

ساختمان داده و الگوریتم ها (CE203)

جلسه دهم:
کران پایین برای مرتب سازی

سجاد شیرعلی شمرضا

پاییز 1400

دوشنبه، 3 آبان 1400

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 8.1
- امتحانک دوم: دوشنبه هفته آینده، 10 آبان 1400 در طی ساعت کلاس به صورت برخط (مشابه امتحانک اول)

کران پایین برای مرتب سازی

آیا می توان الگوریتم مرتب سازی بهتر از $O(n \lg n)$ هم طراحی کرد؟

$O(n \log n)$ ALGORITHMS WE'VE SEEN

- MergeSort
 - Worst-case $\Theta(n \log n)$ time.
- QuickSort
 - Expected: $\Theta(n \log n)$

$O(n \log n)$ ALGORITHMS WE'VE SEEN

- MergeSort
 - Worst-case $\Theta(n \log n)$ time
- QuickSort
 - Expected: $\Theta(n \log n)$

THE QUESTION IS...
***CAN WE DO
BETTER?***

INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

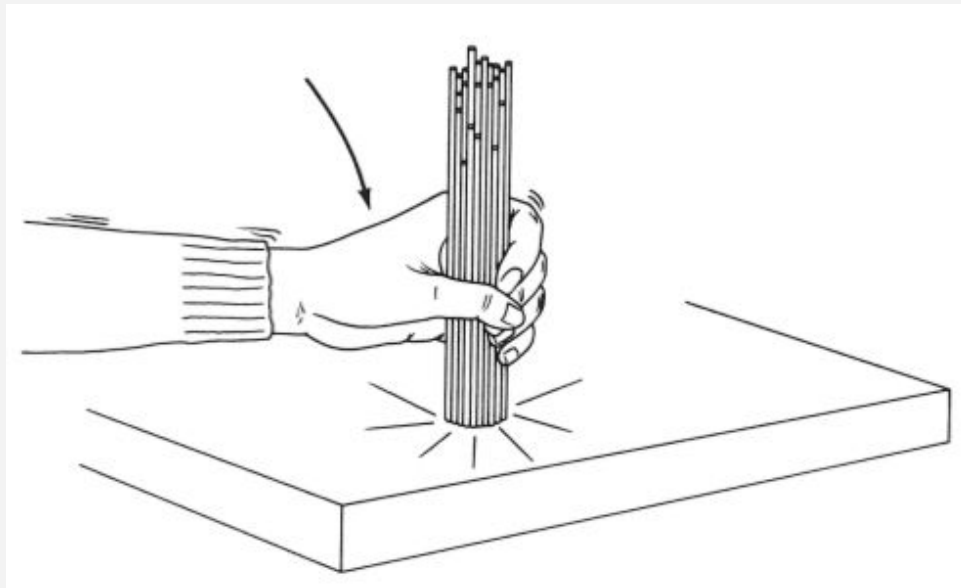
- For each number, break off a piece of spaghetti whose length is that number $O(n)$

INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

- For each number, break off a piece of spaghetti whose length is that number $O(n)$
- Take all the spaghetti in your fist, and push their lower sides against the table $O(1)$

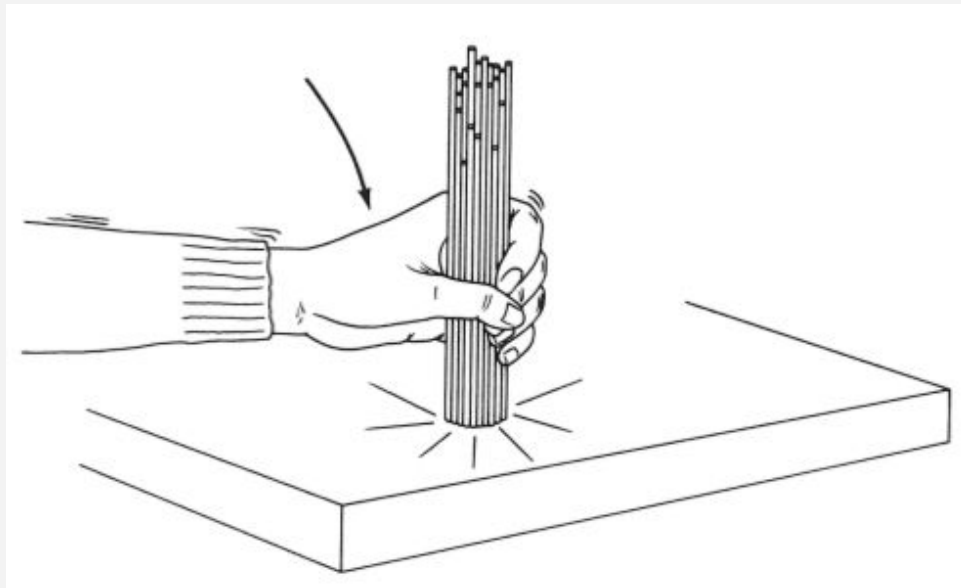


INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

- For each number, break off a piece of spaghetti whose length is that number $O(n)$
- Take all the spaghetti in your fist, and push their lower sides against the table $O(1)$
- Lower your other hand on the bundle of spaghetti - the first spaghetti you touch is the longest one. Remove it, transcribe its length, and repeat until all spaghetti have been removed. $O(n)$

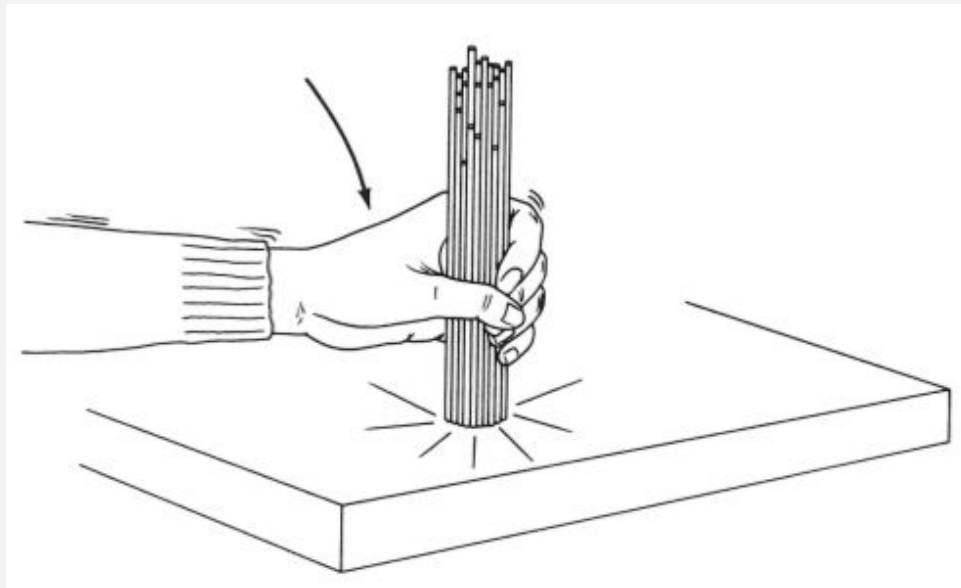


INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

- For each number, break off a piece of spaghetti whose length is that number $O(n)$
- Take all the spaghetti in your fist, and push their lower sides against the table $O(1)$
- Lower your other hand on the bundle of spaghetti - the first spaghetti you touch is the longest one. Remove it, transcribe its length, and repeat until all spaghetti have been removed. $O(n)$



While you shouldn't take this algorithm too seriously... it does raise some important questions!

WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

Input: some real numbers

Output: sorted real numbers

Operations allowed: breaking spaghetti,
dropping on tables, lowering hand

WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

Input: some real numbers

Output: sorted real numbers

Operations allowed: breaking spaghetti,
dropping on tables, lowering hand

In a CS class where we're more concerned with what computers can do, the first model seems more reasonable.



سوال؟

COMPARISON-BASED SORTING

- **You want to sort an array of items**
- **You can't access the items' values directly: you can only *compare* two items and find out which is bigger or smaller.**
- Examples: Insertion Sort, MergeSort, QuickSort

COMPARISON-BASED SORTING

- **You want to sort an array of items**
- **You can't access the items' values directly: you can only *compare* two items and find out which is bigger or smaller.**
- Examples: Insertion Sort, MergeSort, QuickSort

“Comparison-based sorting algorithms” are general-purpose.

The algorithm makes no assumption about the input elements other than that they belong to some totally ordered set.

COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

$A[0]$

$A[1]$

$A[2]$

$A[3]$

COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this YES/NO question to figure out how to sort the items)

$A[0]$

$A[1]$

$A[2]$

$A[3]$

Is $A[1]$ bigger than $A[3]$?

Yes!

A Comparison-based
Sorting Algorithm

*All-knowing
Genie*

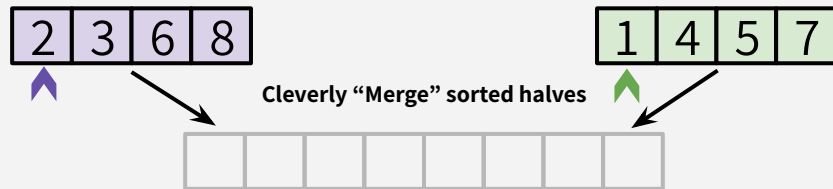
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



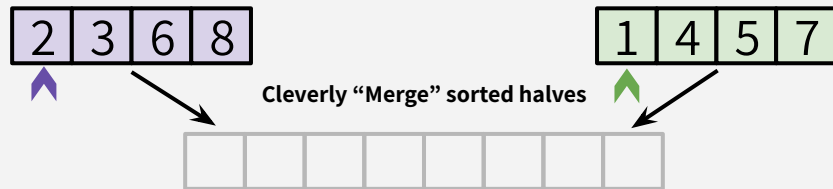
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **1**?

MergeSort
algorithm

*All-knowing
Genie*

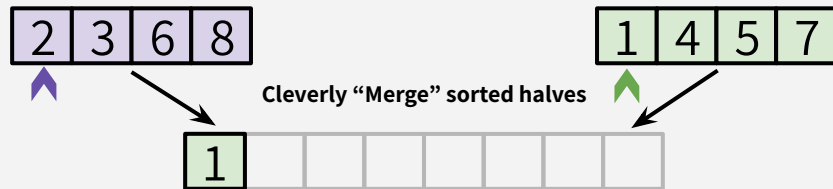
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this *YES/NO* question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **1**?

Yes!

MergeSort
algorithm

*All-knowing
Genie*

COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **4**?

MergeSort
algorithm

*All-knowing
Genie*

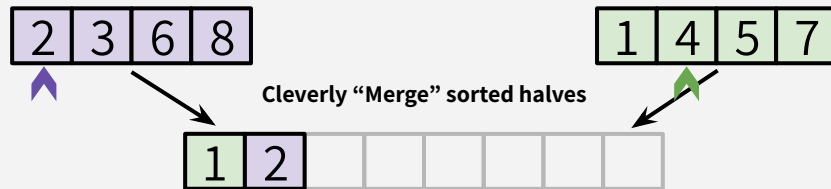
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **4**?

No!

MergeSort
algorithm

*All-knowing
Genie*

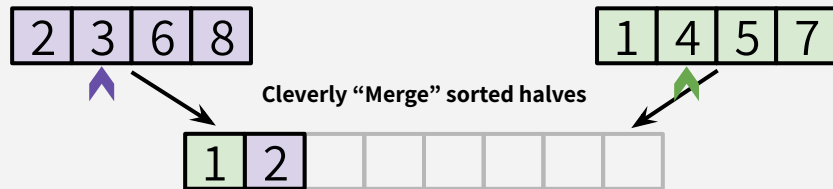
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this *YES/NO* question to figure out how to sort the items)

For example, MergeSort works like this:



Is **3** bigger than **4**?

MergeSort
algorithm

*All-knowing
Genie*

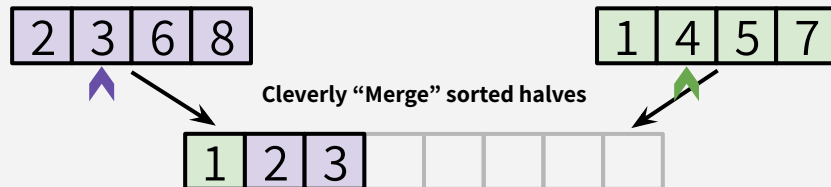
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is 3 bigger than 4 ?

No!

MergeSort
algorithm

*All-knowing
Genie*

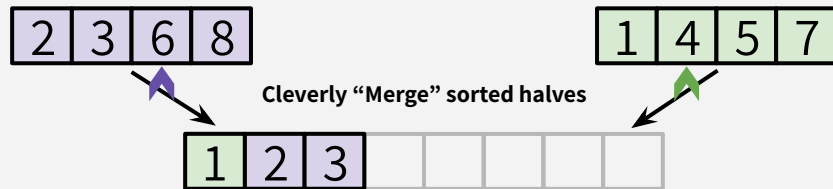
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this *genie* this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **6** bigger than **4**?

MergeSort
algorithm

*All-knowing
Genie*

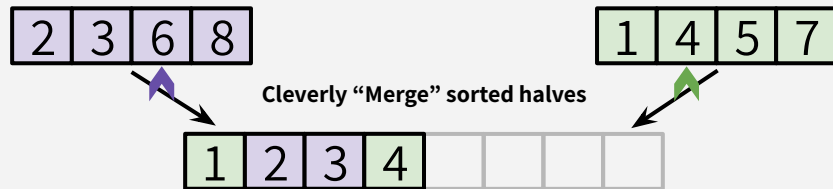
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is 6 bigger than 4 ?

Yes!

MergeSort
algorithm

*All-knowing
Genie*



سوال؟

COMPARISON-BASED SORTING

Theorem:

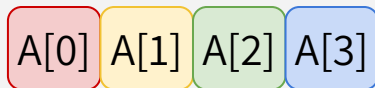
Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

COMPARISON-BASED SORTING

Theorem:

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

Think about it like this: this is the input format that your algorithm is ready to accept.

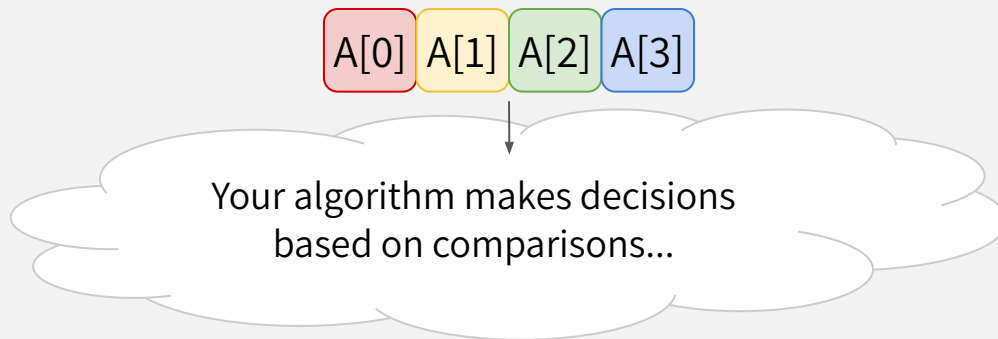


COMPARISON-BASED SORTING

Theorem:

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

Think about it like this: this is the input format that your algorithm is ready to accept.

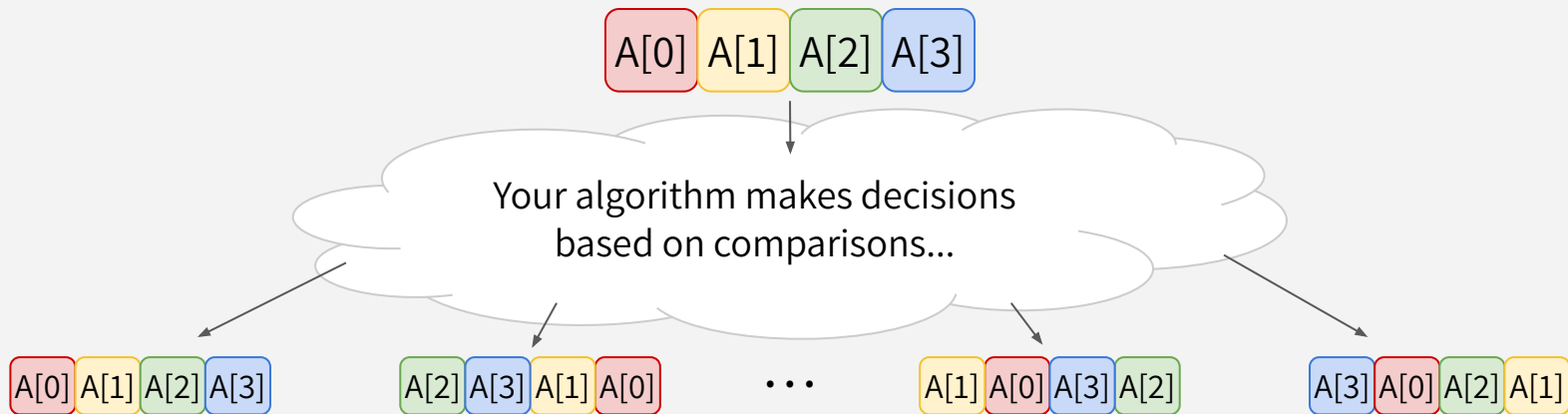


COMPARISON-BASED SORTING

Theorem:

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

Think about it like this: this is the input format that your algorithm is ready to accept.



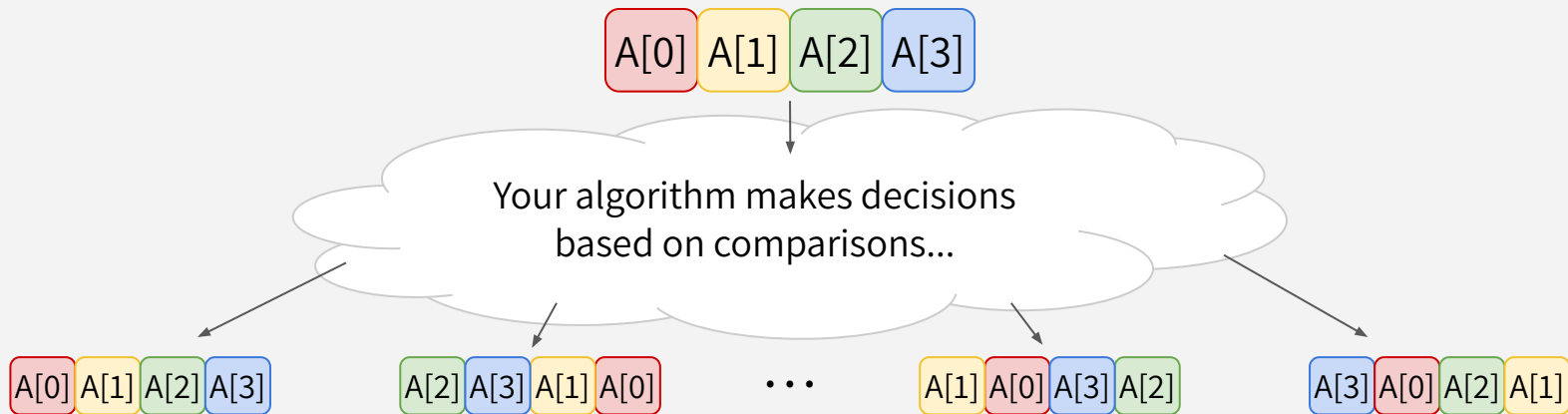
Your algorithm needs to be able to output any one of _____ possible orderings

COMPARISON-BASED SORTING

Theorem:

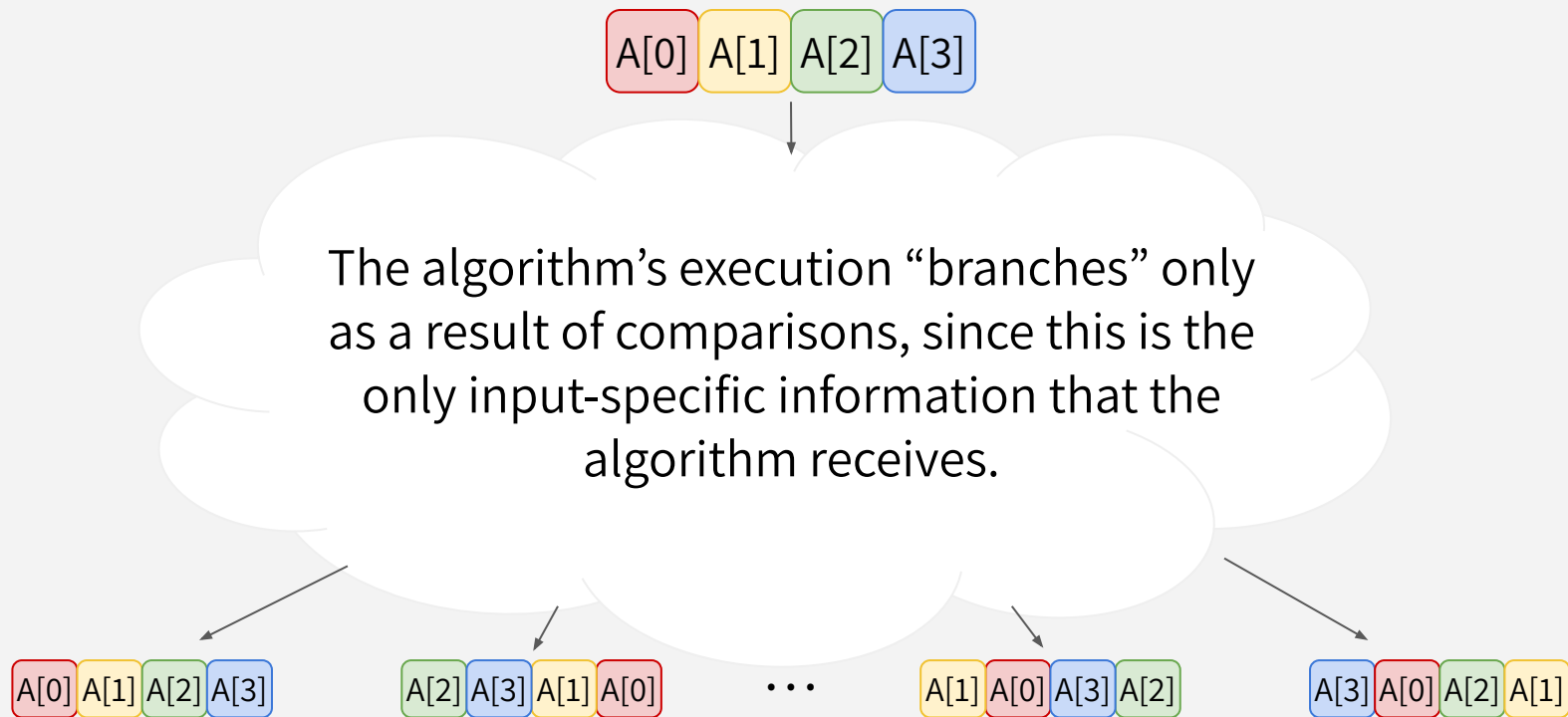
Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

Think about it like this: this is the input format that your algorithm is ready to accept.



Your algorithm needs to be able to output any one of **n!** possible orderings

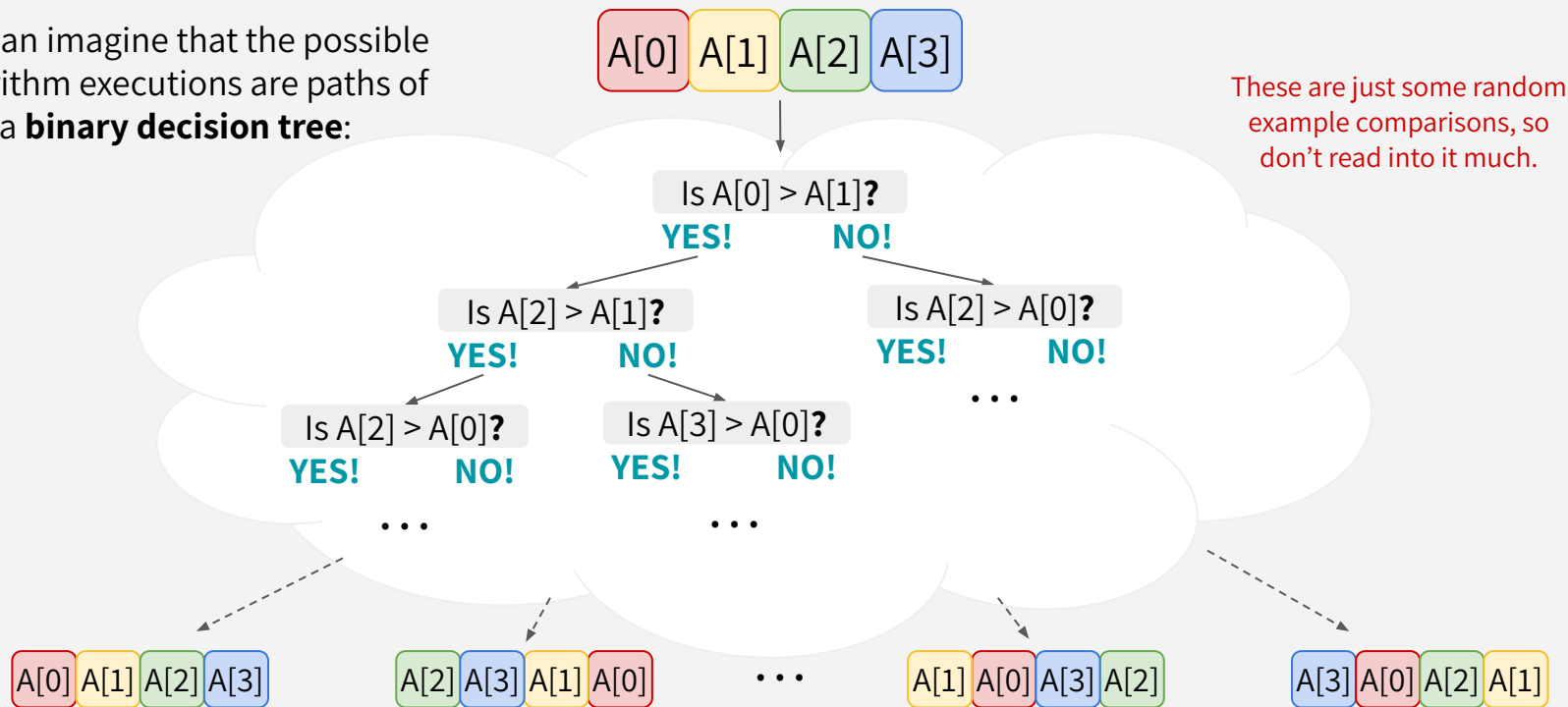
COMPARISON-BASED SORTING



Your algorithm needs to be able to output any one of $n!$ possible orderings

COMPARISON-BASED SORTING

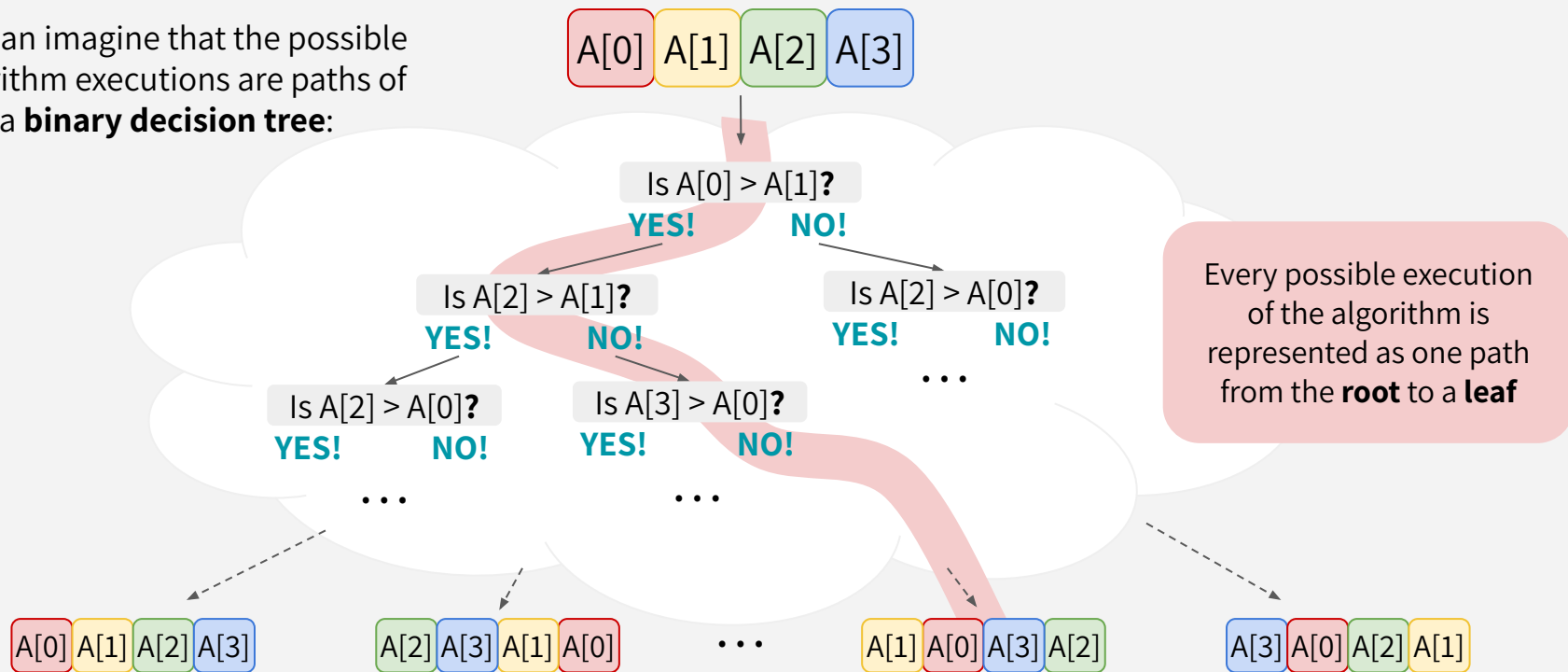
You can imagine that the possible algorithm executions are paths of a **binary decision tree**:



Your algorithm needs to be able to output any one of **$n!$** possible orderings

COMPARISON-BASED SORTING

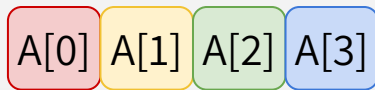
You can imagine that the possible algorithm executions are paths of a **binary decision tree**:



Your algorithm needs to be able to output any one of **n!** possible orderings

COMPARISON-BASED SORTING

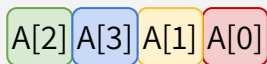
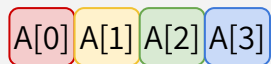
You can imagine that the possible algorithm executions are paths of a **binary decision tree**.



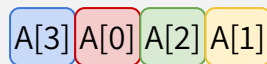
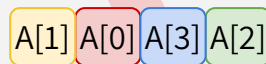
This is a binary tree with at least **$n!$** leaves.

What is the length of the longest possible path?

Every possible execution of the algorithm is represented as one path from the **root** to a **leaf**



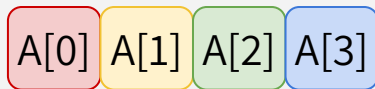
...



Your algorithm needs to be able to output any one of **$n!$** possible orderings

COMPARISON-BASED SORTING

You can imagine that the possible algorithm executions are paths of a **binary decision tree**.

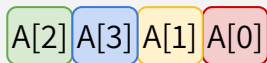
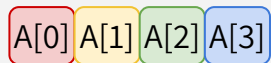


This is a binary tree with at least **$n!$** leaves.

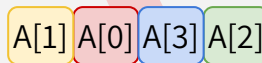
The shallowest tree with **$n!$** leaves is the completely “balanced” one, which has depth **$\log(n!)$**

Thus, in all binary trees with at least **$n!$** leaves, **the longest path has length at least $\log(n!)$**

Every possible execution of the algorithm is represented as one path from the **root** to a **leaf**



...



Your algorithm needs to be able to output any one of **$n!$** possible orderings

COMPARISON-BASED SORTING

The longest path has length at least $\log(n!)$

Consequently, any execution of a comparison-based sorting algorithm has to perform at least $\log(n!)$ steps.

The worst-case runtime is at least $\log(n!) = \Omega(n \log n)$.

COMPARISON-BASED SORTING

The longest path has length at least $\log(n!)$

Consequently, any execution of a comparison-based sorting algorithm has to perform at least $\log(n!)$ steps.

The worst-case runtime is at least $\log(n!) = \Omega(n \log n)$.

$$\begin{aligned}\log(n!) &= \log 1 + \log 2 + \cdots + \log(n-1) + \log n \\ &\geq \log\left(\frac{n}{2} + 1\right) + \log\left(\frac{n}{2} + 2\right) + \cdots + \log(n-1) + \log n \\ &\geq \left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) \\ &= \frac{1}{2}n(\log n - \log 2) \\ &= \Omega(n \log n)\end{aligned}$$

PROOF RECAP

Theorem:

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

- Any deterministic comparison-based algorithm can be represented as a decision tree with $n!$ leaves
- The worst-case runtime is at least the length of the longest path in the decision tree
- All decision trees with $n!$ leaves have a longest path with length at least $\log(n!) = \Omega(n \log n)$
- So, any comparison-based sorting algorithm must have worst-case runtime at least $\Omega(n \log n)$

THE GOOD NEWS

Theorem:

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

This bound also applies to the expected runtime of *randomized* comparison-based sorting algorithms!
The proof is out of scope of this class, but it relies on this theorem.

This means that MergeSort is optimal!

(This is one of the cool things about proving lower bounds - we know when we can declare victory!)

THE GOOD NEWS

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

This bound also applies to
The proof is by induction.

THE QUESTION IS...
**CAN WE DO
BETTER?**

-based sorting algorithms!
theorem.

This means that the algorithm is optimal!

**using a model of computation that's
less silly than spaghetti?*

(This is one of the reasons why we don't prove lower
bounds - we know when we can declare victory!)



سوال؟