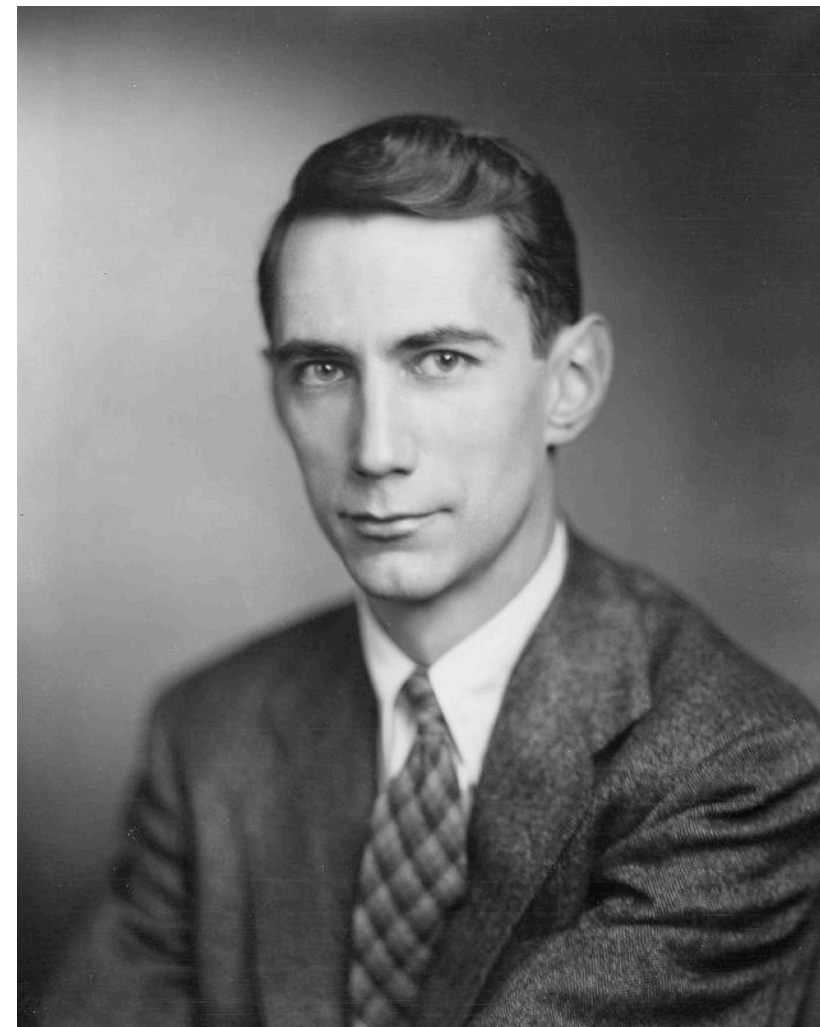


*Information content

- **Information content**, or **Self-information** or **Shannon information** of an event x :

- ▶ $I(x) := -\lg Pr(x)$, or equivalently, $I(x) := \lg \frac{1}{Pr(x)}$

- ▶ It measures the "informational value" of an event depending on its "surprising"
 - If a highly likely event occurs, it carries very little information. In fact, a 100% likely event occurs, it has no information!
 - On the other hand, if a highly unlikely event occurs, it is much more informative.



Claude Shannon



Information content

- For instance, the knowledge that some particular number *will not* be the winning number of a lottery provides very little information, because any particular chosen number will almost certainly not win.
- However, knowledge that a particular number *will* win a lottery has high informational value because it communicates the outcome of a very low probability event.



Information entropy

- Given a discrete random variable X , which takes values in the alphabet \mathcal{X} and is distributed according to $Pr : \mathcal{X} \rightarrow [0,1]$, the entropy of a random variable is the **average level** of "information", "surprise", or "uncertainty" inherent to the variable's possible outcomes.

$$\blacktriangleright H(X) := \sum_{x \in \mathcal{X}} -Pr(X = x) \lg Pr(X = x) = \mathbb{E}[-\lg Pr(X)]$$

- As an example, rolling a die has higher entropy than tossing a coin!

$$\blacktriangleright H(\text{🎲}) = \sum_{x \in [1,2,3,4,5,6]} -Pr(X = x) \lg Pr(X = x) = \sum_{x \in [1,2,3,4,5,6]} -1/6 \lg 1/6 \approx 2.58$$

$$\blacktriangleright H(\text{🌐}) = \sum_{x \in [0,1]} -Pr(X = x) \lg Pr(X = x) = \sum_{x \in [0,1]} -1/2 \lg 1/2 = 1$$



Information entropy

- Consider a coin with probability p of landing on heads and probability $1 - p$ of landing on tails.
 - ▶ The information for landing on head: $I(\text{landing on head}) = -\lg p$
 - The smaller the p , the larger information when landing on head!
 - ▶ The information for landing on tail: $I(\text{landing on tail}) = -\lg(1 - p)$
 - The smaller the $1 - p$, the larger information when landing on tail!
 - ▶ The entropy for tossing a coin: $-\left(\frac{1}{2} \lg p + \frac{1}{2} \lg(1 - p)\right) = -\frac{1}{2} \lg p(1 - p)$
 - The maximum surprise is when $p = 1/2$, for which one outcome is not expected over the other. In this case a coin tossing has an entropy of one **bit**.



Information-theoretic lower bound

- Sorting an array of n size, the entropy of such a random permutation S is $\log n!$ bits.
- Since a comparison can give only two results, the **maximum** amount of information it provides is 1 bit.
- Therefore, after k comparisons the remaining entropy of the permutation, given the results of those comparisons, is at least $\log_2(n!) - k$ bits on average. To perform the sort, complete information is needed, so the **remaining entropy** must be 0. It follows that k must be at least $\log_2(n!)$ in **on average**.

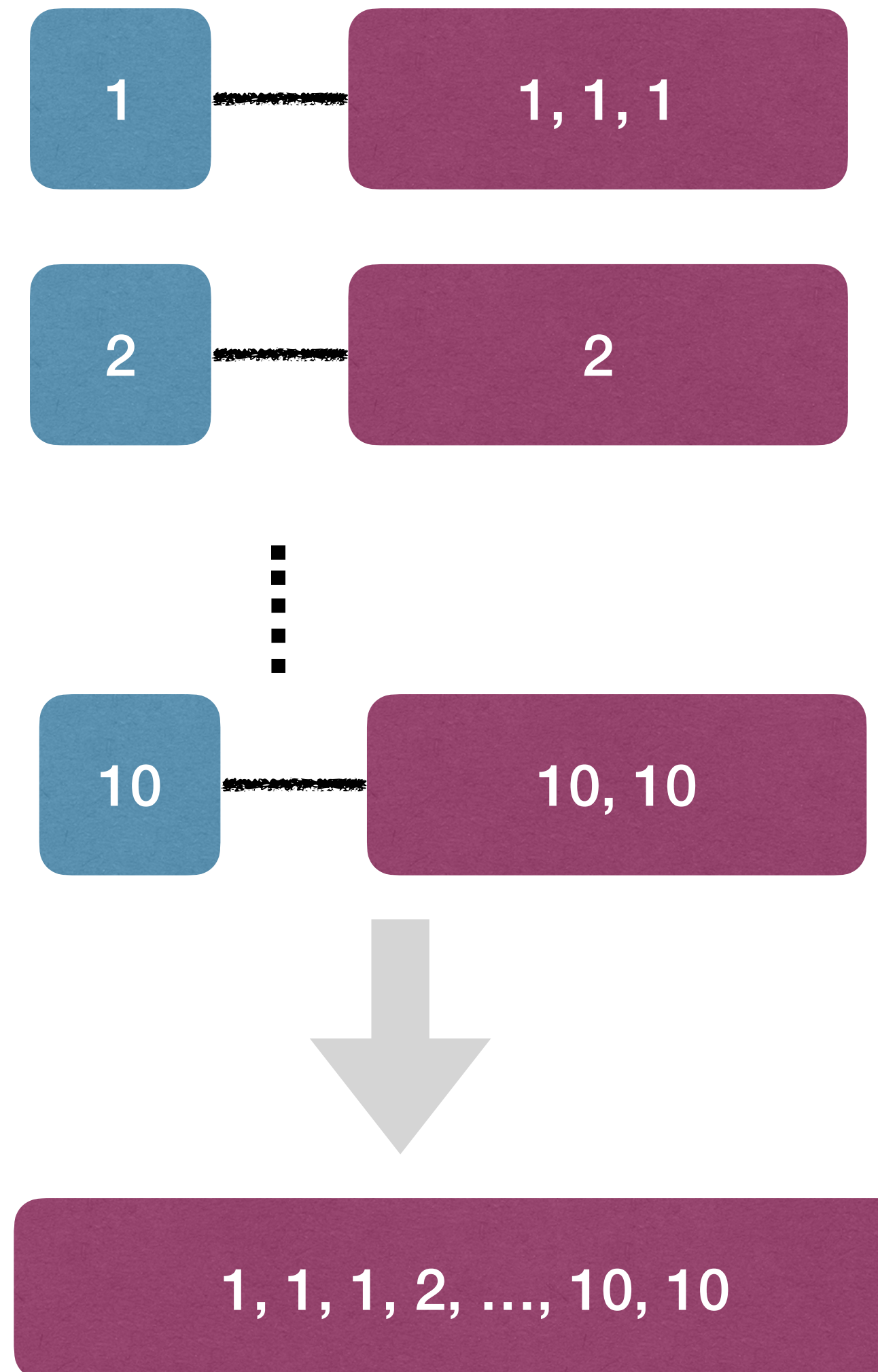


Non-comparison-based sorting



Bucket sort

- Assume we want to sort n integers, and we know each item is from the set $[10]$. Can we beat $\Theta(n \log n)$?
- Of course, very easy!
 - ▶ Create 10 empty lists. (These are the **buckets**.)
 - ▶ Scan through input, for each item, append it to the end of the corresponding list.
 - ▶ Concatenate all lists.





Bucket sort

1. Create d empty lists. (These are the **buckets**.)
 2. Scan through input, for each item, append it to the end of the corresponding list.
 3. Concatenate all lists.
- This algorithm only takes $\Theta(n)$ time.
 - This is not a comparison based algorithm.
 - ▶ No comparison between items are made.
 - ▶ Instead the algorithm uses **actual values** of the items.



Bucket sort

- In general, if the input items are all from set $[d]$, then we can use the following algorithm to sort them.

BucketSort(A, d):

$\langle L_1, L_2, \dots, L_d \rangle = \text{CreateBuckets}(d)$

for $i := 1$ **to** $A.length$

$\text{AssignToBucket}(A[i])$

$\text{CombineBuckets}(L_1, L_2, \dots, L_d)$

- Total time complexity is $\Theta(n + d)$.
 - $\Theta(d)$ time to create buckets.
 - $\Theta(n)$ time to assign items to buckets.
 - $\Theta(d)$ time to combine buckets.

What if $n \ll d$?
Say sort 1000 64-bit integers.



Bucket sort

- If the range of items' values is too large, allow each bucket to hold multiple values.
- Allocate k buckets each responsible for an interval of size d/k .
- But now we need to sort each bucket before combining them.

BucketSort(A, k):

$\langle L_1, L_2, \dots, L_k \rangle = \text{CreateBuckets}(k)$

for $i := 1$ **to** $A.length$

 AssignToBucket($A[i]$)

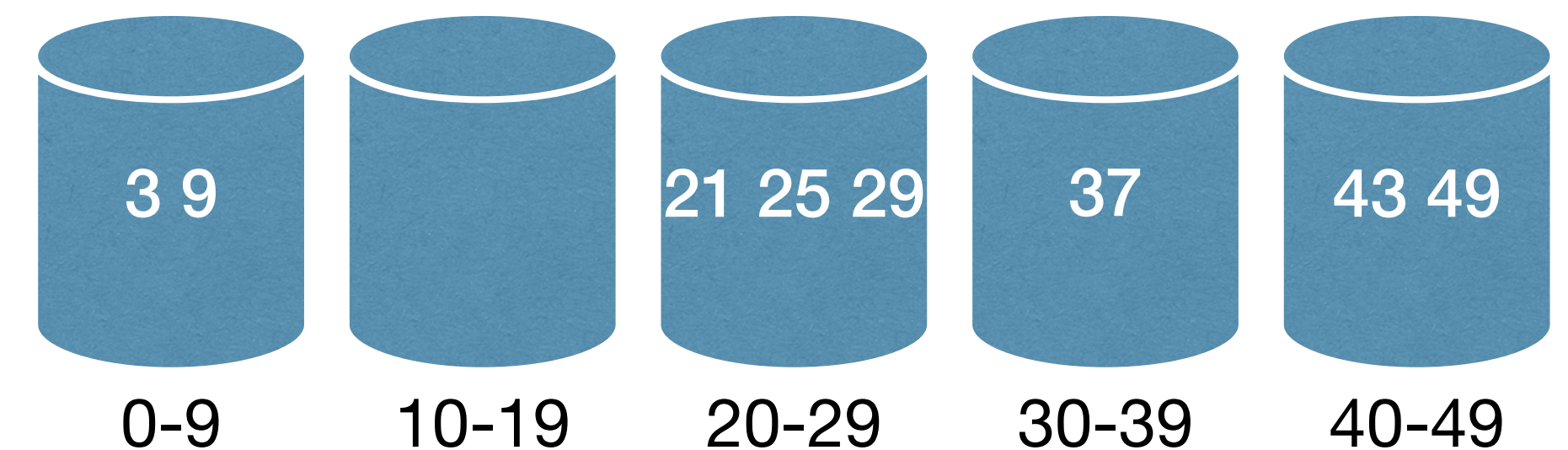
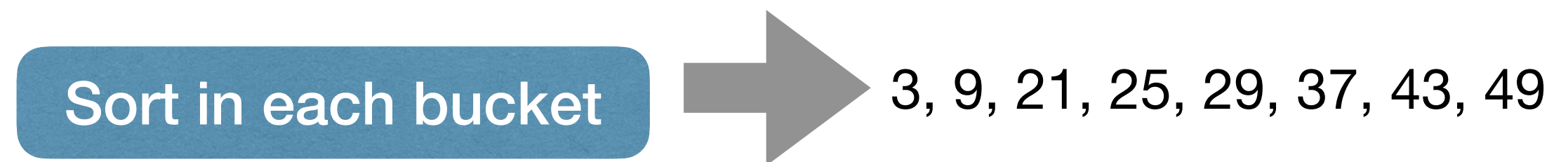
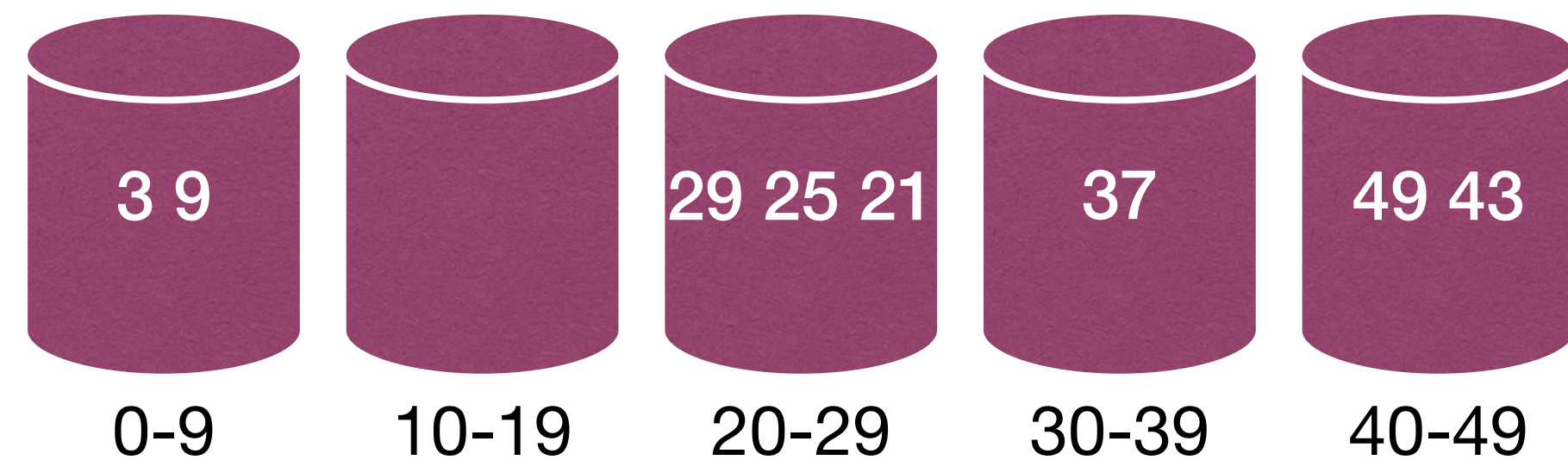
for $j := 1$ **to** k

 SortWithinBucket(L_j)

CombineBuckets(L_1, L_2, \dots, L_d)

29 25 3 49 9 37 21 43

$n = 8, d = 50, k = 5$





Bucket sort

- Runtime is $\Theta(n + k)$, plus cost for sorting within buckets.
 - If items are uniformly distributed and we use insertion sort, expected cost for sorting is $O(k \cdot (n/k)^2) = O(n^2/k)$.
- Expected total runtime is $O(n + k + (n^2/k))$, which is $O(n)$ when we have $k \approx n$ buckets.
- BucketSort can be stable.

BucketSort(A, k):

$\langle L_1, L_2, \dots, L_k \rangle = \text{CreateBuckets}(k)$

for $i := 1$ **to** $A.length$

 AssignToBucket($A[i]$)

for $j := 1$ **to** k

 SortWithinBucket(L_j)

CombineBuckets(L_1, L_2, \dots, L_d)



Radix sort

- Assume we want to sort n decimal integers each of d -digits.
- How about recursive bucket sort?
 - ▶ Based on most significant bit, assign items to 10 buckets.
 - ▶ Sort recursively in each bucket (i.e., use 2nd most significant bit).
 - ▶ Valid but not for now...
- RadixSort: iterative, starting from **least** significant bit.

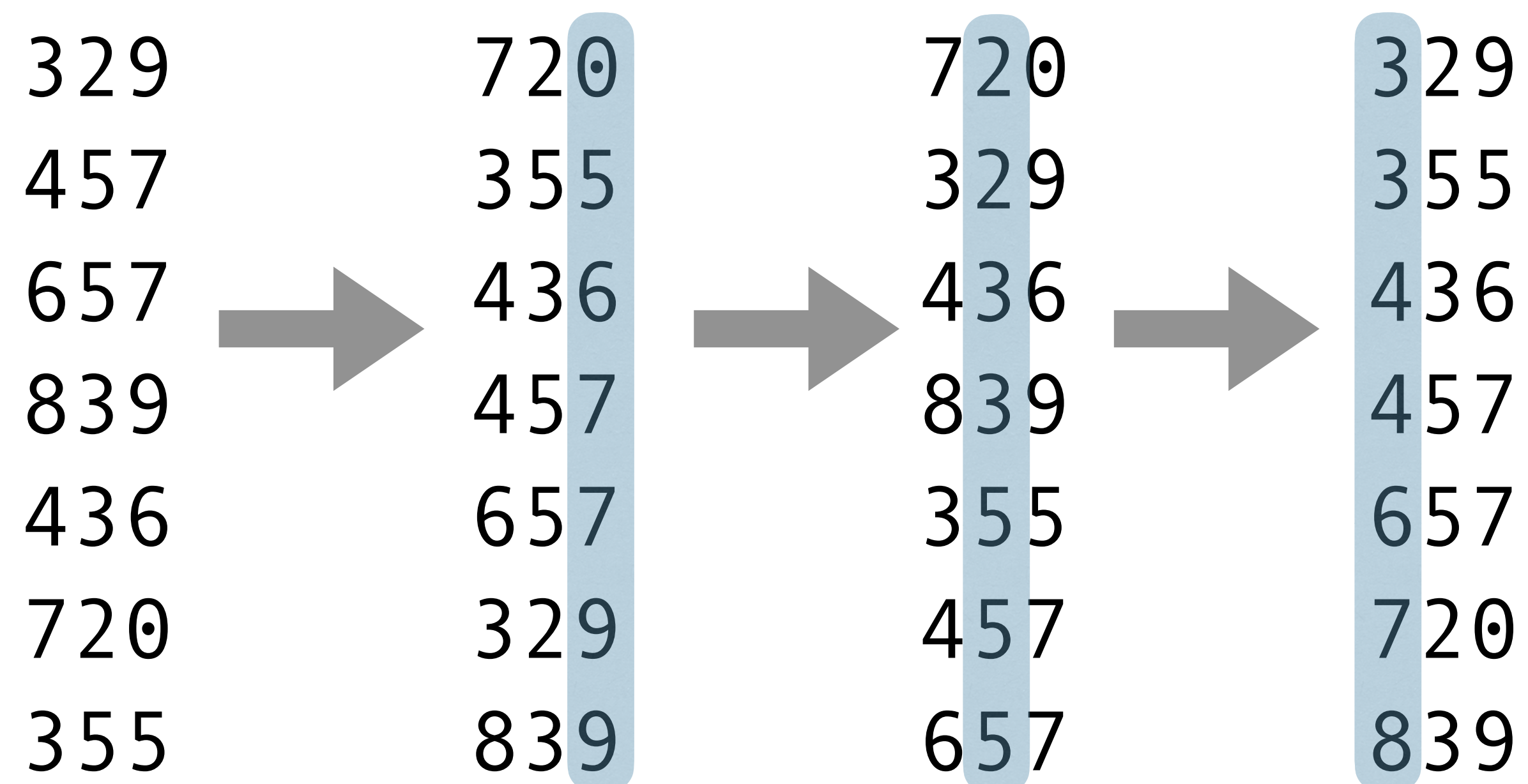


Radix sort

RadixSort(A, d):

for $i := 1$ to d

use-a-stable-sort-to-sort-A-on-digit- i





Radix sort

Claim : after i^{th} iteration, items are sorted by their rightmost i bits.

- Use induction to prove the claim.
- **[Basis]** The claim holds after the first iteration.
- **[Hypothesis]** Assume the claim holds after the first $k - 1$ iterations.
- **[Inductive Step]** Consider two items a and b after k iterations.
 - W.l.o.g., assume a appears before b . Thus, $a[k] \leq b[k]$.
 - If $a[k] < b[k]$, then it must be $a[k...1] < b[k...1]$.
 - If $a[k] = b[k]$, since we use **stable sort**, it must be $a[(k-1)...1] \leq b[(k-1)...1]$. Again, $a[k...1] \leq b[k...1]$.



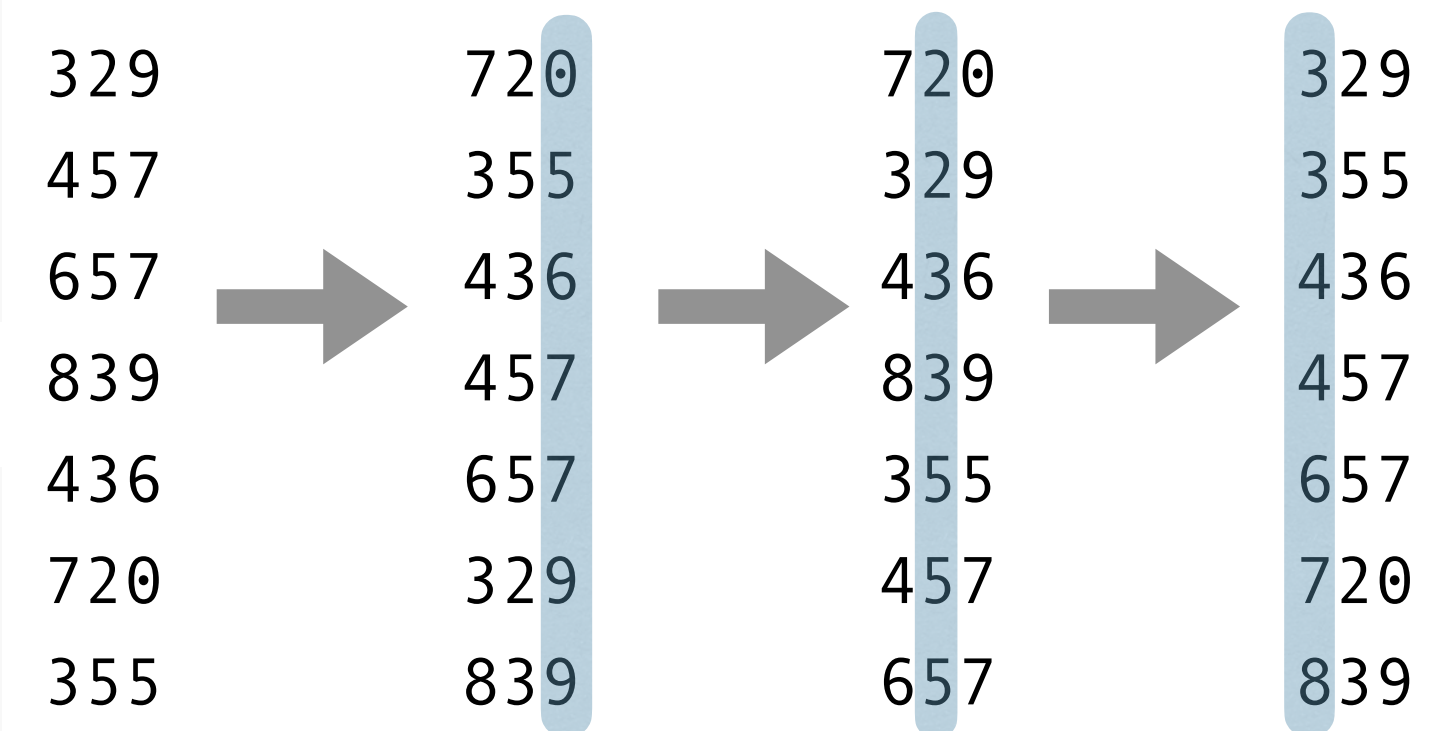
Radix sort

RadixSort(A, d):

for $i := 1$ **to** d
 use-a-stable-sort-to-sort-A-on-digit- i

RadixSort(A, d):

for $i := 1$ **to** d
 use-**bucket-sort**-to-sort-A-on-digit- i



- Since only considering decimal numbers, we only need $10 = \Theta(1)$ buckets.
- RadixSort can sort n decimal d -digits numbers in $O(dn)$ time.



Lower bound for sorting by querying the value

- Assume we want to sort n integers.



“What is a_{n-1} ”

...

“What is a_1 ”

...

$n - 1$ queries, $\Theta(n)$ work

a_1	a_2	...	a_{n-1}	a_n
-------	-------	-----	-----------	-------

a_1	a_2	...	a_{n-1} = 1	a_n
-------	-------	-----	------------------	-------

a_1 = 1	a_2	...	a_{n-1} = 1	a_n
--------------	-------	-----	------------------	-------

a_1 = 1	a_2 = 1	...	a_{n-1} = 1	a_n = 1
--------------	--------------	-----	------------------	--------------



“ $a_{n-1} = 1$ ”

“ $a_1 = 1$ ”

... (always say 1)

sorry but $a_2 > 1$

Place a_2 at beginning or end?

a_2 < 1	a_1 = 1	...	a_{n-1} = 1	a_n = 1
--------------	--------------	-----	------------------	--------------

a_1 = 1	a_3 = 1	...	a_n = 1	a_2 > 1
--------------	--------------	-----	--------------	--------------



Lower bound for sorting by querying value

- The algorithm, which queries the input $n - 1$ times, **does not** solve the problem.
- **Any** algorithm which queries the input at most $n - 1$ times does not solve the problem.
- Solving the “sort n integers” problem by querying values of input has a time complexity of $\Omega(n)$.



Summary

- **Lower Bounds:**
 - ▶ Sorting needs $\Omega(n)$ time. (**adversary argument**)
 - ▶ Comparison-based sorting needs $\Omega(n \log n)$. (**decision tree**)
- **Upper Bounds:**
 - ▶ There are $O(n \log n)$ **comparison-based** sorting algorithms.
 - ▶ BucketSort, RadixSort can be $\Theta(n)$ in many cases.



Further reading

- [CLRS] Ch.8

