Sorting Lower Bounds & Linear Sorting Algorithms

For classroom use only, do not share the slides publicly.

Outline for Today

Sorting Lower Bounds

- Comparison-based sorting algorithms
- [Example] Insertion Sort, Merge Sort (revisited)
- Sorting Lower Bounds

Linear-time sorting algorithms

- Space-Time relationship in algorithm design
- Counting Sort, Bucket Sort, Radix Sort

Sorting Lower Bounds

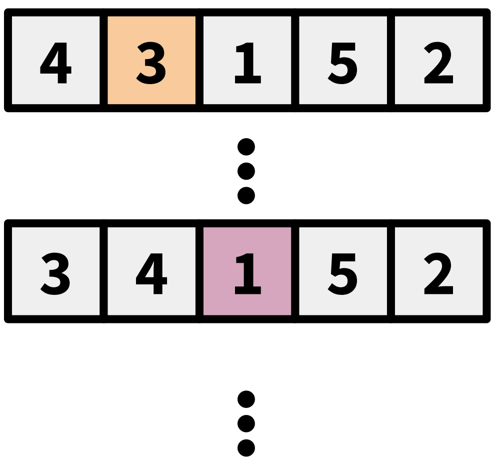
Comparison-Based Sorting

These algorithms use "comparisons" to achieve their output.

`insertion_sort` and `mergesort` are comparison-based sorting algorithms.

A comparison compares two values. e.g. Is **A[0]** < **A[1]**? Is **A[0]** < **A[4]**?

Recall, insertion sort.



Is 3 < 4?

Is 1 < 4? Is 1 < 3?

`mergesort`: comparison happens in the merge subroutine. (explain on board)
`select_k` is a comparison-based algorithm (compare each value with pivot)

Next week, we'll learn about a randomized comparison-based sorting algorithm called `quicksort`.

Comparison-Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.

Remember: not all sorting algorithms require $\Omega(n \log(n))$ time, some algorithms can be faster than this.

Keywords:

Deterministic -> the list will be accurately sorted for sure when the algorithm terminates. There are some algorithms sort the list accurately only with a probability, or sort the list approximately, but are faster.

Comparison-based -> there are some algorithms do not need to do comparison for sorting, e.g. counting sort (will discuss it later)

Proof:

Hmm …

Comparison-Based Sorting

We can represent the comparisons made by a comparison-based sorting algorithm as a decision tree.

Suppose we want to sort three items in **A**.

Is **A[0]** < **A[1]**?

Yes!

No!

Is **A[1]** < **A[2]**?

...

Yes!

No!

A[0], A[1], A[2]

Is **A[0]** < **A[2]**?

Yes!

No!

A[0], A[2], A[1]

A[2], A[0], A[1]

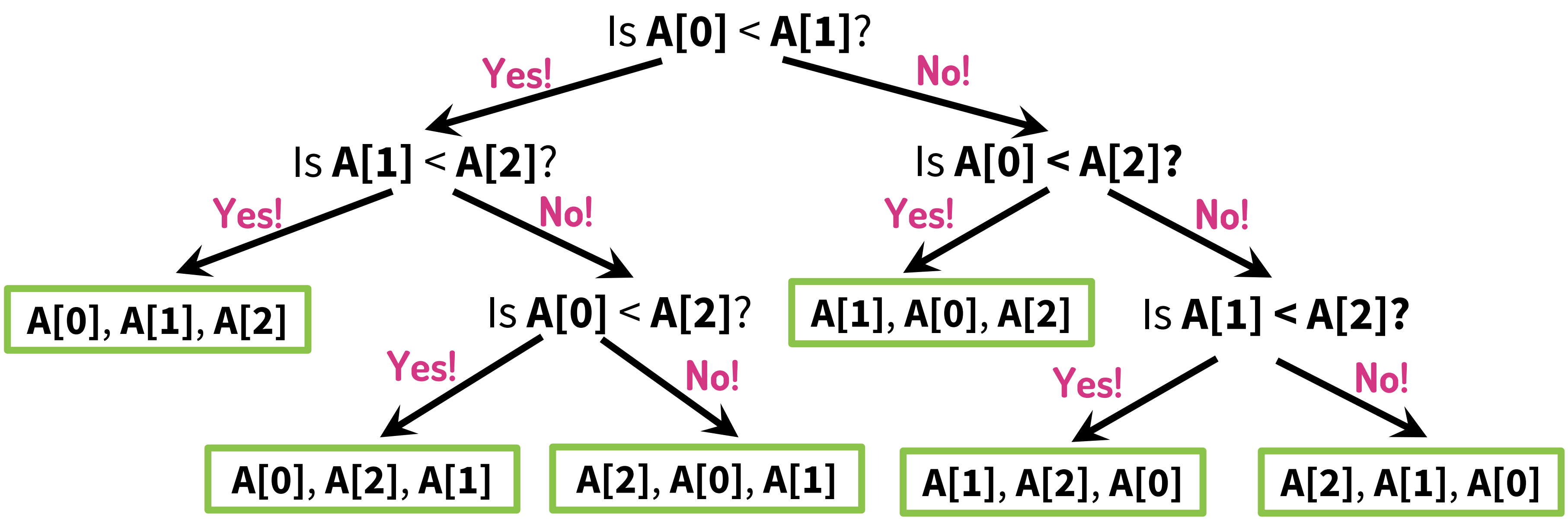
What does the decision tree look like for `insertion_sort`? For `mergesort`?

Explain on board

Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

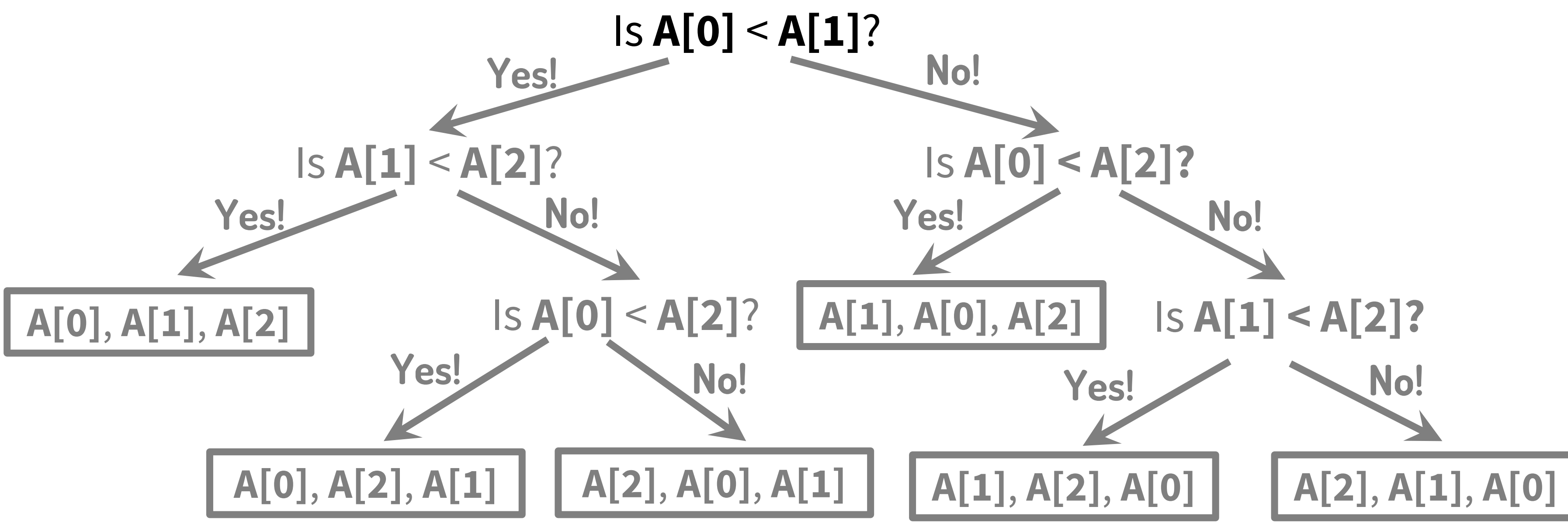


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2]
1 2 3

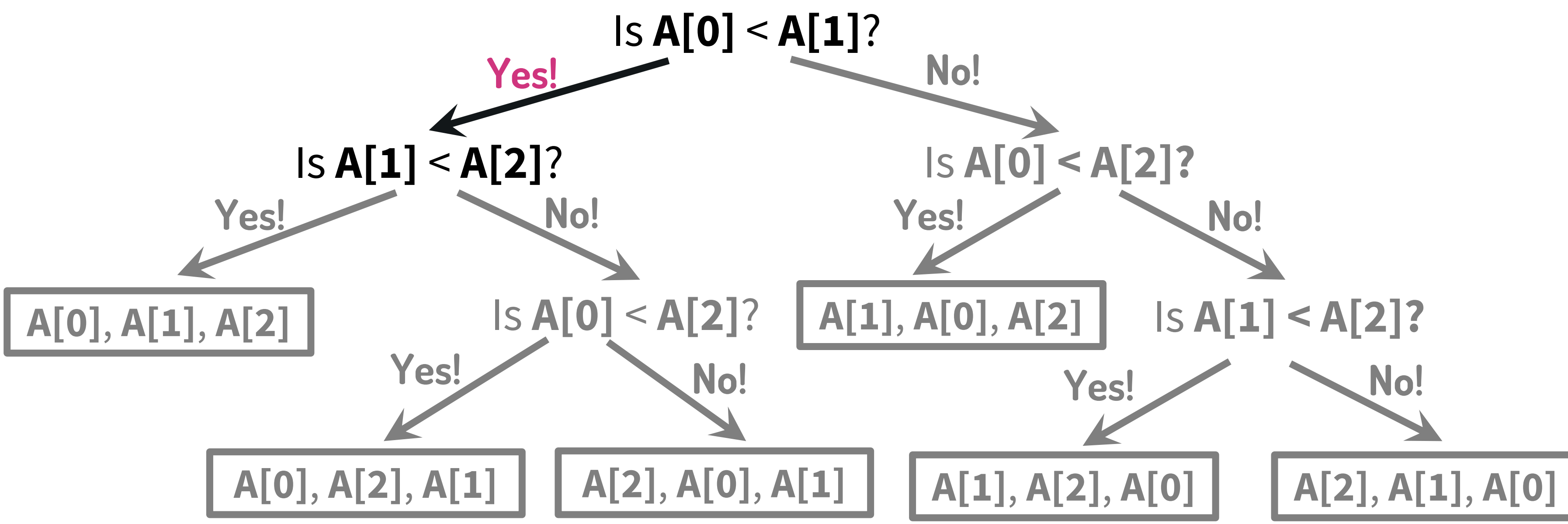


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2]
1 2 3

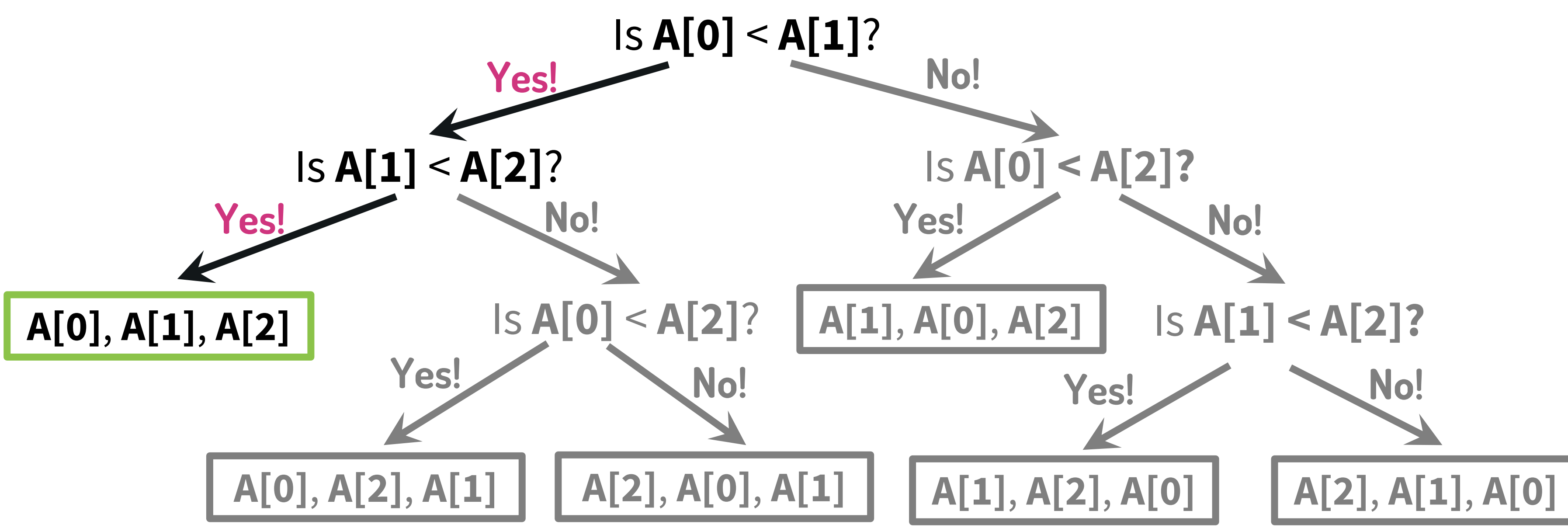


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0]	A[1]	A[2]
1	2	3

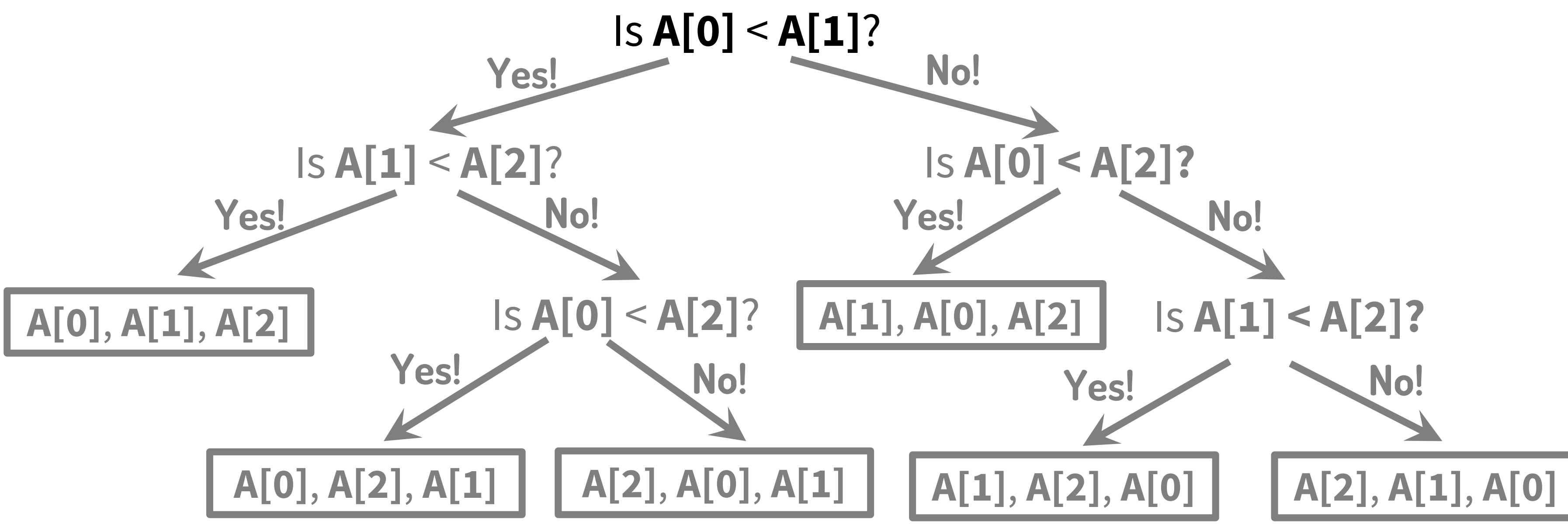


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2]
2 3 1

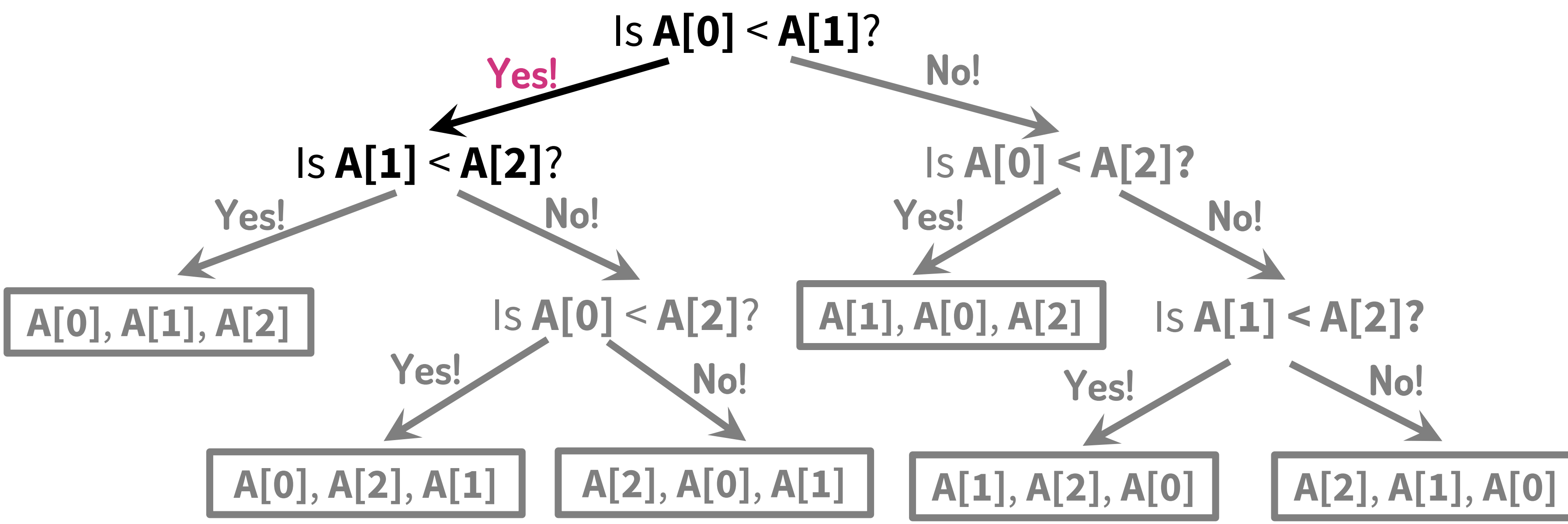


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2]
2 3 1



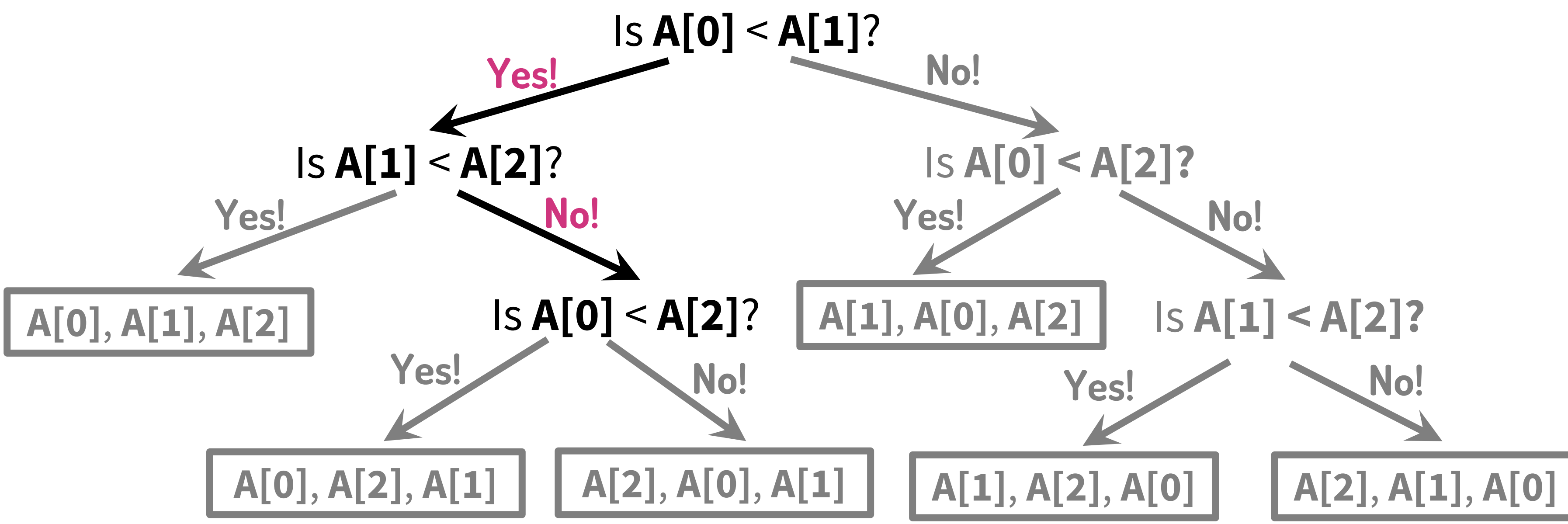
Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0]	A[1]	A[2]
2	3	1

A[0]	A[2]	A[1]
2	1	3

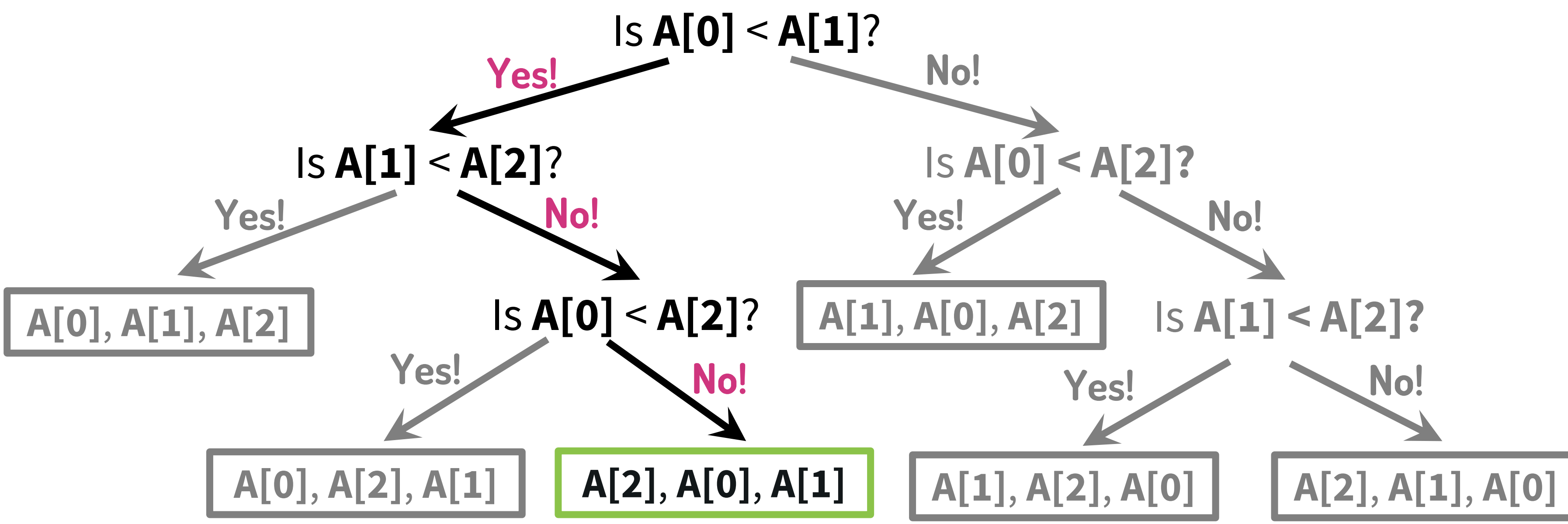


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2]	A[0] A[2] A[1]	A[2] A[0] A[1]
2 3 1	2 1 3	1 2 3

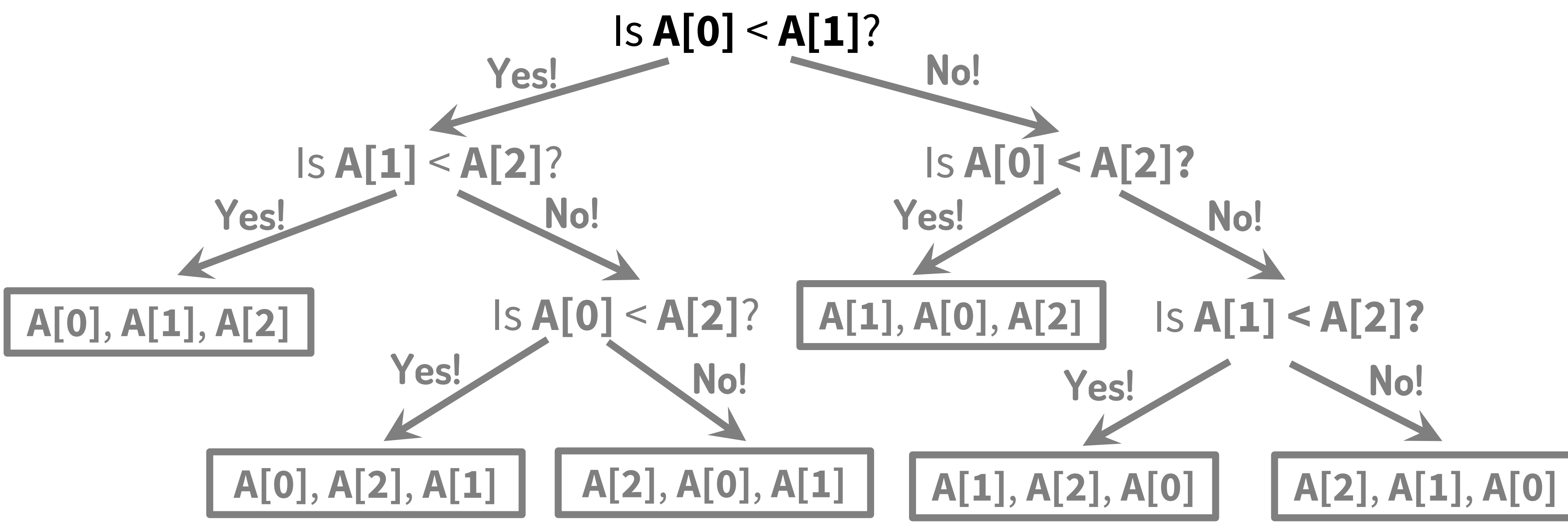


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0]	A[1]	A[2]
3	2	1

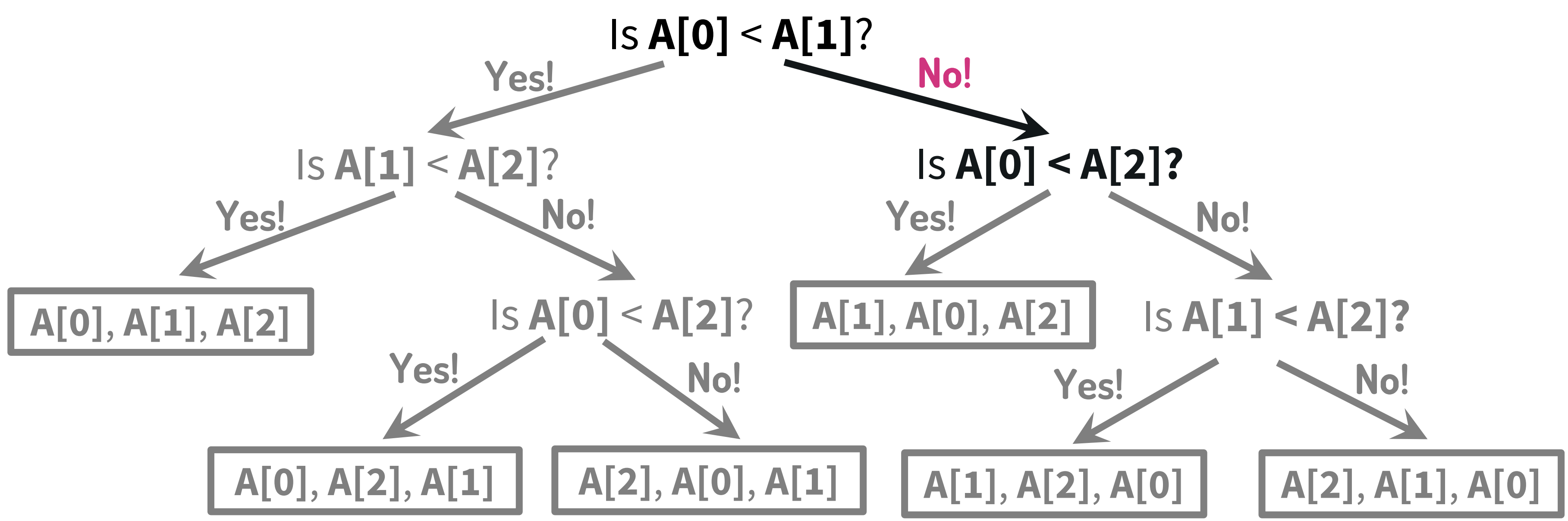


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0]	A[1]	A[2]		A[1]	A[0]	A[2]
3	2	1		2	3	1



Comparison-Based Sorting

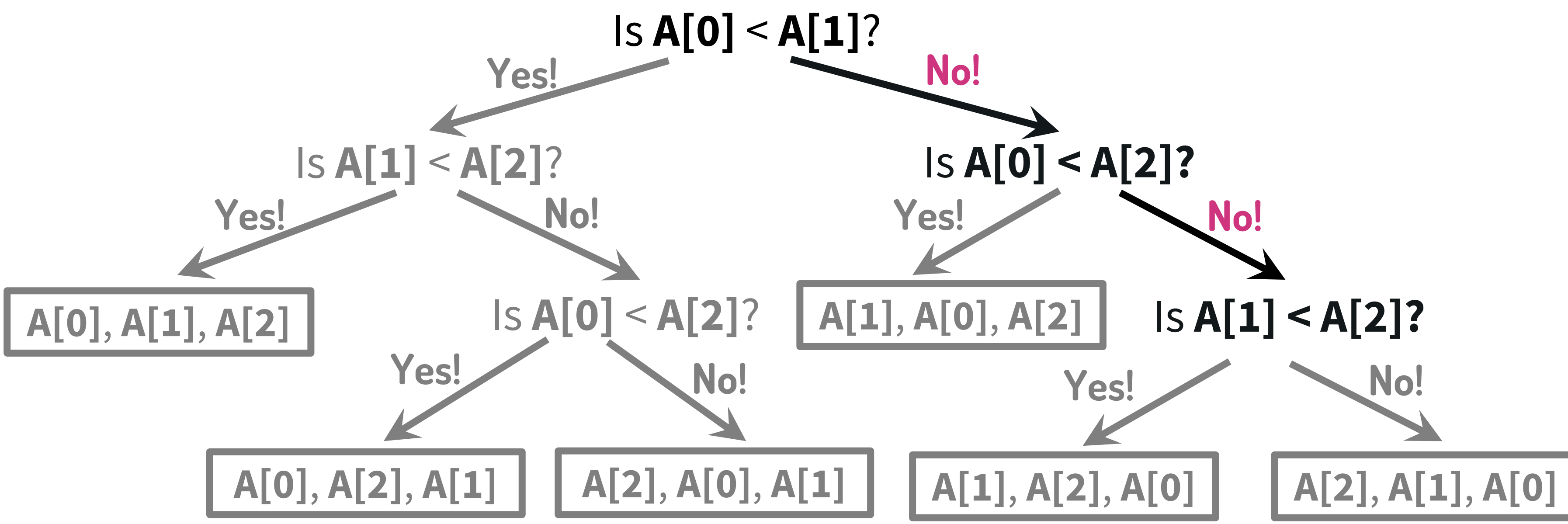
The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0]	A[1]	A[2]
3	2	1

A[1]	A[0]	A[2]
2	3	1

A[1]	A[2]	A[0]
2	1	3

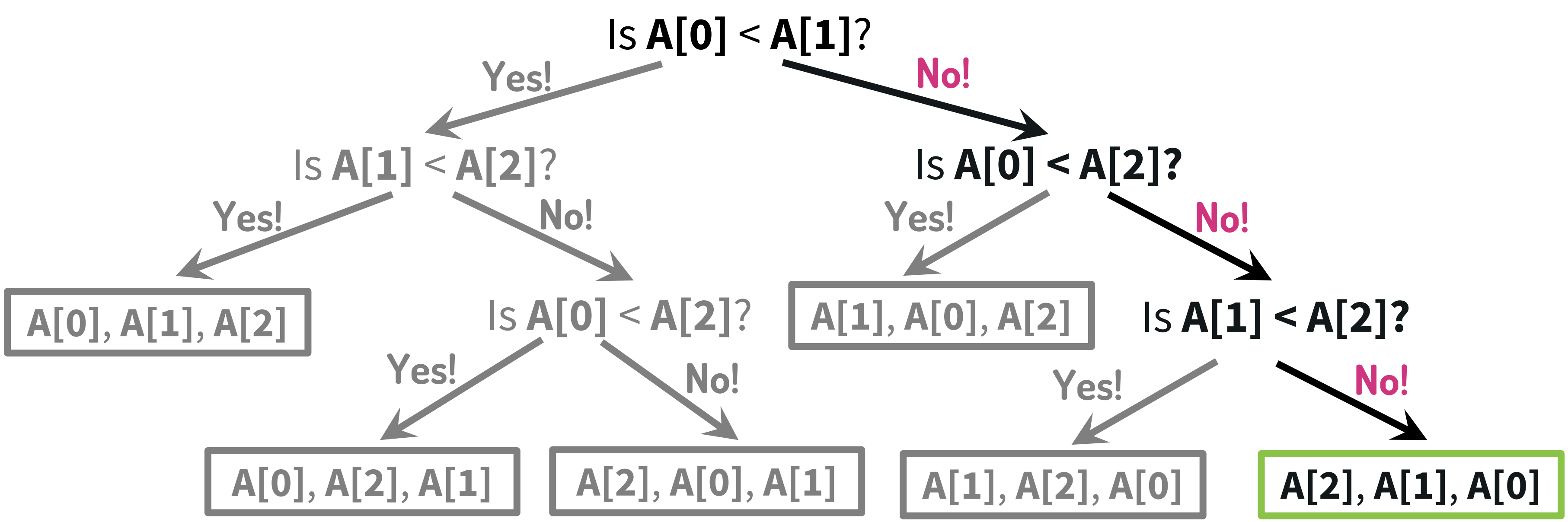


Comparison-Based Sorting

The decision for insertion sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

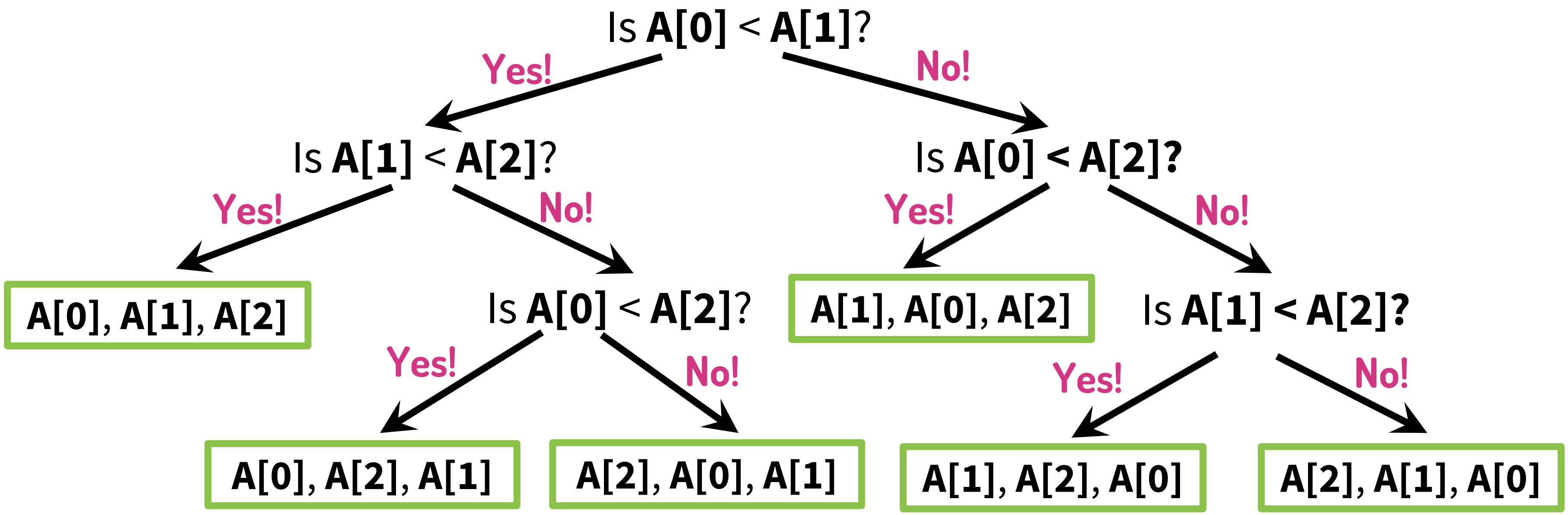
A[0] A[1] A[2]	A[1] A[0] A[2]	A[1] A[2] A[0]	A[2] A[1] A[0]
3 2 1	2 3 1	2 1 3	1 2 3



Comparison-Based Sorting

The decision for insertion sort

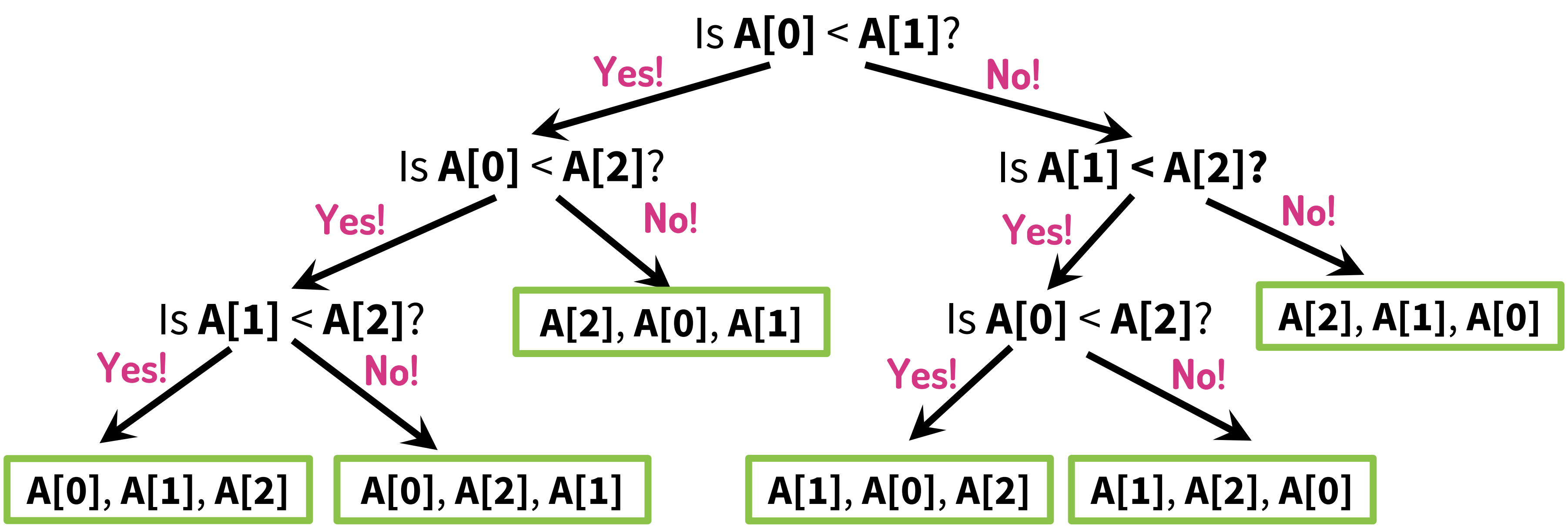
Different input is routed through different paths in the tree
Each leaf node corresponds to a possible ordering of the input
Time complexity is the worst case run time, so it corresponds to the longest path



Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**



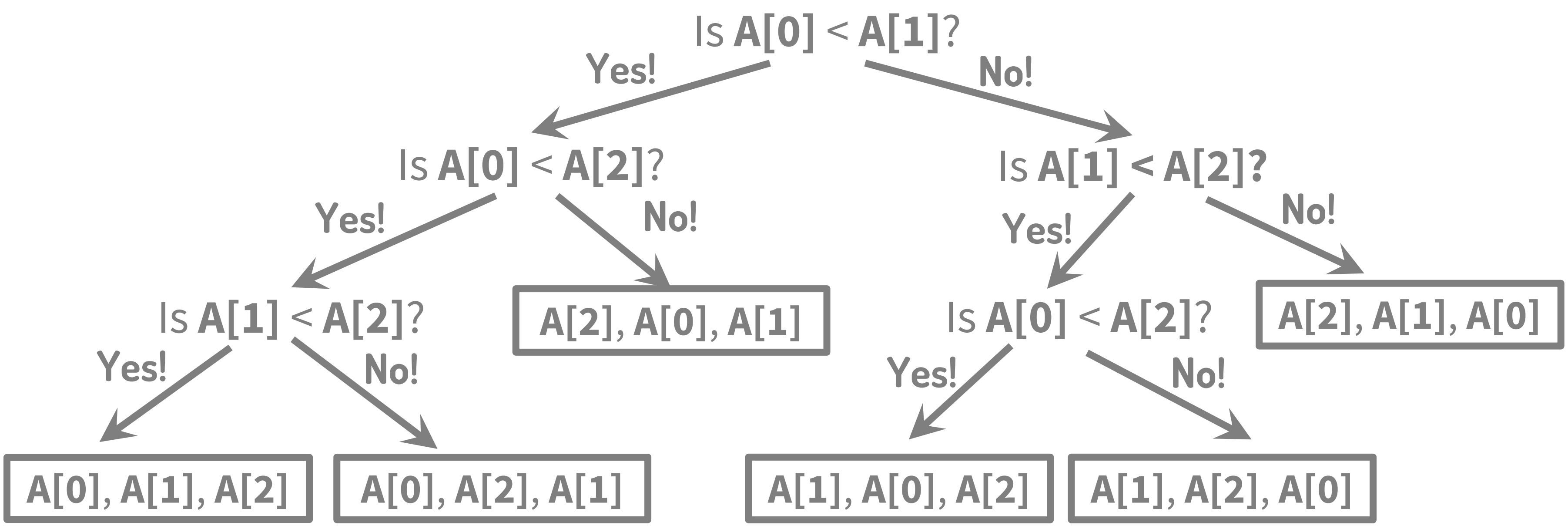
Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2] $\xrightarrow{\text{Divide into smallest problems}}$ (A[0]) (A[1]) (A[2])

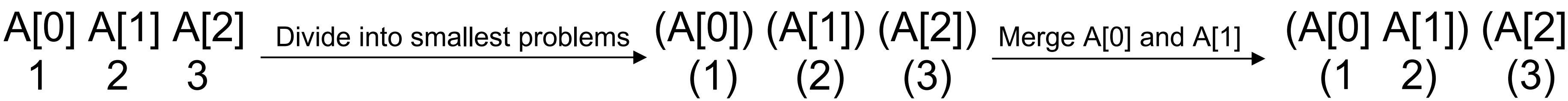
1 2 3 → (1) (2) (3)

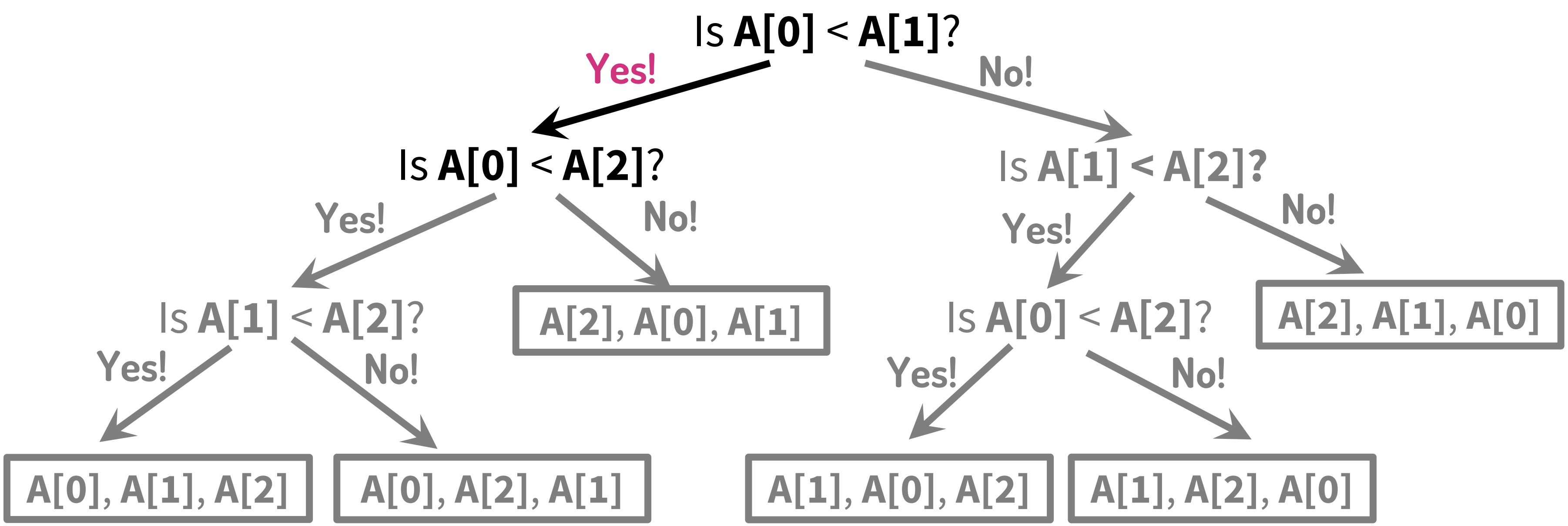


Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

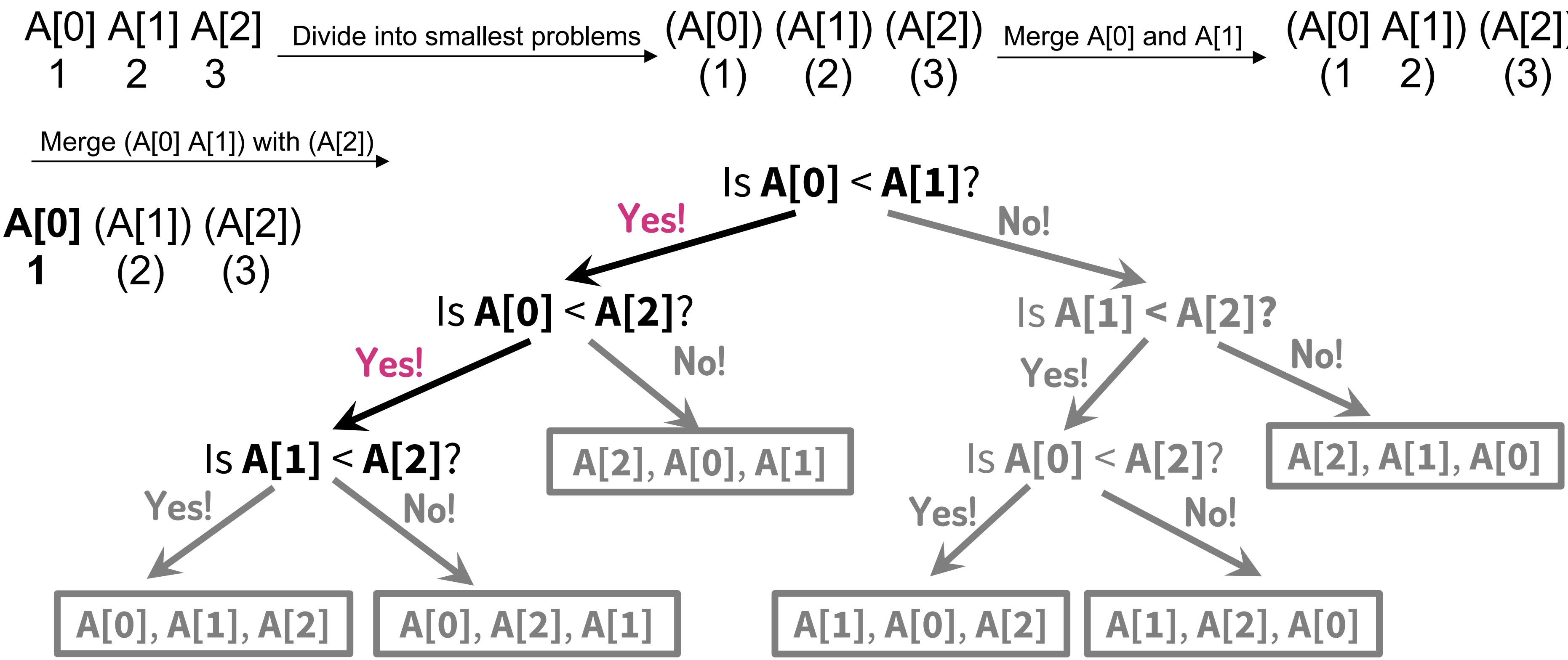




Comparison-Based Sorting

The decision for merge sort

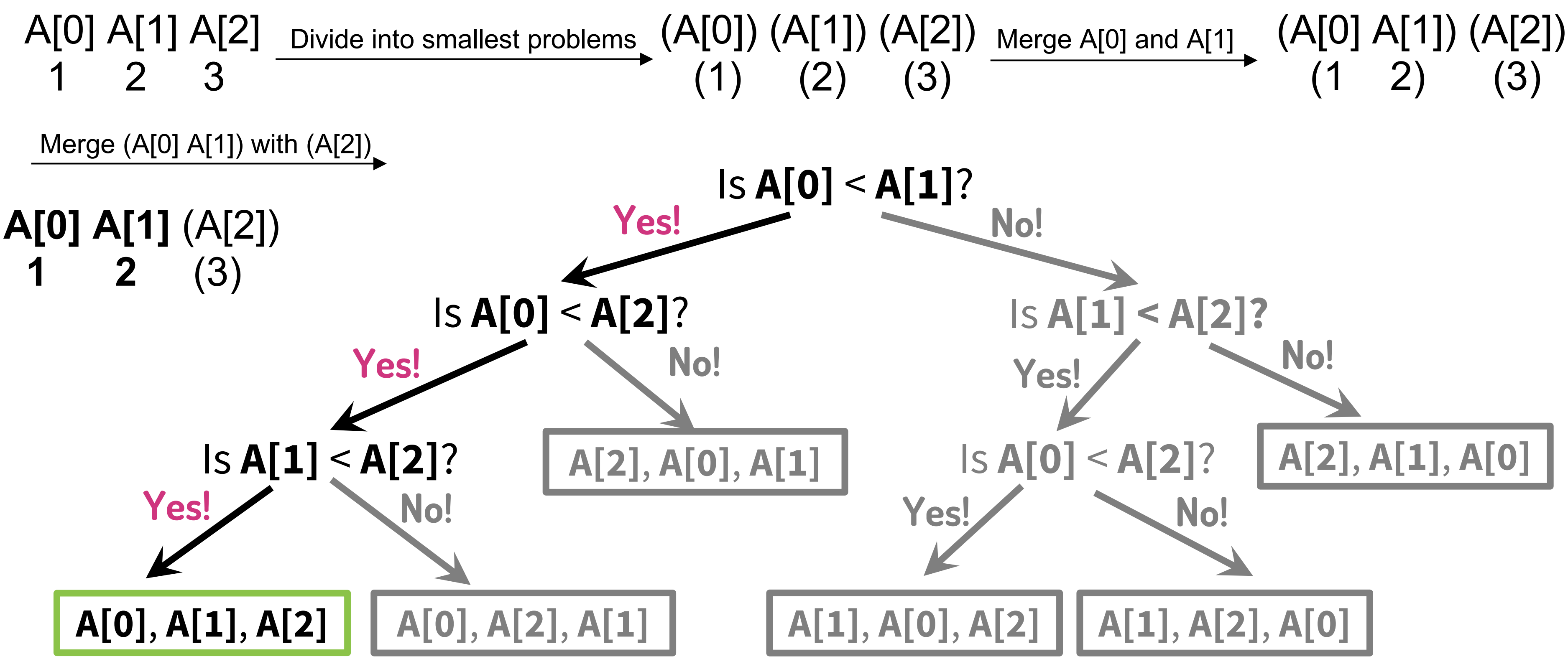
Suppose we want to sort three items in **A: A[0] A[1] A[2]**



Comparison-Based Sorting

The decision for merge sort

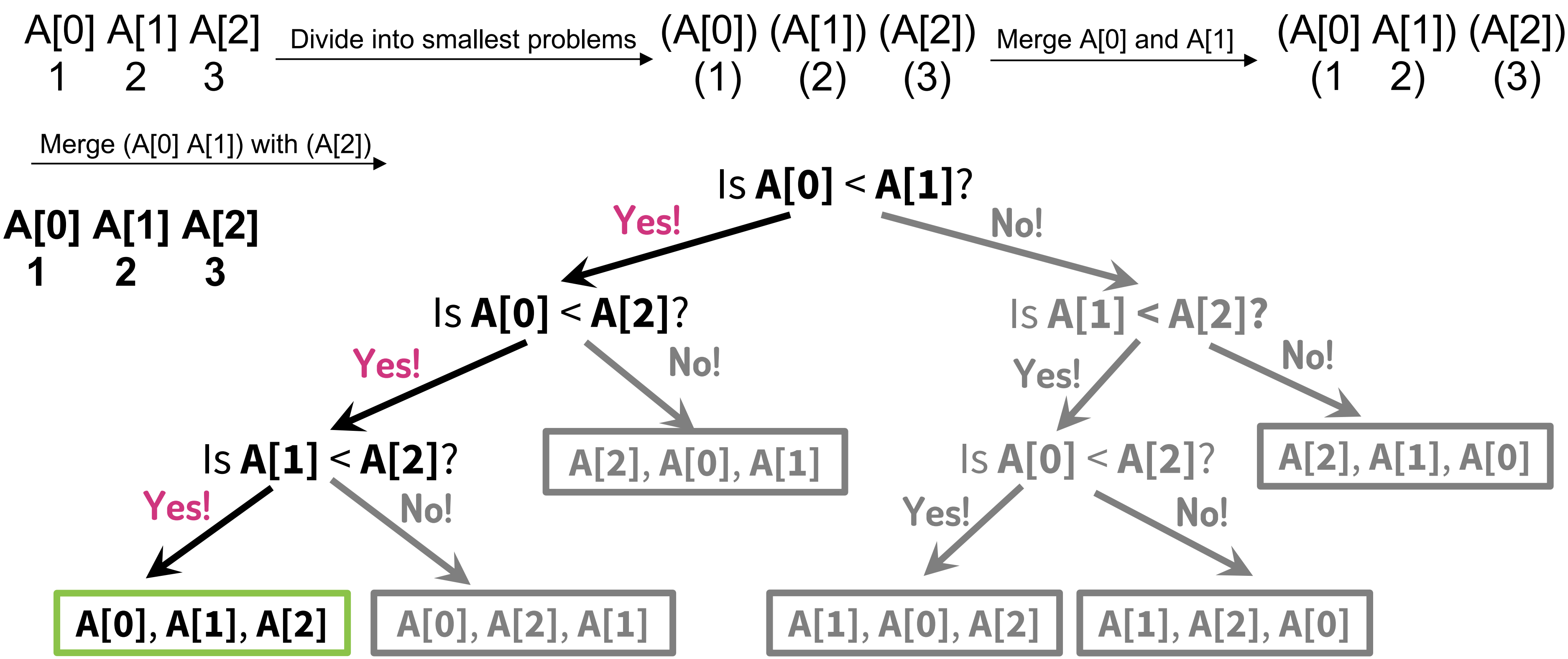
Suppose we want to sort three items in **A: A[0] A[1] A[2]**



Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**



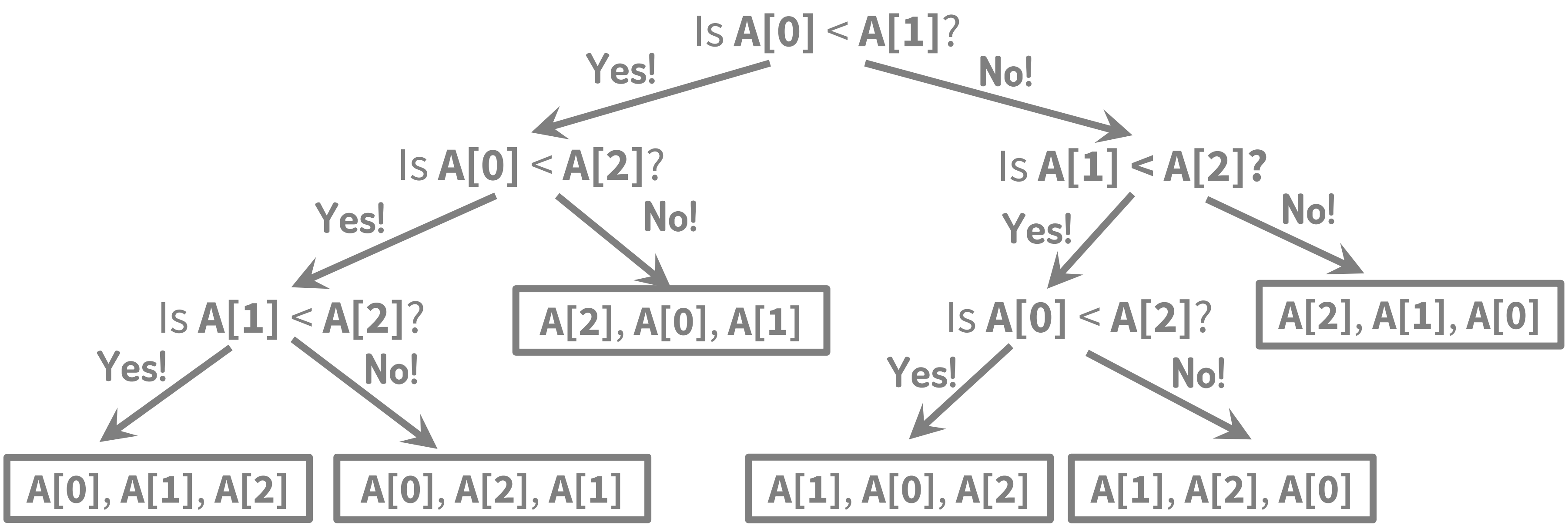
Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2] → Divide into smallest problems → (A[0]) (A[1]) (A[2])

3 2 1 → (3) (2) (1)



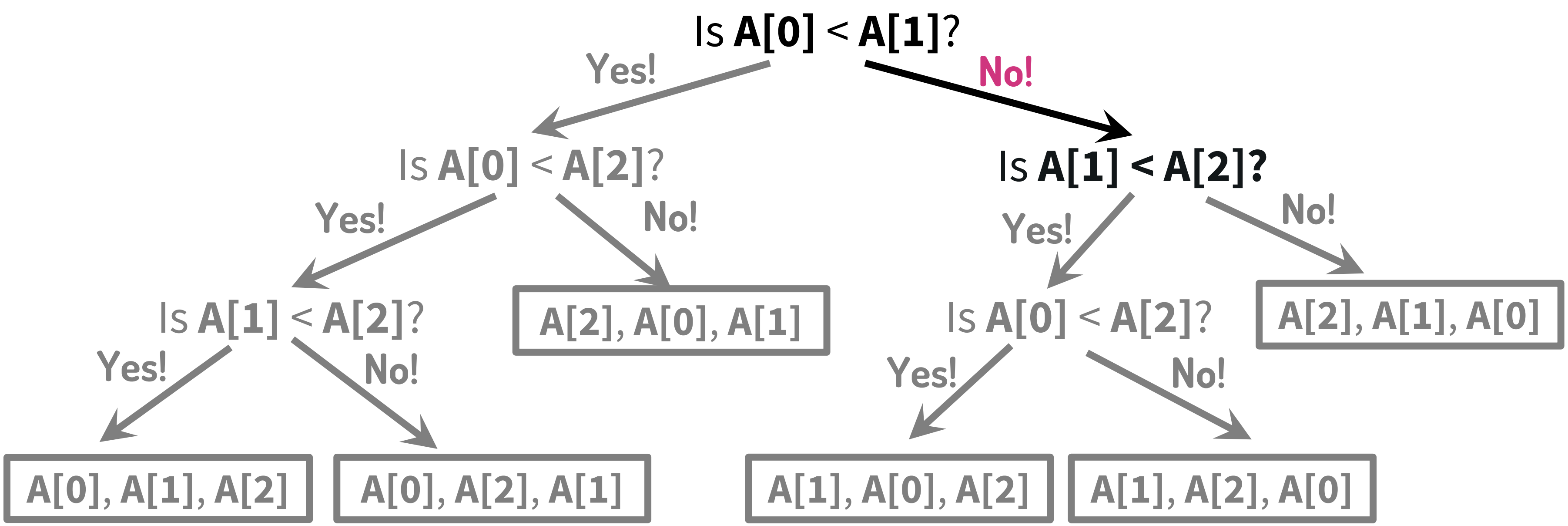
Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2] $\xrightarrow{\text{Divide into smallest problems}}$ (A[0]) (A[1]) (A[2]) $\xrightarrow{\text{Merge A[0] and A[1]}}$ (A[1] A[0]) (A[2])

3 2 1 → (3) (2) (1) → (2 3) (1)



Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

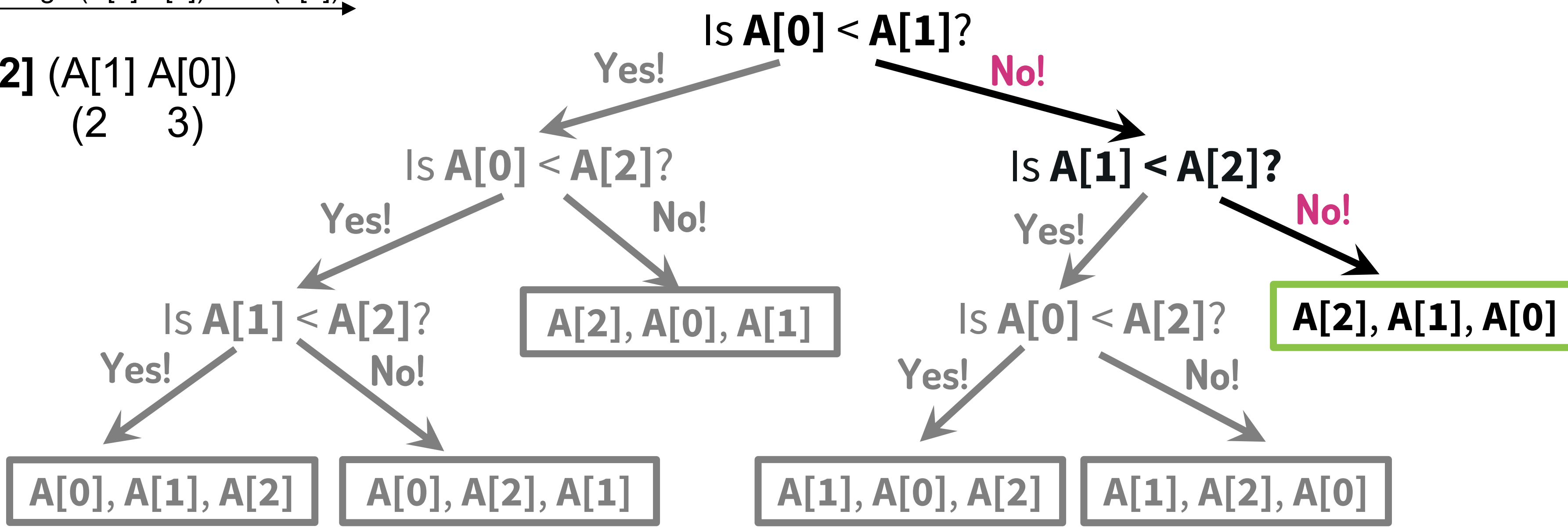
A[0] A[1] A[2] → Divide into smallest problems → (A[0]) (A[1]) (A[2]) → Merge A[0] and A[1] → (A[1] A[0]) (A[2])

3 2 1 → (3) (2) (1) → (2 3) (1)

Merge (A[1] A[0]) with (A[2]) →

A[2] (A[1] A[0])

1 (2 3)



Comparison-Based Sorting

The decision for merge sort

Suppose we want to sort three items in **A: A[0] A[1] A[2]**

A[0] A[1] A[2]
3 2 1

Divide into smallest problems →

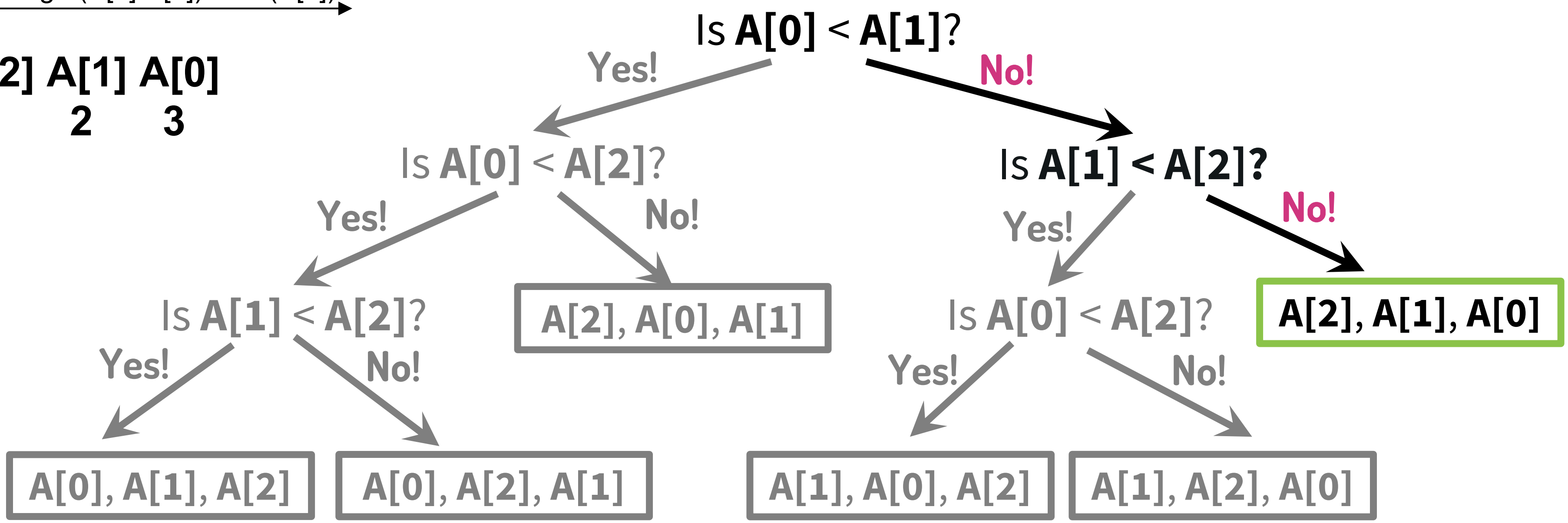
(A[0]) (A[1]) (A[2])
(3) (2) (1)

Merge A[0] and A[1] →

(A[1] A[0]) (A[2])
(2 3) (1)

Merge (A[1] A[0]) with (A[2]) →

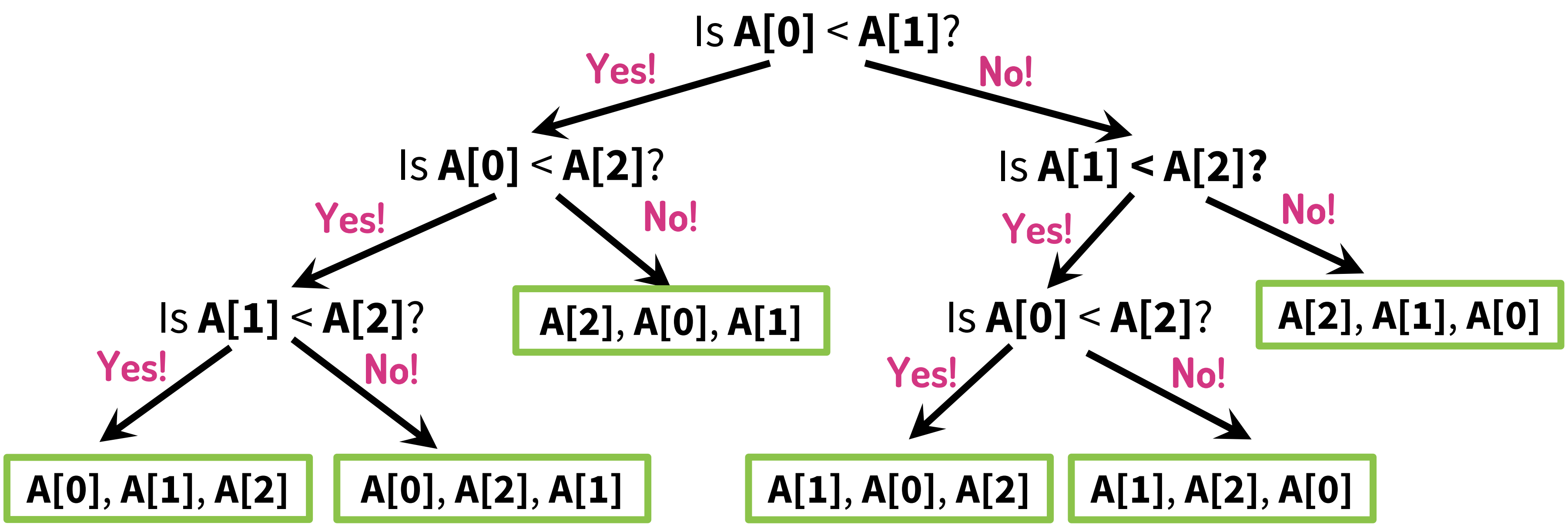
A[2] A[1] A[0]
1 2 3



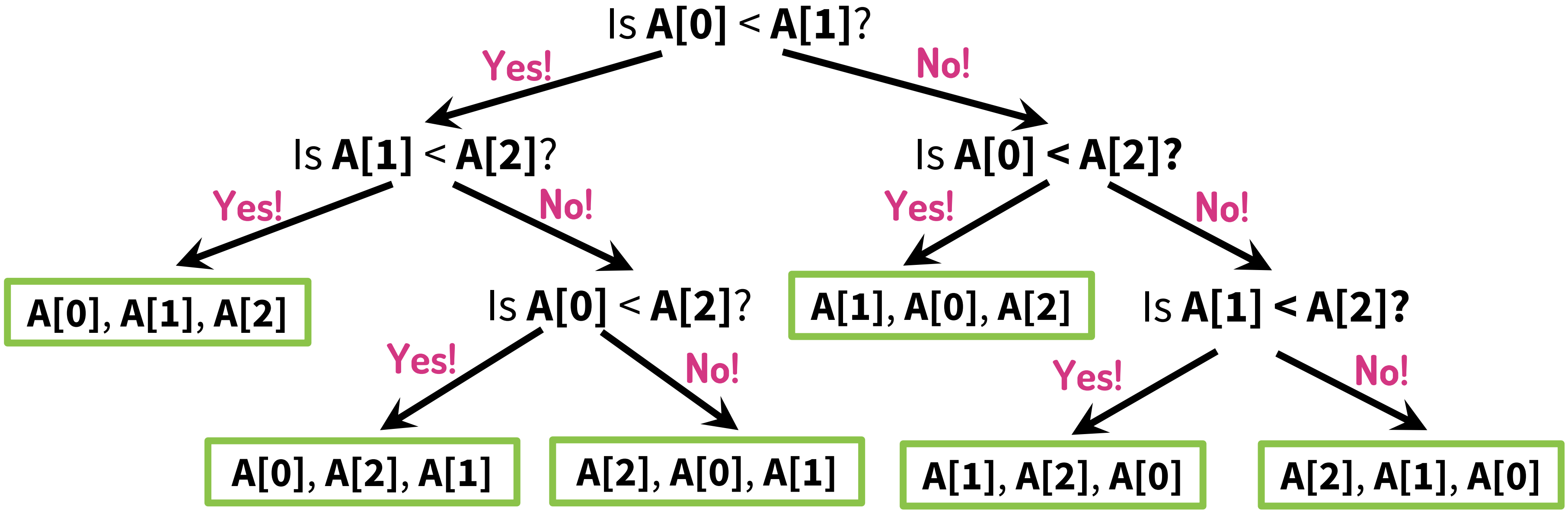
Comparison-Based Sorting

The decision for merge sort

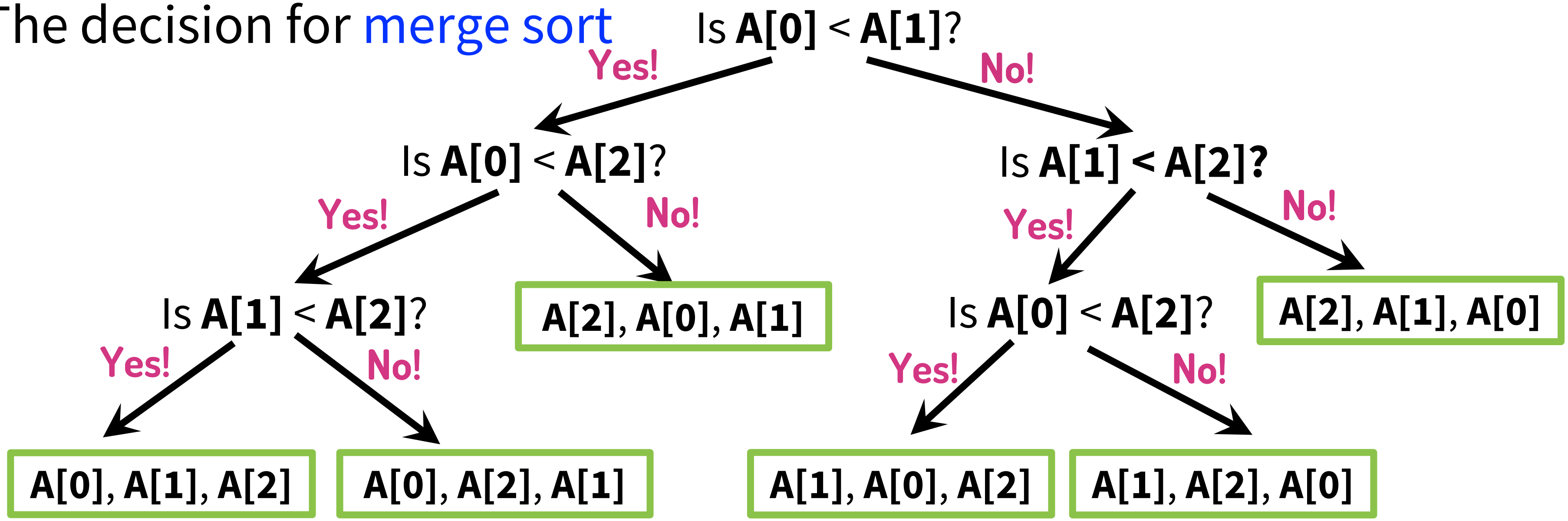
Different input is routed through different paths in the tree
Each leaf node corresponds to a possible ordering of the input
Time complexity is the worst case run time, so it corresponds to the longest path



The decision for insertion sort



The decision for merge sort

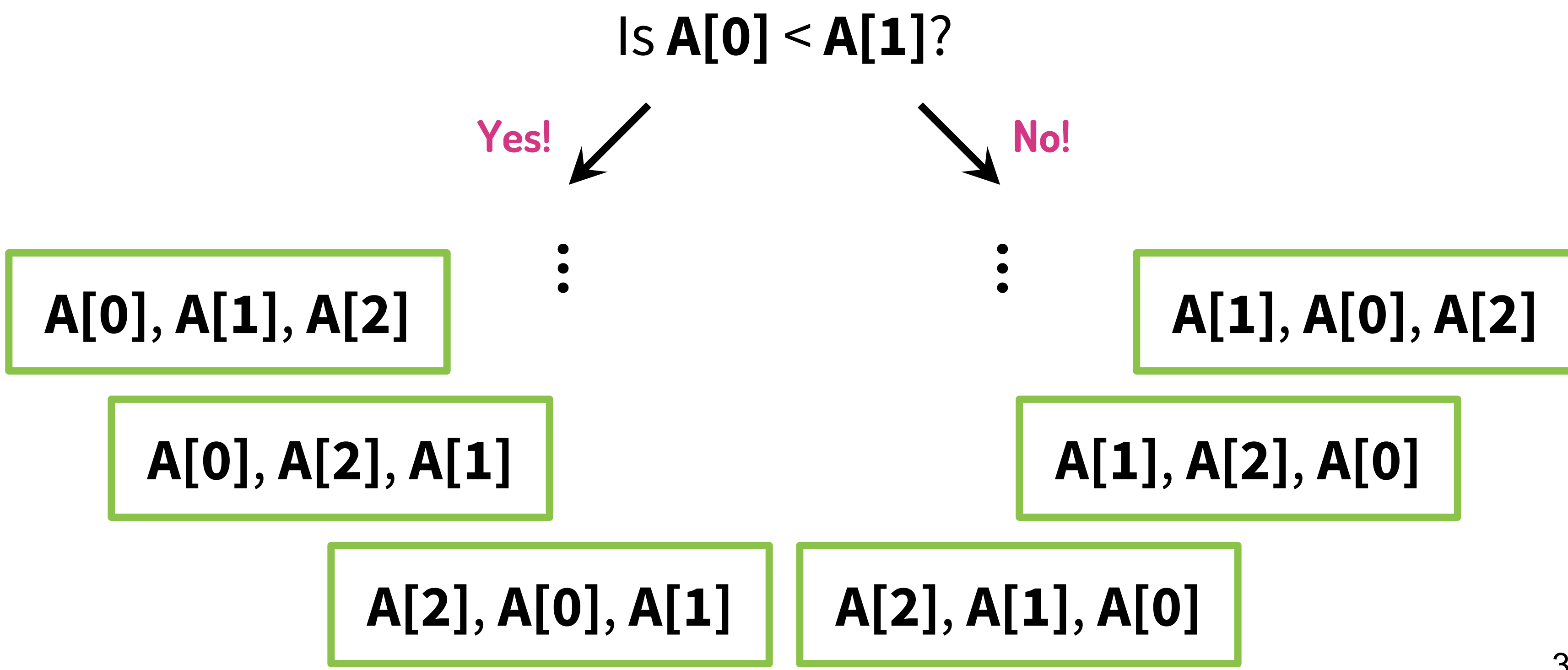


Tree structure could be different; leaf node are all possible number orderings; worst case runtime is longest path

Comparison-Based Sorting

The leaves are all of the possible orderings of the items.

The worst-case runtime must be at least
Ω(length of the longest path).



Comparison-Based Sorting

How long is the longest path?

- At least how many leaves must this decision tree have?
- What is the depth of the shallowest tree with this many leaves?

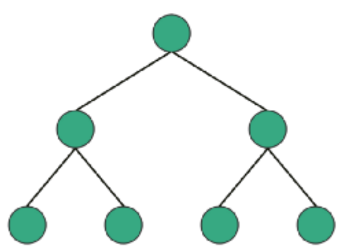
Comparison-Based Sorting

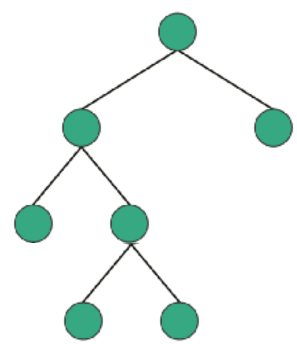
How long is the longest path?

- At least how many leaves must this decision tree have? **n!**
- What is the depth of the shallowest tree with this many leaves? **log(n!)**

The longest path is at least log(n!), so the worst-case runtime must be at least **Ω(log(n!))** = **Ω(n log(n))**.

To produce the same amount of leaves, the balanced binary tree gives the shortest depth (4 leaves as an example).





The Stirling's approximation: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Explain on board: Ω(log(n!)) = Ω(n log(n))

Comparison-Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time.

Proof:

Any deterministic comparison-based sorting algorithm can be represented as a decision tree with n! Leaves.

The worst-case runtime is at least the depth of the decision tree.

All decision trees with n! leaves have depth $\Omega(n \log(n))$.

Therefore, any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$-time

Beyond Comparisons

But then what's this nonsense about linear-time sorting algorithms?

- We achieve O(n) worst-runtime if we make assumptions on the input.
- e.g. They are integers that range from 0 to k-1.

Space-Time relationship in Algorithm Design

- Use more space (memory) in exchange for time (better efficiency)

Counting Sort

Counting sort

```
algorithm counting_sort(A, k):
  # A consists of n ints, ranging from
  # 0 to k-1
  counts = [0 * k]  # list of k zeros
  for a_i in A:
    counts[a_i] += 1
  result = []
  for a_i = 0 to length(counts)-1:
    append counts[a_i] a_i's to results
  return results
```

Runtime: $O(n+k)$

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	**0**	**3**	**1**	**1**	**3**	**1**	**0**

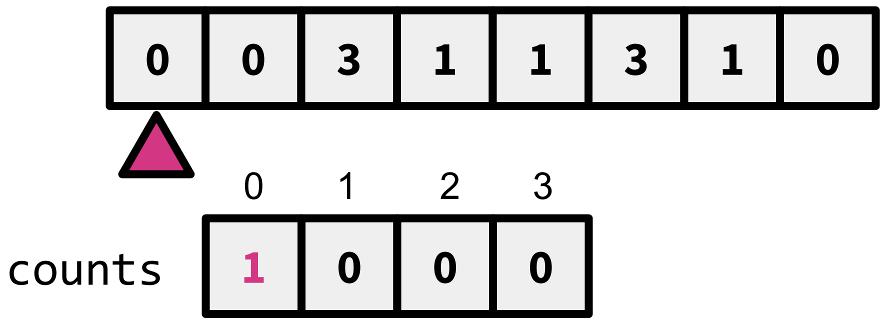
	0	1	2	3
`counts`	**0**	**0**	**0**	**0**

Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

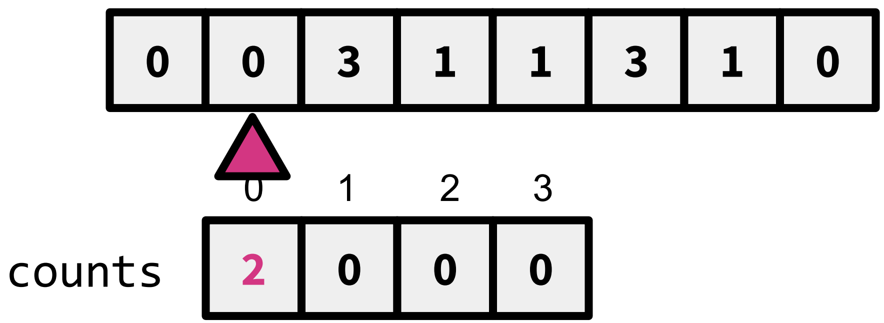


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

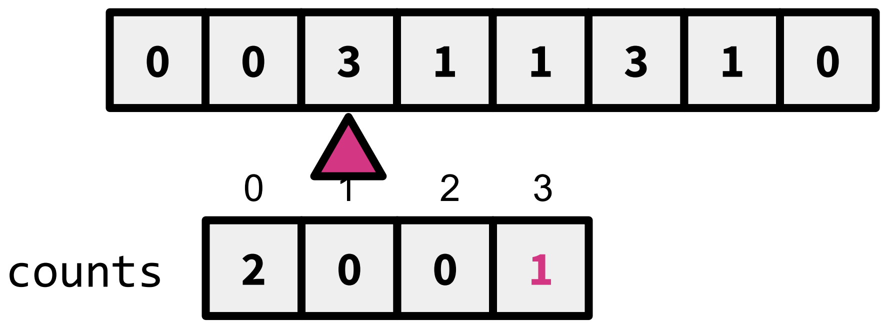


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

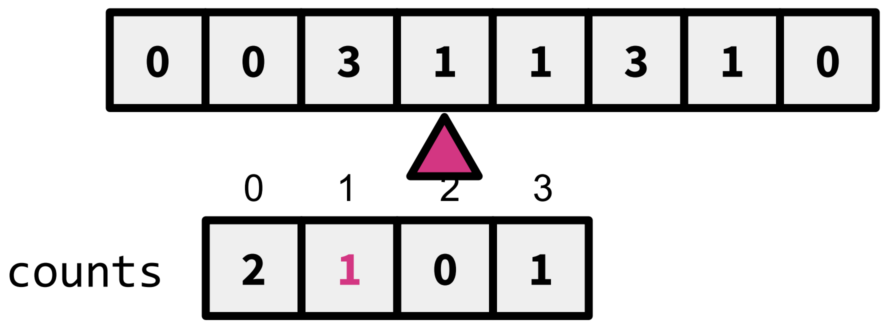


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

`counting_sort(A, 4)`

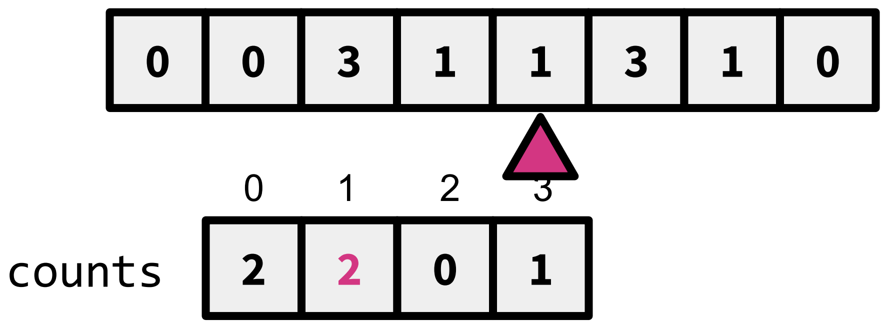


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

`counting_sort(A, 4)`

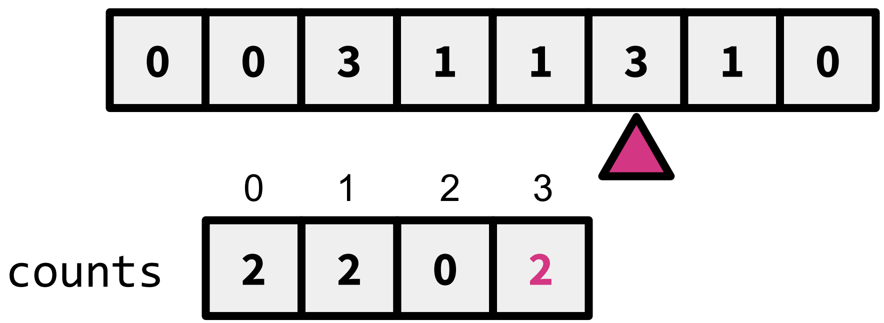


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

`counting_sort(A, 4)`

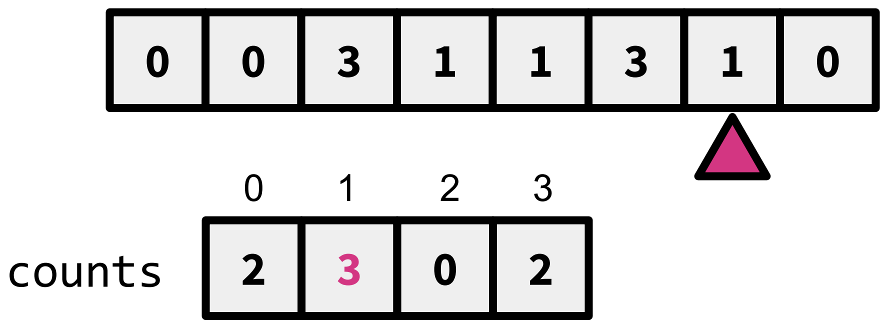


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

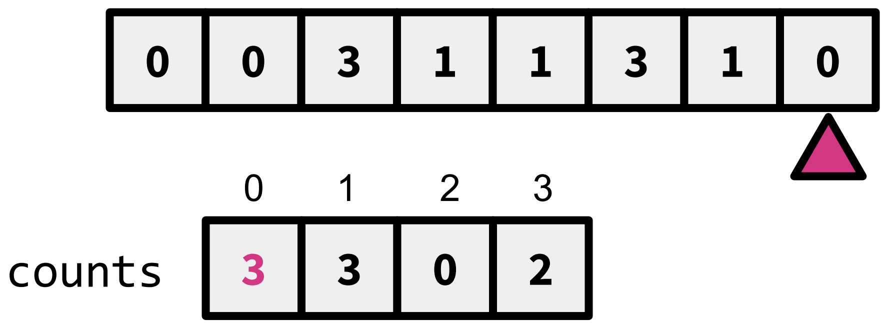


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

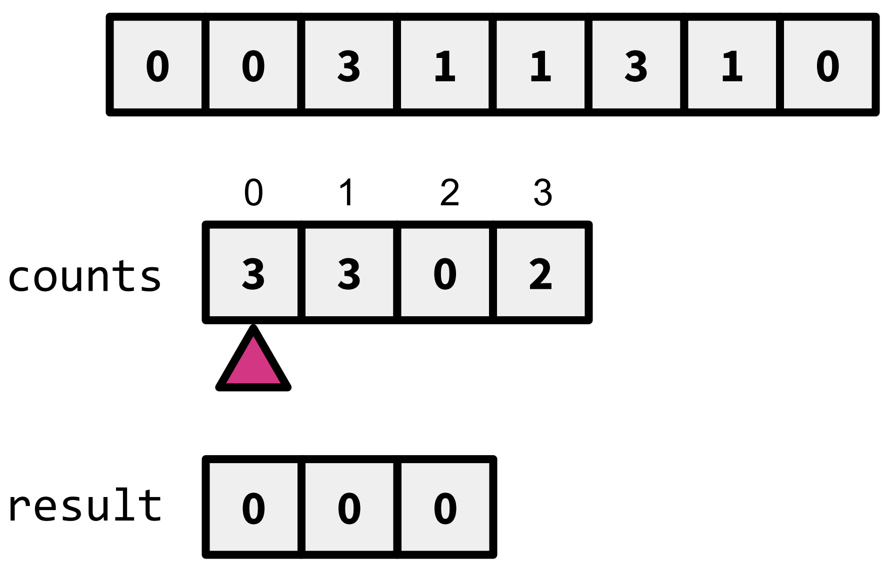


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

`counting_sort(A, 4)`



Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	0	3	1	1	3	1	0

counts

0	1	2	3
3	3	0	2

result

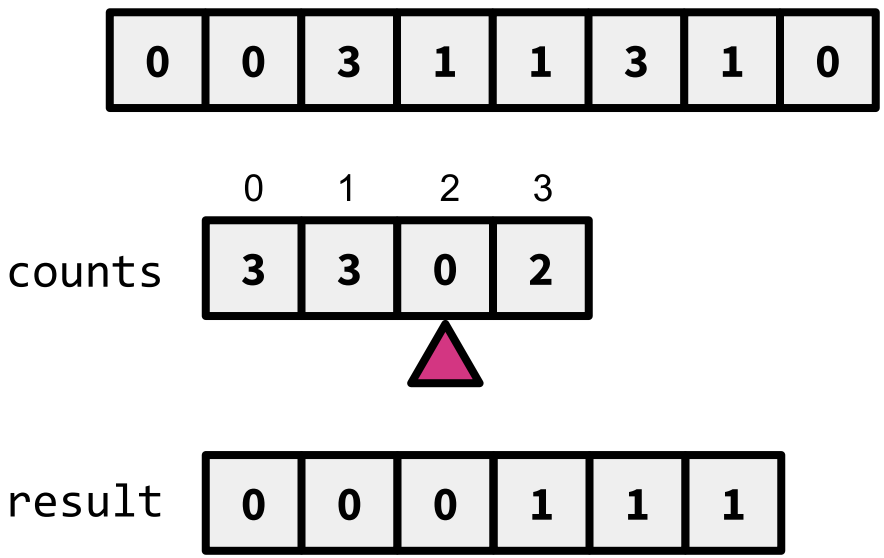
0	0	0	1	1	1

Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

`counting_sort(A, 4)`

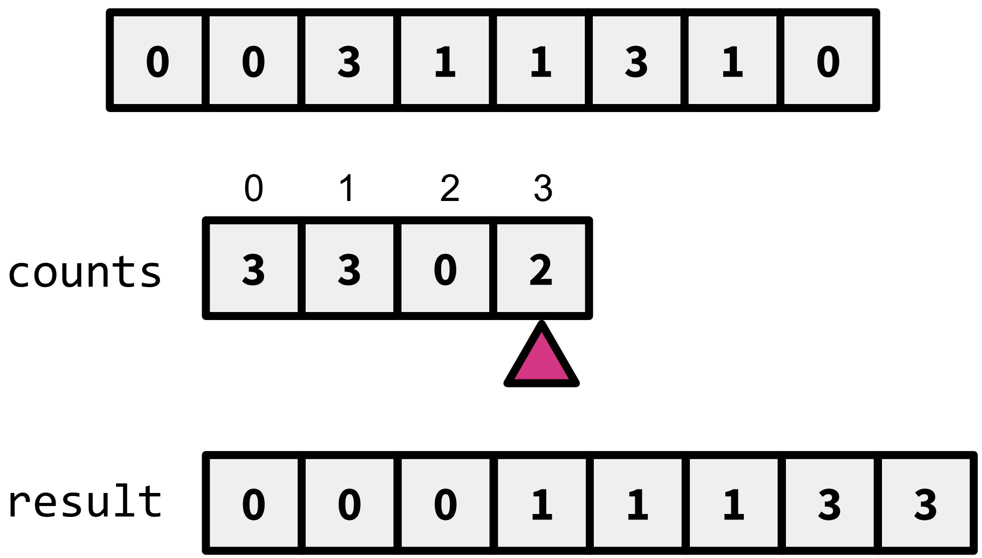


Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Counting sort

```
algorithm counting_sort(A, k):
  # A consists of n ints, ranging from
  # 0 to k-1
  counts = [0 * k]  # list of k zeros
  for a_i in A:
    counts[a_i] += 1
  result = []
  for a_i = 0 to length(counts)-1:
    append counts[a_i] a_i's to results
  return results
```

Runtime: $O(n+k)$

5-Minute Break

Bucket Sort

Bucket sort

```
algorithm bucket_sort(A, k, num_buckets):
  # A consists of n (key, value) pairs,
  # with keys ranging from 0 to k-1
  buckets = [[] * num_buckets]
  for key, value in A:
    buckets[get_bucket(key)].append((key, value))
  if num_buckets < k:
    for bucket in buckets:
      stable_sort(bucket) by their keys
  result = concatenate buckets by their values
  return result
```

Runtime: O(n+k) or O(nlogn)

Only guaranteed if
num_buckets >= k

Bucket sort

Two cases for k and num_buckets in `bucket_sort`:

(1) k ≤ num_buckets: At most one key per bucket, so buckets do not require an additional `stable_sort` to be sorted (similar to `counting_sort`).

(2) k > num_buckets: Maybe multiple keys per bucket, so buckets require an additional `stable_sort` to be sorted.

Note: Stable sort means the order of two equal numbers are kept as before.

Bucket sort

Two cases for k and num_buckets in `bucket_sort`:

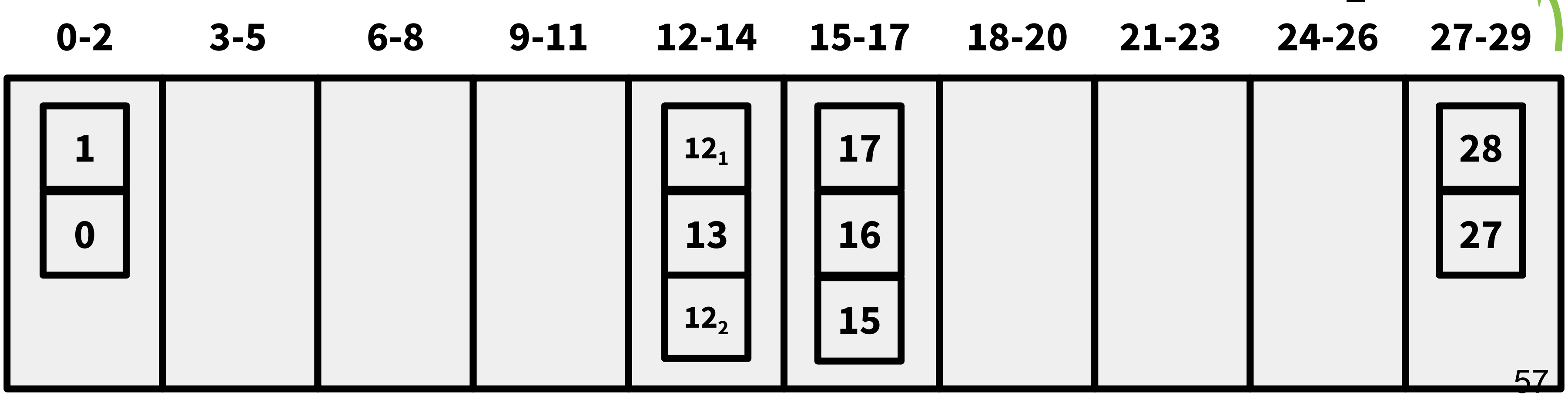
(1) k ≤ num_buckets: At most one key per bucket, so buckets do not require an additional `stable_sort` to be sorted (similar to `counting_sort`).

(2) k > num_buckets: Maybe multiple keys per bucket, so buckets require an additional `stable_sort` to be sorted.

Suppose k = 30 and num_buckets = 10. Then we group keys 0 to 2 in the same bucket, 3 to 5 in the same bucket, etc.

A= [17, 12_1, 13, 16, 12_2, 15, 1, 28, 0, 27] produces:

Only the keys in the (key, value) pairs are shown here, and all of the buckets require `stable_sort`.



What if we have 30 bukets?

Bucket sort

Two cases for k and num_buckets in `bucket_sort`:

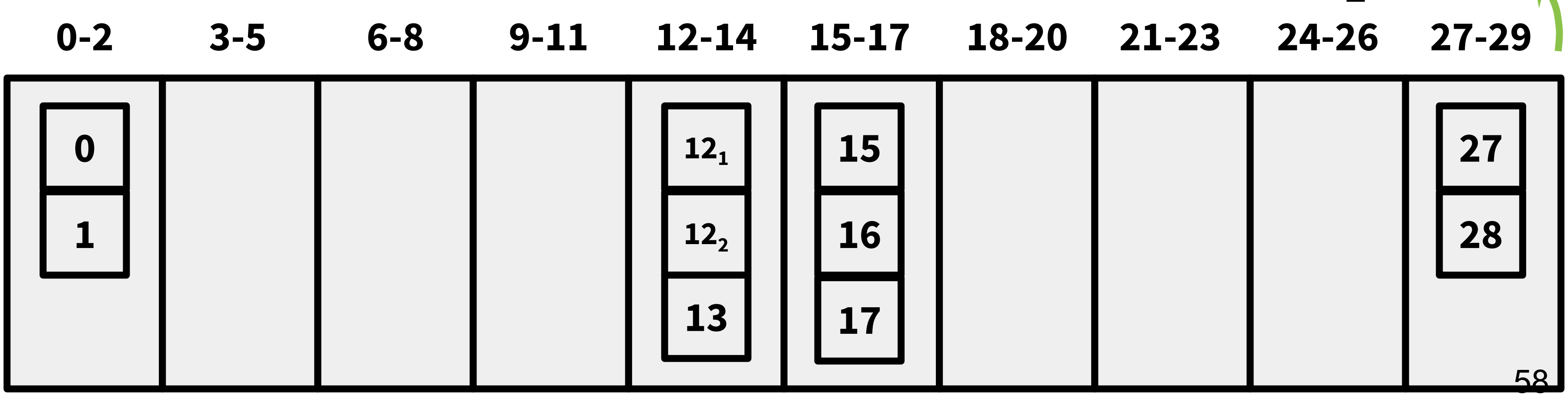
(1) k ≤ num_buckets: At most one key per bucket, so buckets do not require an additional `stable_sort` to be sorted (similar to `counting_sort`).

(2) k > num_buckets: Maybe multiple keys per bucket, so buckets require an additional `stable_sort` to be sorted.

Suppose k = 30 and num_buckets = 10. Then we group keys 0 to 2 in the same bucket, 3 to 5 in the same bucket, etc.

A= [17, 12_1, 13, 16, 12_2, 15, 1, 28, 0, 27] produces:

Only the keys in the (key, value) pairs are shown here, and all of the buckets require `stable_sort`.



What if we have 30 bukets?

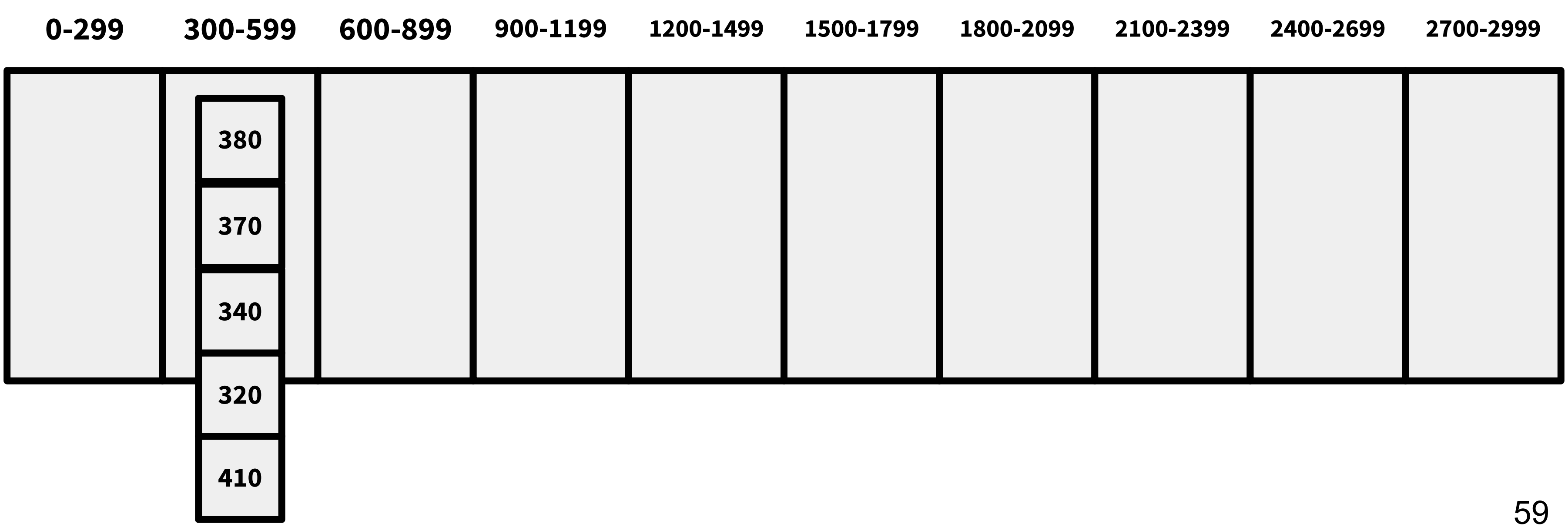
Bucket sort, case (2)

Why `O(nlogn)` in case (2)?

With multiple keys per bucket, a bucket might receive all of the inserted keys.

Suppose the `bucket_sort` caller specifies k = 3000 and num_buckets = 10, but then inserts elements all from the same bucket.

A = [380, 370, 340, 320, 410] would need to `stable_sort` all of the elements in the original list since they all fall in the same bucket.



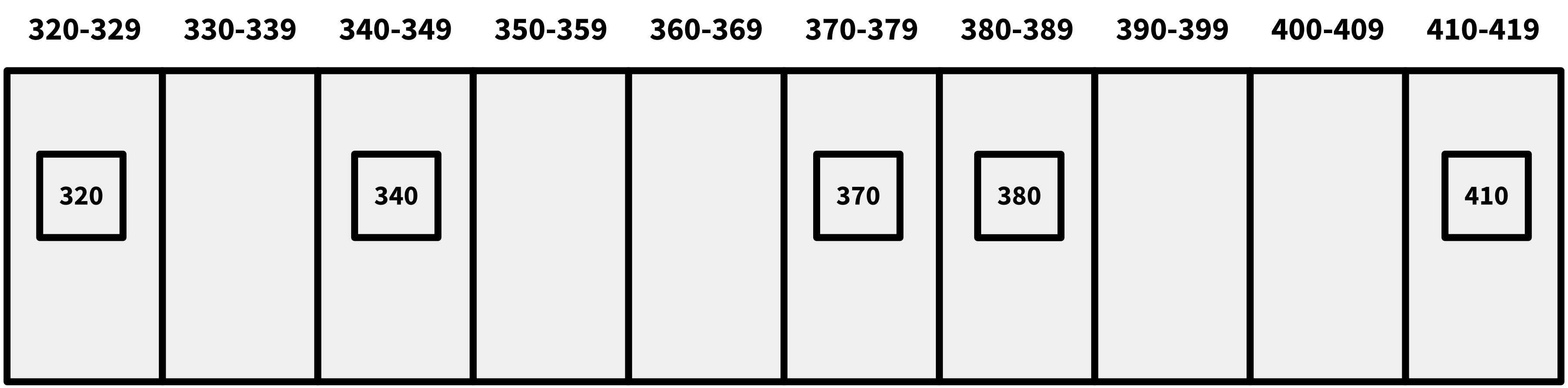
Bucket sort, case (2)

What to do in practice?

Find the exact smallest and largest number in the list (costs O(n)), then design more tight buckets to split numbers into the buckets as equally as possible.

min_value = 320, max_value = 410, number range = 90, we have num_buckets = 10, thus 10 values / bucket is enough to contain all values.

A = [380, 370, 340, 320, 410]



Radix sort

```
algorithm radix_sort(A, d, k):
  # A consists of n d-digit ints, with
  # digits ranging 0 -> k-1
  for j = 0 to d-1:
    A_j = A converted to (key, value) pairs, where
          key is the jth digit of value
    result = bucket_sort(A_j, k, k)
    A = result
  return A
```

Runtime: $O(d(n+k))$

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A	31	5	210	14	95	477	555	125

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A

031	005	210	014	095	477	555	125

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A

031	005	210	014	095	477	555	125

j

0

A_j

(1, 031)	(5, 005)	(0, 210)	(4, 014)	...	(5, 125)

result

210	031	014	005	095	555	125	477

Explain on board: using bucket sort to sort A_j with 10 buckets (bucket 0 to bucket 9)

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A	210	031	014	005	095	555	125	477

j	1

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A

210	031	014	005	095	555	125	477

j

1

A_j

(1, 210)	(3, 031)	(1, 014)	(0, 005)	...	(7, 477)

result

005	210	014	125	031	555	477	095

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A	005	210	014	125	031	555	477	095

j	2

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A

005	210	014	125	031	555	477	095

j

2

A_j

(0, 005)	(2, 210)	(0, 014)	(1, 125)	...	(0, 095)

result

005	014	031	095	125	210	477	555

Radix sort

Stable sort is very important in radix sort:

e.g., when the numbers have been sorted by digit-0, now sorting digit-1

```
radix_sort(A, 3, 10)
```

A

210	031	014	005	095	555	125	477

j

1

A_j

(1, 210)	(3, 031)	(1, 014)	(0, 005)	...	(7, 477)

result

005	210	014	125	031	555	477	095

Both 210 and 014 have value 1 on digit-1, but their digit-0 values (0 and 4) have been properly sorted in the previous round, this ordering should not be broken in this round.

Radix sort

Lemma: If **A** is sorted by its x least-significant digits by the end of iteration j = x of the loop, then **A** will be sorted by its x+1 least-significant digits by the end of iteration j = x+1 of the loop.

Proof:

Since `bucket_sort` is stable, the elements within each bucket are still sorted by their x least-significant digits.
(E.g., in the second round 210 and 014 are still sorted on 0 and 4, although the middle digit are both 1.)

`bucket_sort` sorts **A** by the x+1 digit of the elements, so the elements are sorted by their x+1 least-significant digits. ■

Radix sort

Theorem: Radix sort sorts the input list.

Proof:

At by the end of the 0-th iteration of the loop, **A** is sorted by its 0-th least-significant digits.

By our lemma, if **A** is sorted by its x least-significant digits by the end of iteration j = x of the loop, then **A** will be sorted by its x+1 least-significant digits by the end of iteration j = x+1 of the loop.

The loop terminates at the start of iteration j = d. The collection of d-digit integers in **A** are sorted by their d least-significant digits, which implies that **A** is sorted when the loop ends. ■

Summary

Sorting lower bounds

For any deterministic comparison-based sorting algorithm, the lower bound of computing time is $\Omega(n \log(n))$.

Linear Sorting Algorithms

If we know extra information about the input list, we may design linear-time sorting algorithms.

Counting Sort

Bucket Sort

Radix Sort

Summary

Sorting lower bounds

For any deterministic comparison-based sorting algorithm, the lower bound of computing time is $\Omega(n \log(n))$.

Linear Sorting Algorithms

If we know extra information about the input list, we may design linear-time sorting algorithms.

Counting Sort

Bucket Sort

Radix Sort

Acknowledgement: Part of the materials are adapted from Mary Wootter, Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.